Продвинутые алгоритмы и структуры данных

Марчелло Ла Рокка

Предисловие Луиса Серрано

MANNING

Advanced Algorithms and Data Structures

MARCELLO LA ROCCA

MANNING
SHELTER ISLAND

Продвинутые алгоритмы и структуры данных

МАРЧЕЛЛО ЛА РОККА

ПИТЕР®

Санкт-Петербург • Москва • Минск

2024

ББК 32.973.2-018
УДК 004.421
Л25

Ла Рокка Марчелло

Л25 Продвинутые алгоритмы и структуры данных. — СПб.: Питер, 2024. — 848 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1946-2

Познакомьтесь с самыми необходимыми алгоритмами решения сложных задач программирования в области анализа данных, машинного обучения и графов.

Вы постоянно сталкиваетесь с бесчисленными проблемами программирования, которые поначалу кажутся запутанными, трудными или нерешаемыми? Не отчаивайтесь! Многие из «новых» проблем уже имеют проверенные временем решения. Эффективные подходы к решению широкого спектра сложных задач кодирования легко адаптировать и применять в собственных приложениях, а при необходимости — создавать собственные структуры данных под конкретную задачу. Сбалансированное сочетание классических, продвинутых и новых алгоритмов обновит ваш инструментарий программирования, добавив в него новые перспективы и практические методы.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.421

ISBN 978-1617295485 англ.
ISBN 978-5-4461-1946-2

Краткое содержание

Часть II
Многомерные запросы

Часть III
Планарные графы и минимальное число пересечений

Приложения

Оглавление

Часть I
Улучшаем базовые структуры данных

Часть II
Многомерные запросы

Часть III
Планарные графы и минимальное число пересечений

Приложения

Не позволяйте страху мешать вам,
не бойтесь говорить «я не знаю»
или «я не понимаю» — ни один вопрос
не является глупым.

Маргарет Х. Гамильтон
(Margaret H. Hamilton)

Простота — великая добродетель, но
требуется напряженная работа, чтобы
достичь ее, и образование, чтобы ее понять.
И что еще хуже: сложность лучше продается.

Эдсгер Дейкстра (Edsger Dijkstra)

Cu `un fa nenti `un sbagghia nenti.

(Не ошибается только тот, кто ничего
не делает.)

Известная поговорка

Предисловие

Алгоритмы и структуры данных лежат на пересечении красивых теорий и передовых технологий. И если основой достижений в вычислениях является аппаратное обеспечение, то основой достижений в области алгоритмов и структур данных, безусловно, является разум. Многие из недавних достижений в области технологий воплотились в жизнь благодаря эффективному использованию вычислительных ресурсов и, как правило, разработке и реализации эффективных алгоритмов и структур данных.

Специалист по информатике, разработчик программного обеспечения, специалист по обработке данных и вообще любой профессионал, чья эффективная работа зависит от вычислений, должен свободно владеть языком алгоритмов и структур данных. Вот почему на собеседованиях в компаниях Кремниевой долины соискателям часто предлагают задачи из этой области.

Даже профессионалам сложно выучить и запомнить все детали всех существующих алгоритмов. Однако иметь хорошее представление о них, чтобы затем умело использовать в качестве строительных блоков при создании крупных проектов или при решении проблем, совершенно необходимо. Чтобы получить такое представление, специалисту надо освоить обширную теоретическую и математическую базу, получить прочные знания в области программирования и достичь глубокого понимания основных концепций.

Обрести все эти знания и развить понимание вам поможет книга «Продвинутые алгоритмы и структуры данных», в которой Марчелло сочетает строгость теории с универсальностью практики, разбавляя их увлекательными историями и примерами из реальной жизни.

Чтобы дать читателю четкое и ясное представление о некоторых наиболее важных алгоритмах и структурах данных, применяемых в различных отраслях и областях исследований, Марчелло использует весь свой обширный опыт работы в самых известных технологических компаниях, сочетая его с компетентностью исследователя в области машинного обучения.

В этой книге Марчелло смог приподнять завесу тайны над сложными темами, такими как алгоритмы отображения-свертки, генетические алгоритмы и имитация отжига, изложить их с той же легкостью, с какой он рассказал о деревьях и кучах. Я настоятельно рекомендую эту книгу всем желающим получить четкое представление об основах computer science. Единственный вопрос, который возник у меня после прочтения книги: «Где она была, когда я готовился к своему первому собеседованию в Кремниевой долине?»

Д-р Луи Серрано (Luis Serrano), PhD,
инженер-исследователь в Quantum Artificial Intelligence
Zapata Computing

Вступление

Я рад, что вы решили предпринять путешествие в мир структур данных и алгоритмов. Надеюсь, что для вас оно окажется таким же захватывающе интересным, как и для меня.

Обсуждаемые в книге темы способствовали прогрессу в области разработки программного обеспечения и изменили мир вокруг нас. Они все еще имеют большое значение, и вы, вероятно, каждый день взаимодействуете с десятками устройств и служб, использующих эти алгоритмы.

Изучение алгоритмов началось задолго до появления информатики. Например, алгоритму Эйлера и всей теории графов уже три столетия. Но этот срок ничто по сравнению с двумя тысячелетиями, прошедшими с тех пор, как впервые был изобретен алгоритм «сито Эратосфена» (используется для составления таблиц простых чисел).

И все же в течение долгого времени, уже после появления компьютеров, алгоритмами в основном занимались в академических кругах. Хотя были и некоторые редкие исключения, такие как компания Bell Labs, чьи научно-исследовательские группы добились больших успехов в этой сфере в период между 1950-ми и 1990-ми годами. Эти коллективы разработали динамическое программирование, алгоритм Беллмана — Форда и сверхточные нейронные сети для распознавания изображений.

К счастью, с тех пор многое изменилось. В новом тысячелетии крупнейшие компании-разработчики программного обеспечения все чаще привлекают к работе математиков и айтишников; появились новые области, такие как машинное обучение; после долгого застоя вновь возрос интерес к искусственному интеллекту (ИИ) и нейронным сетям. Ведущие специалисты из этих областей сейчас принадлежат к числу самых высокооплачиваемых профессионалов во всей индустрии разработки программного обеспечения.

Лично я увлекся алгоритмами еще в колледже. В старших классах я уже изучал алгоритмы поиска и сортировки, но, только познакомившись с деревьями и графами, осознал, какое большое значение может иметь эта тема, и по-настоящему влюбился в нее.

Это был первый раз, когда я почувствовал, что написание действующего кода не является ни единственной, ни главной целью. Как именно он работает, насколько эффективно он работает и насколько он ясен, ничуть не менее важно. (К сожалению, пришлось ждать еще пару лет, чтобы меня посетило такое же озарение по поводу тестирования.)

Чтобы написать эту книгу, мне пришлось приложить много усилий, гораздо больше, чем я планировал четыре года назад, когда впервые пришел со своими идеями к издателю. Она имеет весьма забавную историю (по крайней мере, так мне кажется

сейчас, когда я оглядываюсь назад), но не буду вас утомлять изложением рассказов о делах минувших. Просто знайте, что, пока я писал эту книгу, я прожил четыре года, сменил три работы, две страны и пять разных квартир!

Конечно, работа над книгой требует больших (командных) усилий, но, как правило, это компенсируется. Прежде всего такой вид деятельности означает начало роста, потому что неважно, как долго вы занимаетесь обсуждаемой темой и насколько хорошо ее знаете. Чтобы написать книгу на любую тему, придется подвергнуть сомнению все, что вы знаете, углубиться в детали, которые вы, возможно, упустили из виду, когда применяли свои знания ранее, и потратить много времени на исследование, переваривание и обработку идей, пока не появится полная уверенность в своей способности объяснить что-то тому, кто ничего не знает о предмете. Обычно хороший тест для оценки результата — привлечь члена своей семьи, который не работает в этой области, и заставить его выслушать вашу лекцию. Нужно лишь постараться найти кого-то очень терпеливого.

Благодарности

Я хотел бы поблагодарить за помощь на этом пути многих людей.

Спасибо моим редакторам в издательстве Manning: Хелен Стергиус (Helen Stergius) — перед ней стояла непростая задача помочь мне привести первоначальный вариант моей рукописи в соответствие со стандартами Manning — и Дженнифер Стаут (Jennifer Stout), работавшей над этим проектом последние пару лет. Без их помощи я не смог бы достичь желаемой цели. Мне было приятно работать с вами обеими. Спасибо за помощь, бесценные советы и терпение! Сотрудничая с вами, я многое узнал о том, как правильно писать книги, учить читателей и обращаться к ним. Вы помогли мне сделать эту книгу намного лучше.

Далее я хочу поблагодарить обоих научных редакторов, работавших над этой книгой.

Особая благодарность моему другу Аурелио Де Роса (Aurelio De Rosa). Мне выпала большая честь заполучить его на роль редактора блога о JavaScript, куда мы оба писали задолго до того, как появилась эта книга. Он вносил огромный вклад в нее, попутно помогая мне учиться писать технические тексты. Он был ее первым техническим редактором, задавал общее направление работы над книгой, обсуждал включенные в нее темы и просматривал код. Когда я искал издателя для своей рукописи, именно Аурелио познакомил меня с представителями Manning Publishing.

Огромное спасибо Артуру Зубареву (Arthur Zubarev), присоединившемуся к команде пару лет назад, давшему массу отличных советов и всегда высказывавшему справедливую критику. Работать с вами было удовольствием и честью.

Спасибо также всем остальным сотрудникам издательства Manning, трудившимся над опубликованием и продвижением книги: редактору проекта Дейдре Хиам (Deirdre Hiam), литературному редактору Кэти Петито (Katie Petito), корректору Мелоди Долаб (Melody Dolab) и редактору-рецензенту Александару Драгосавлевичу (Aleksandar Dragosavljevic). Это была действительно командная работа. И огромное спасибо рецензентам: Андрею Формиге (Andrei Formiga), Кристоферу Финку (Christoffer Fink), Кристоферу Хаупту (Christopher Haupt), Дэвиду Т. Кернсу (David T. Kerns), Эдду Мелендесу (Eddu Melendez), Джорджу Томасу (George Thomas), Джиму Амрейну (Jim Amrhein), Джону Монтгомери (John Montgomery), Лукасу Джерардо Теттаманти (Lucas Gerardo Tettamanti), Мачею Юрковски (Maciej Jurkowski), Маттео Гилдоне (Matteo Gildone), Майклу Дженсену (Michael Jensen), Майклу Кумму (Michael Kumm), Мишель Уильямсон (Michelle Williamson), Расмусу Киркби Стробаку (Rasmus Kirkeby Strøbæk), Рикардо Новелло (Riccardo Noviello), Ричу Уорду (Rich Ward), Ричарду Вогану (Richard Vaughan), Тимми Хосе (Timmy Jose), Тому Дженис (Tom Jenice), Урсуле Сервантес (Ursula Cervantes), Винсенту Забалле (Vincent Zaballa) и Захари Флейшманну (Zachary Fleischmann). Они уделяли время моей рукописи на различных этапах работы и давали бесценные отзывы о ней.

Я не могу ограничиться этим и не поблагодарить мою семью и друзей, потому что все они поддерживали и терпели меня в течение этих лет. Не помню, упоминал ли я об этом, но писать книгу было довольно трудно! Если когда-нибудь на вашу долю выпадет подобный опыт, вы увидите, что это будет означать отказ от многих вечеров, праздников и выходных, когда вместо того, чтобы пойти на озеро, попить пива с друзьями или просто позаниматься своими делами, нужно сосредоточиться и работать над рукописью. Я бы не справился без помощи и терпения самых близких мне людей, и теперь мне придется наверстывать все то время, которое мы упустили (и делать все те дела, которые я не сделал)!

Наконец, упомяну людей, сыгравших особую роль в моем становлении как специалиста в области computer science, кем я являюсь сегодня.

Прежде всего я хочу выразить благодарность бывшим преподавателям в Университете Катании. Здесь у меня есть возможность назвать только основных моих наставников: профессора Галло (Gallo), профессора Кутелло (Cutello) и профессора Паппалардо (Pappalardo), но на самом деле у меня очень длинный список имен. Я чувствую, что в наше время, когда полезность дипломов о высшем образовании ставится под сомнение и предпочтение отдается их более быстрым и практичным альтернативам, важно сказать о выдающейся работе, проделанной преподавателями в моей альма-матер. Массовые онлайн-курсы — отличный способ обучения и, бесспорно, шаг в направлении доступного и демократичного образования, становится возможным получить его независимо от местонахождения и социального статуса обучаемого. Но есть вещи, которые, как мне кажется, я бы упустил без учебы в колледже, — формирование и развитие критического отношения к окружающему, мышления ученого, направленного на проникновение в суть проблем и получение более широкого набора навыков, чем необходимый для работы минимум.

Должен признаться, что прежде я скептически относился к некоторым курсам в программе моего обучения, например к курсу линейной алгебры, потому что не понимал, как применять эти знания на практике. Но через несколько лет после выпуска я начал изучать машинное обучение, и тогда все эти знания пригодились и даже обеспечили меня некоторым преимуществом.

Наконец я хочу упомянуть человека, который ценой многих жертв вырастил меня и не скупился на поддержку, когда я учился: мою маму. Она обеспечила мне возможность следовать за моей путеводной звездой. Ее поддержка позволила мне реализовать все карьерные цели, включая и написание этой книги, поэтому в некотором смысле все достоинства данного опуса являются и маминой заслугой.

О книге

Я могу назвать по крайней мере три веские причины потратить время на изучение алгоритмов.

1. *Производительность*. Выбор правильного алгоритма может значительно ускорить приложение. Даже ограничившись чем-то простым, таким как поиск, можно заметить огромный выигрыш при переходе от линейного поиска к бинарному.
2. *Безопасность*. Если вы выбрали неправильный алгоритм, злоумышленник может воспользоваться этой оплошностью, чтобы вывести из строя ваш сервер, узел или приложение. Осуществить, к примеру, DoS-атаку на хеш[1], где хеш-таблица используется в качестве словаря для переменных, отправляемых с запросами *POST*, и перегрузить ее последовательностью, вызывающей огромное количество коллизий. Это, в свою очередь, не позволит серверу своевременно отвечать на запросы. Еще один интересный пример — когда ненадежные генераторы случайных чисел[2] однажды были использованы для взлома сайтов игры в покер онлайн.
3. *Эффективность разработки кода*. Если вы знаете, что для получения желаемого результата можно использовать готовые строительные блоки, то разработка будет двигаться вперед быстрее, а сам код будет чище.

И все же почему стоит прочитать именно *эту* книгу? Важная причина заключается в том, что я постарался подобрать и представить в ней передовые алгоритмы, которые помогут разработчикам улучшить свой код и справиться с отдельными проблемами современных систем.

Более того, я постарался использовать иной подход, чем принято в обычных пособиях для вузов. Здесь, как и в обычных учебниках, объясняется теория, лежащая в основе алгоритмов, но в то же время предпринимается попытка показать примеры практического применения каждого описанного алгоритма и ситуации, когда его целесообразно использовать.

В повседневной работе часто приходится иметь дело с небольшими специфическими частями более крупного (возможно, устаревшего) программного обеспечения. Однако иногда требуется создавать большие программные приложения с нуля. В таких ситуациях может оказаться полезной большая часть обсуждаемого здесь материала,

---

[1] http://ocert.org/advisories/ocert-2011-003.html.

[2] Подобные атаки описываются, например, в статье How we learned to cheat at online poker: A study in software security Эркина (Arkin), Брада (Brad) и др. на сайте developer.com, http://www.bluffnakedpoker.com/PDF/developer_gambling.pdf.

и я постараюсь дать вам представление о том, как писать чистый и быстрый код для решения ряда наиболее важных задач, с которыми можно столкнуться.

Благодаря новому подходу с перечислением в каждой главе задач, для решения которых можно воспользоваться той или иной структурой данных, вы получаете справочник и сможете обратиться к нему в любое время, когда понадобится выяснить наилучший способ повышения производительности приложения.

И последнее, но не менее важное: если вы читали книгу *Grokking Algorithms*[1] Адитьи Бхаргава (Aditya Y. Bhargava) и она вам понравилась, то мой опус сможете рассматривать как следующий шаг для желающих продолжить изучение алгоритмов. Если вы еще не читали ее, я настоятельно рекомендую сделать это: материал в ней объясняется языком, понятным любой аудитории. Не случайно эта книга пользуется большой популярностью. Я надеюсь, что и моя тоже покажется вам понятной и полезной.

КОМУ АДРЕСОВАНА ЭТА КНИГА

Большинство глав в книге ориентированы на читателей, имеющих базовые представления об алгоритмах, программировании и математике. Вам будет проще постигать материал, если вы уже прошли подготовительные этапы.

- Хорошо знаете математику, в частности алгебру. Это поможет вам понять теоретические разделы. Тем не менее в приложении Б содержится краткое введение в нотацию «О большое» и асимптотический анализ.
- Посещали вводный курс по информатике или даже по алгоритмам. Тогда вы уже знакомы с базовыми структурами данных, которые станут основой обсуждений в этой книге.
- Знакомы с такими понятиями, необходимыми для полного понимания структур данных, как:
  - основные структуры для хранения данных, такие как массивы и связные списки;
  - хеш-таблицы и хеширование;
  - деревья;
  - контейнеры (очереди и стеки);
  - основы рекурсии.

Тем, кому нужно освежить знания, предлагается краткий обзор этих базовых тем в приложении В.

---

[1] *Бхаргава А.* Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. — СПб.: Питер, 2019.

СТРУКТУРА КНИГИ

Книга начинается с главы 1, в ней кратко рассказывается, в какой форме будут обсуждаться разные темы, и дается представление об организации типичной главы.

В главе 1 также определяются алгоритмы и структуры данных, описываются различия между ними, на примере объясняется процесс рассмотрения разных алгоритмов, подходящих для решения задачи, и демонстрируется их применение для поиска лучших решений.

Начиная с главы 2, книга делится на три части и приложения. Каждая часть посвящена определенной теме, которая может быть абстрактной или затрагивать класс конкретных решаемых проблем.

Часть I посвящена продвинутым структурам данных, позволяющим усовершенствовать некоторые базовые операции, например отслеживание объектов или их групп. Читая эту часть книги, вы должны осознать и усвоить, что существует множество способов выполнения операций с данными и выбор лучшего из них зависит от контекста и требований.

В главе 2 представлен улучшенный вариант двоичных куч — *d*-куча. Здесь также описывается, как структурирован материал в каждой из глав этой части, каковы особенности его подачи.

В главе 3 продолжается исследование куч на примере *декартова дерева* — гибрида двоичного дерева поиска и кучи[1], который может пригодиться во многих ситуациях.

Глава 4 переключается на *фильтры Блума* — продвинутую форму хеш-таблицы, помогающую экономить память при сохранении постоянным времени поиска.

В главе 5 представлены несколько альтернативных структур данных для работы с *непересекающимися множествами*. Такие структуры формируют основу для создания сложных алгоритмов, имеющих вполне конкретное практическое применение.

В главе 6 представлены две структуры данных, которые показывают лучшую производительность, чем универсальные контейнеры, с точки зрения хранения и поиска строк: *префиксное дерево* и *базисное дерево* (также известное как компактное префиксное дерево).

В главе 7 демонстрируется применение структур данных, представленных в первых шести главах, для построения *LRU-кэша* — составной структуры данных для реализации эффективного кэширования. Здесь также подробно обсуждаются вариант применения *LFU-кэша* и проблемы синхронизации совместно используемых контейнеров в многопоточных окружениях.

---

[1] Отсюда его англоязычное название treap — комбинация слов tree («дерево») и heap («куча»). — *Примеч. пер.*

В части II представлен еще один особый случай поиска: работа с многомерными данными, их индексация и выполнение пространственных запросов. Здесь снова демонстрируется, как специальные структуры данных помогают повысить производительность по сравнению с базовыми алгоритмами поиска. Но, кроме того, в этой части затрагиваются другие важные темы: кластеризация, интенсивное использование пространственных запросов и распределенные вычисления, в частности модель программирования MapReduce.

Материал главы 8 знакомит с задачей *ближайшего соседа*.

В главе 9 описываются *k-мерные деревья*, применяемые для эффективного поиска в многомерных наборах данных.

Глава 10 содержит материал о более совершенных версиях этих деревьев: SS- и R-деревьях.

В главе 11 основное внимание уделяется приложениям *поиска ближайшего соседа* и подробно описывается пример их практического применения (поиск ближайшего склада, с которого товары должны отправляться покупателям).

В главе 12 представлены примеры использования эффективных алгоритмов поиска ближайших соседей: три алгоритма кластеризации, метод *k-средних*, *DBSCAN* и *OPTICS*.

Глава 13 завершает эту часть введением в MapReduce, мощную модель распределенных вычислений, и описанием ее применения к алгоритмам кластеризации, обсуждавшимся в главе 12.

Часть III посвящена единственной структуре данных — графам. На их примере представлено несколько методов оптимизации, лежащих в основе современных алгоритмов, применяемых в области искусственного интеллекта и больших данных.

В главе 14 дается краткое введение в *графы* и описываются основы этой фундаментальной структуры данных, необходимые для понимания части III. Здесь также демонстрируются алгоритмы *DFS*, *BFS*, алгоритмы *Дейкстры* и *A** и рассказывается, как их использовать для решения задачи «кратчайшего пути».

Глава 15 знакомит с *векторными представлениями* графов (graph embeddings), планарностью и ставит пару задач, которые мы попытаемся решить в последующих главах: поиск векторного представления графа с минимальным числом пересечений (*Minimum Crossing Number*, *MCN*) и красивое рисование графа.

Материал главы 16 описывает фундаментальный алгоритм машинного обучения — *градиентный спуск* и показывает, как можно его применить к графам и векторным представлениям.

Глава 17 основана на предыдущей главе и представляет *метод имитации отжига* (simulated annealing) — мощную технику оптимизации, которая пытается преодолеть недостатки градиентного спуска, когда приходится иметь дело с недифференцируемыми функциями или функциями с несколькими локальными минимумами.

Наконец, в главе 18 описываются *генетические алгоритмы* — еще более продвинутый метод оптимизации, помогающий ускорить сходимость.

Главы следуют друг за другом по определенной системе, которая направляет читателя на пути от постановки задачи к разработке структуры данных для ее решения и далее к реализации этого решения и выявлению потребностей во времени выполнения и памяти.

Но хватит об этом; более детально манера и порядок изложения материала описаны в главе 2.

В конце книги есть дополнительный раздел с приложениями. Информация, изложенная в них, очень важна для навигации по темам этой книги. Структура приложений отличается от структуры глав: в них не рассматриваются конкретные примеры, а приводится краткое освещение тем, с которыми читатель должен ознакомиться, прежде чем приступить к изучению глав. В большинстве приложений рассматриваются основные классы алгоритмов, и в некоторых главах есть ссылки на приложения, чтобы читатель мог воспользоваться ими в нужное время. Но вообще желательно хотя бы бегло просмотреть приложения, прежде чем переходить к главе 2.

В приложении А представлены обозначения, которые используются в псевдокоде для описания алгоритмов.

В приложении Б дается обзор *нотации «О большое»* и пространственно-временного анализа.

Приложения В и Г содержат перечень *основных структур данных*, которые используются в качестве строительных блоков для других, более продвинутых структур.

В приложении Д объясняется идея *рекурсии* — сложного метода программирования, с его помощью можно создавать более четкие и краткие определения алгоритмов, хотя и с некоторыми оговорками.

В приложении Е приводится определение *рандомизированных алгоритмов*, таких как *Монте-Карло* и *Лас-Вегас*. Здесь вы можете ознакомиться с задачами классификации и оценки рандомизированных решений.

О ПРИМЕРАХ ПРОГРАММНОГО КОДА

Для описания алгоритмов в книге широко используется псевдокод, поэтому от вас не требуется знать конкретный язык программирования.

Однако предполагается, что вы знакомы с базовыми концепциями программирования, такими как циклы и условия, не говоря уже о логических операторах, переменных и присваивании.

Чтобы упростить разбор псевдокода в книге, в приложении А приводится краткое руководство, описывающее синтаксис (точнее, псевдосинтаксис), который будет использоваться в главах. Желательно, чтобы вы заглянули в приложение А, прежде

чем приступать к главе 1, но если вы чувствуете себя достаточно уверенно, то можете сразу начать просматривать фрагменты кода и обращаться к приложению А, только если почувствуете, что синтаксис вам непонятен.

Кроме того, если вы знаете какой-то язык программирования, интересуетесь им или хотите увидеть воплощение представленных в книге идей в реальном выполняемом коде, можете обратиться к созданному нами репозиторию на GitHub[1]. Там найдете их реализации на нескольких языках (в том числе на Java, Python и JavaScript).

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction.

Об авторе

Марчелло Ла Рокка (Marcello La Rocca) — старший инженер-программист в Tundra.com. В фокусе его профессиональных интересов находятся графы, алгоритмы оптимизации, генетические алгоритмы, машинное обучение и квантовые вычисления. Он участвовал в разработке крупномасштабных веб-приложений и инфраструктур данных в таких компаниях, как Twitter, Microsoft и Apple, проводил прикладные исследования как в научной сфере, так и в промышленной. Марчелло придумал и реализовал алгоритм адаптивной сортировки NeatSort.

Иллюстрация на обложке

На обложке размещена иллюстрация под названием *Femme de Fiume*, или «Женщина из Риеки», из коллекции костюмов разных стран, собранной Жаком Грассе де Сен-Совером (Jacques Grasset de Saint-Sauveur) (1757–1810). Коллекция имеет название *Costumes de Différents Pays*, она была издана во Франции в 1797 году. Все иллюстрации в этом собрании тщательно прорисованы и раскрашены вручную. Богатое разнообразие представленных в коллекции Грассе де Сен-Совера костюмов с очевидностью демонстрирует, насколько далекими друг от друга в культурном отношении были города и регионы мира всего 200 лет назад. Изолированные друг от друга, люди говорили на разных языках и диалектах. В те времена по одежде легко было определить место жительства, профессию и социальное положение.

С тех пор многое изменилось, и богатое разнообразие региональных особенностей исчезло. Сейчас трудно отличить друг от друга жителей разных континентов, не говоря уже о разных городах, регионах или странах. Если взглянуть на это с оптимистической точки зрения, то окажется, что мы пожертвовали культурным и внешним разнообразием в угоду разнообразию частной жизни или ради более разнообразной и интересной интеллектуальной и технической деятельности.

В наше время, когда трудно отличить одну техническую книгу от другой, издательство Manning проявило инициативу и деловую сметку, украсив обложки книг изображениями, основанными на самобытности жизненного уклада народов двухвековой давности, подарив новую жизнь иллюстрациям из коллекции Жака Грассе де Сен-Совера.

1 Введение в структуры данных

В этой главе

- ✓ Почему стоит изучать структуры данных и алгоритмы.
- ✓ Разница между алгоритмами и структурами данных.
- ✓ Построение абстрактной задачи на основе конкретной.
- ✓ Переход от задач к решениям.

Итак, вы решили изучить алгоритмы и структуры данных. И это правильно!

Если вы все еще размышляете, есть ли такая необходимость, я надеюсь, что вводная глава развеет все сомнения и пробудит интерес к такой замечательной теме.

Зачем изучать алгоритмы? Короткий ответ: чтобы попытаться стать лучшим разработчиком программного обеспечения. Можно сказать, что знание структур данных и алгоритмов будет весьма полезным добавлением в ваш чемоданчик с инструментами.

Вы когда-нибудь слышали о молотке Маслоу, также известном как золотой молоток или закон инструмента? Это основанная на наблюдениях гипотеза, что люди, которым известен только один способ сделать что-либо, обычно применяют его во всевозможных ситуациях.

Если в чемоданчике с инструментами есть только молоток, возникнет соблазн относиться ко всему как к гвоздю. Если вы знаете только, как отсортировать список,

будет появляться искушение сортировать список задач каждый раз, когда в него добавляется новая задача или когда приходится выбирать, какую задачу взять в дальнейшую работу. Однако так вы не сумеете использовать понимание контекста, чтобы найти более эффективное решение.

Цель этой книги — дать вам множество инструментов, которые можно использовать при решении задач. Мы будем опираться на алгоритмы, обычно представленные в базовых курсах информатики в университетах (курс 101 по принятой в США кодировке), но я познакомлю вас и с более сложным материалом.

Прочитав книгу, вы должны научиться распознавать ситуации, в которых можно улучшить производительность кода, используя определенную структуру данных и/или алгоритм.

Конечно же, не стоит ставить перед собой цель запомнить наизусть все детали всех обсуждаемых структур данных. Скорее я постараюсь показать, как вникать в суть задач и где найти идеи алгоритмов, которые могут помочь в их решении. Книга также может послужить справочником, своего рода сборником рецептов, содержащим некоторые типовые сценарии и наиболее подходящие для этих сценариев структуры. В дальнейшем будет полезно классифицировать задачи по соответствию таким типовым сценариям.

Имейте в виду, что некоторые темы довольно сложны, и, когда мы углубимся в детали, может потребоваться прочесть один и тот же фрагмент текста несколько раз, чтобы все понять. Структура книги позволяет продемонстрировать несколько уровней углубленного анализа, при этом наиболее продвинутые разделы, как правило, сгруппированы в конце каждой главы, так что, если вы захотите получить только базовое понимание темы, сможете не углубляться в теорию.

1.1. СТРУКТУРЫ ДАННЫХ

Чтобы начать наше путешествие, сначала нужно договориться об общем языке для описания и оценки алгоритмов.

Способ описания алгоритмов довольно стандартен: алгоритмы описываются через принимаемые ими входные данные и возвращаемые ими выходные данные. Детали алгоритмов могут быть отражены с помощью псевдокода (без учета особенностей языков программирования) или фактического кода.

То же верно и для структур данных (СД), но для них этого недостаточно. Описывая структуру данных, необходимо также указывать и действия, которые можно с ней выполнить. Обычно каждое действие описывается как алгоритм со входом и выходом, но в дополнение к этому необходимо упомянуть и объяснить *побочные эффекты* для СД — изменения, которые действие может вызвать в самой структуре данных.

Чтобы осознать, что все это означает, нужно сначала правильно определить понятие «структура данных».

1.1.1. Определение структуры данных

Структура данных — это специальное решение для организации данных, которое предоставляет место для размещения элементов и возможность для их сохранения и извлечения[1].

Самый простой пример структуры данных — это массив. Так, массив символов обеспечивает хранилище для конечного числа символов и методы для извлечения каждого символа в зависимости от его позиции в массиве. На рис. 1.1 показано, как хранится `array = ['C', 'A', 'R']`: массив символов, в котором символы `C`, `A` и `R` размещаются таким образом, чтобы, например, элемент `array[1]` соответствовал значению `A`.

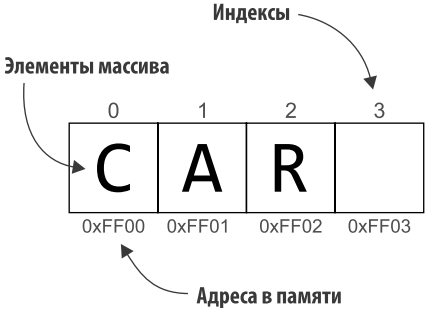


Рис. 1.1. Внутреннее представление массива (упрощенно). Каждому элементу массива на картинке соответствует байт памяти[2], адрес которого указан под ним. Над ним отображается индекс каждого элемента. Массив хранится как непрерывный блок памяти, и адрес каждого элемента может быть получен путем добавления его индекса в массиве к смещению первого элемента. Например, четвертый символ массива (array[3], на рисунке пустой) имеет адрес 0xFF00 + 3 = 0xFF03

Структуры данных могут быть абстрактными или конкретными. Между собой они соотносятся так, как указано в перечне ниже.

- *Абстрактный тип данных (АТД)* определяет операции, которые могут выполняться с некоторыми данными, и вычислительную сложность этих операций. Никаких подробностей о том, как хранятся данные или как используется физическая память, не приводится.
- *Структура данных (СД)* — это конкретная реализация спецификации, предоставляемой АТД.

---

[1] А именно, хотя бы один метод для добавления нового элемента в структуру и один метод либо для извлечения указанного элемента, либо для выполнения запроса к структуре данных.

[2] В современных архитектурах/языках элемент массива может соответствовать слову, а не байту, но для простоты предположим, что массив символов хранится как массив байтов.

ЧТО ТАКОЕ АТД

АТД можно представить как шаблон, задающий спецификации, а СД — перевод этих спецификаций в реальный код.

АТД определяется с точки зрения того, кто его использует. Пользователь описывает поведение АТД в терминах его возможных значений, возможных операций над ним, а также результатов и побочных эффектов этих операций.

Более формальное определение описывает АТД через набор типов, определенный тип из этого набора типов, набор функций и набор аксиом.

В свою очередь, структура данных, являющаяся конкретным представлением данных, описывается с точки зрения разработчика, а не пользователя.

Вернемся к примеру с массивом. Возможный АТД для статического массива — это, например, контейнер, который, во-первых, может хранить фиксированное количество элементов, каждый из которых ассоциирован с индексом (позицией элемента в массиве), и, во-вторых, предоставляет возможность обращаться к любому элементу, используя его позицию (произвольный доступ).

Обращаю ваше внимание на то, что при его реализации необходимо учесть:

- будет ли размер массива зафиксирован при создании, или его можно изменить;
- будет ли память для массива выделяться статически или динамически;
- будет ли массив содержать элементы одного типа или любого типа;
- будет ли он реализован как просто непрерывная область памяти или как объект;
- какие у него будут атрибуты?

Даже для такой базовой структуры данных, как массивы, разные языки решают перечисленные выше вопросы по-разному. Но все они обеспечивают, чтобы их версия массивов соответствовала описанному ранее АТД массива.

Еще одним хорошим примером, помогающим понять разницу между АТД и СД, может быть стек. Стеки описаны в приложениях В и Г, но я предполагаю, что вы слышали о них и раньше.

Возможное определение АТД стека таково: контейнер, который может хранить неопределенное количество элементов и удалять элементы по одному, начиная с самого последнего, в соответствии с порядком, обратным вставке.

В альтернативном его определении можно привести разбиение действий, которые могут быть выполнены с контейнером. Итак, стек — это контейнер, который поддерживает два основных метода:

- вставку элемента;
- удаление элемента: если стек не пуст, добавленный последним элемент будет возвращен и удален.

Это все еще высокоуровневое описание, но оно более четкое и модульное, чем предыдущее.

Оба определения достаточно абстрактны, чтобы их можно было легко обобщить, и это позволяет реализовать стек в широком диапазоне языков программирования, парадигм и систем[1].

Однако в какой-то момент придется перейти к конкретной реализации и обсудить детали, например те, которые приведены в следующем далеко не полном перечне.

- Где хранить элементы:
  - в массиве;
  - в связном списке;
  - в B-дереве на диске?
- Как мы будем отслеживать порядок вставки? (Это тесно связано с предыдущим вопросом.)
- Будет ли известен и фиксирован заранее максимальный размер стека?
- Может ли стек содержать элементы любого типа, или они все должны быть одного типа?
- Что произойдет, если вызвать удаление из пустого стека? (Например, это может быть возврат `null` или выдача ошибки?)

Можно продолжать задавать подобные вопросы, но, надеюсь, идея понятна.

1.1.2. Описание структуры данных

Наиболее важно при определении АТД перечислить набор операций, которые он разрешает. Это эквивалентно определению API[2], то есть контракта с клиентами.

Каждый раз, когда вам нужно описать структуру данных, можно выполнить несколько простых шагов, чтобы предоставить исчерпывающую и однозначную спецификацию.

1. Сначала указать ее API, уделяя особое внимание вводам и выводам методов.
2. Описать ее поведение на высоком уровне.
3. Подробно описать поведение ее конкретной реализации.
4. Проанализировать эффективность ее методов.

Именно такой рабочий процесс мы будем использовать для структур данных, представленных в книге, после того как опишем конкретный сценарий практического применения каждой структуры.

---

[1] В принципе, это не обязательно должно иметь отношение к информатике. Например, вы можете описать в качестве системы стопку папок, которые необходимо изучить, или — распространенный пример на уроках информатики — стопку посуды, которую нужно вымыть.

[2] Application programming interface (программный интерфейс приложения)

Во второй главе вместе с описанием первой структуры данных я более подробно объясню соглашения, которые используются для описания API.

1.1.3. Алгоритмы и структуры данных: так есть ли разница?

Это все же не совсем одно и то же; технически они не эквивалентны. Тем не менее иногда можно использовать эти два термина как синонимы и для краткости применять термин *«структура данных»* для обозначения как СД, так и всех соответствующих методов.

Есть много способов обозначить разницу между этими двумя терминами, но мне особенно нравится такая метафора: структуры данных подобны *существительным*, а алгоритмы — *глаголам*.

На мой взгляд, такое сравнение не только намекает на разницу в поведении структур и алгоритмов, но и неявно показывает взаимную зависимость между ними. Например, в английском языке, чтобы построить осмысленную фразу, нужны как существительные, так и глаголы: подлежащее (или дополнение) и выполненное им (или над ним) действие.

Структуры данных и алгоритмы взаимосвязаны; они нужны друг другу:

- структуры данных — это основа, способ организации области памяти для представления данных;
- алгоритмы — это процедуры, последовательность инструкций, направленных на преобразование данных.

Структуры данных были бы просто битами, хранящимися на микросхеме памяти, если бы не было алгоритмов их преобразования; и наоборот, без структур данных, с которыми можно было бы работать, большинство алгоритмов просто не существовало бы.

Более того, каждая структура данных сама неявно определяет алгоритмы, которые могут быть для нее выполнены, например методы для добавления, извлечения и удаления элементов в структуру данных и из нее.

Некоторые структуры данных фактически определены именно для того, чтобы один или несколько алгоритмов могли эффективно работать с ними. Вспомните о хеш-таблицах и поиске по ключу[1].

Итак, использование терминов «алгоритм» и «структура данных» как синонимов допустимо только потому, что в конкретном контексте одно подразумевает другое. Например, когда описывается СД, чтобы это описание было значимым и точным, обязательно нужно описать ее методы (то есть алгоритмы).

---

[1] Больше информации по этой теме вы найдете в приложении В.

1.2. ЦЕЛЕПОЛАГАНИЕ: ВАШИ ОЖИДАНИЯ ОТ ПРОЧТЕНИЯ ЭТОЙ КНИГИ

У вас может возникнуть такой вопрос: «Придется ли мне когда-нибудь писать собственные структуры данных?»

Весьма вероятно, что в большинстве случаев у вас будет возможность не создавать структуры данных с нуля. Сегодня нетрудно найти библиотеки, реализующие наиболее распространенные структуры данных на большинстве языков программирования, и обычно эти библиотеки написаны экспертами, которые знают, как оптимизировать производительность или позаботиться о безопасности.

По сути, основная цель книги — познакомить вас с обширным набором инструментов и научить распознавать возможности их использования для улучшения кода. Понимание внутреннего устройства этих инструментов, по крайней мере на высоком уровне, является важной частью процесса обучения. Тем не менее существуют определенные ситуации, в которых может потребоваться окунуться в код; например, если используется совершенно новый язык программирования, для которого не так много библиотек, или если нужно подстроить структуру данных под решение особого случая.

В конце концов, соберетесь ли вы писать собственную реализацию структур данных, зависит от многих факторов.

Например, от того, насколько продвинута структура данных, которая вам нужна, и насколько широко распространен язык, который вы используете.

Проиллюстрирую это на примере кластеризации.

Если вы работаете с популярным языком, например Java или Python, очень вероятно, что вы найдете множество надежных библиотек для метода *k*-средних, одного из простейших алгоритмов кластеризации.

Если же используется нишевой язык (предположим, вы экспериментируете с одним из недавно созданных, например с *Nim* или *Rust*), тогда может оказаться сложно найти библиотеку с открытым исходным кодом, реализованную командой, которая тщательно протестировала код и будет поддерживать библиотеку в дальнейшем.

Аналогично, если нужен продвинутый алгоритм кластеризации, такой как *DeLiClu*, будет непросто найти его реализации даже для Java или Python, такие, которым можно доверять в достаточной степени, чтобы запустить их в работу в производственной среде как часть вашего приложения.

Другая ситуация, в которой будет важно понимать внутреннее устройство этих алгоритмов, — возникновение необходимости подстройки одного из них. Например, может понадобиться оптимизировать его для среды реального времени, или нужно будет добавить в него какую-либо конкретную особенность (скажем, настроить его для одновременной работы и обеспечения безопасности потоков), или даже потребуется немного отличное от стандартного поведение.

Но даже если вы сосредоточитесь только на уяснении, когда и как использовать структуры данных, — а с этого начинается описание каждой из них, — уже одно это кардинально повлияет на ваше развитие как программиста, позволит поднять навыки кодирования на новый уровень. Воспользуюсь примером, чтобы показать важность алгоритмов в реальном мире, и заодно продемонстрирую способ описания алгоритмов, принятый в книге.

1.3. СОБИРАЕМ РЮКЗАК: СТРУКТУРЫ ДАННЫХ В РЕАЛЬНОМ МИРЕ

Поздравляем, вас выбрали для заселения первой марсианской колонии! В продуктовых магазинах на Марсе ситуация по-прежнему так себе... да и найти их трудновато. Так что готовьтесь, вам придется выращивать себе еду. Между тем на первые несколько месяцев вы будете обеспечены товарами для поддержания жизнедеятельности, их доставят на том же корабле, что и вас.

1.3.1. Абстрагирование задачи

Проблема в том, что ваши ящики не могут весить более 1000 кг, и это жесткое ограничение.

Задача усложняется тем, что вы можете выбирать только из ограниченного перечня продуктов, уже упакованных в коробки:

- картофель, 800 кг;
- рис, 300 кг;
- пшеничная мука, 400 кг;
- арахисовое масло, 20 кг;
- консервированные помидоры, 300 кг;
- фасоль, 300 кг;
- клубничное варенье, 50 кг.

Воду вы получите бесплатно, так что об этом беспокоиться не приходится. Что касается каждого продукта, то можно либо взять целую упаковку, либо не брать его вообще. Вы, конечно, хотели бы иметь в запасе какое-то разнообразие, а не тонну картошки (как было в «Марсианине»).

В то же время критически важно поддерживать собственное хорошее самочувствие и энергичность на протяжении всего пребывания на Марсе, поэтому главным фактором при выборе продуктов, которые отправятся с вами в путешествие, будет питательная ценность. Положим, ее хорошим индикатором будет общее число калорий. Рассмотрим табл. 1.1, которая показывает список доступных товаров с учетом новых данных.

Таблица 1.1. Перечень доступных товаров с указанием их веса и калорийности

Продукт	Вес, кг	Общее число калорий
Картофель	800	1 501 600
Пшеничная мука	400	1 444 000
Рис	300	1 122 000
Фасоль (консервированная)	300	690 000
Помидоры (консервированные)	300	237 000
Клубничное варенье	50	130 000
Арахисовое масло	20	117 800

Поскольку фактическое содержимое не имеет значения для принятия решения (несмотря на ваши понятные протесты, центр управления полетом очень тверд в этом вопросе), единственное, что принимается в рассмотрение, — это вес продуктов и общее число калорий в каждой из упаковок.

Следовательно, нашу задачу можно сформулировать в абстрактном виде так: «Выбрать любое количество предметов из набора, *не имея возможности взять часть какого-либо предмета*, при этом их общий вес не должен превысить 1000 кг и общее число калорий должно быть максимальным».

1.3.2. Поиск решений

Теперь, когда задача сформулирована, можно приступить к поиску решений. У вас может возникнуть соблазн начать упаковывать свой ящик с той коробки, в которой содержится наибольшее число калорий. Это будет ящик с картофелем весом 800 кг.

Но если вы это сделаете, ни рис, ни пшеничная мука не поместятся в ящик, а их общее число калорий намного превосходит любую другую комбинацию, которую можно создать в рамках оставшихся 200 кг. Лучшее, что можно получить при этой стратегии, — 1 749 400 калорий при выборе картофеля, клубничного джема и арахисового масла.

Итак, то, что выглядело бы наиболее естественным подходом, — *жадный*[1] алгоритм, который на каждом шаге выбирает кажущийся наилучшим здесь и сейчас вариант, — не работает. Эту задачу нужно обдумать более тщательно.

Время для мозгового штурма. Вы собираете свою логистическую команду и вместе ищете решение.

---

[1] Жадный алгоритм — это стратегия решения задач, которая находит оптимальное решение, делая локально оптимальный выбор на каждом шаге. Пользуясь им, можно найти лучшее решение только для небольшого подкласса задач, но его также можно использовать в качестве эвристики для поиска приближенных (неоптимальных) решений.

Вскоре кто-то предлагает смотреть не на общее число калорий, а на число калорий на килограмм. Таким образом, табл. 1.1 дополняется новым столбцом и соответствующим образом сортируется. Результат представлен в табл. 1.2.

Таблица 1.2. Список из табл. 1.1, отсортированный по числу калорий на килограмм

Продукт	Вес, кг	Общее число калорий	Число калорий на кг
Арахисовое масло	20	117 800	5890
Рис	300	1 122 000	3740
Пшеничная мука	400	1 444 000	3610
Клубничное варенье	50	130 000	2600
Фасоль (консервированная)	300	690 000	2300
Картофель	800	1 501 600	1877
Помидоры (консервированные)	300	237 000	790

Затем вы пытаетесь идти сверху вниз, выбирая еду с самым высоким соотношением калорий на единицу веса, и в итоге получаете набор из арахисового масла, риса, пшеничной муки и клубничного джема, что в сумме составляет 2 813 800 калорий.

Это намного лучше, чем первый результат. Понятно, что шаг был сделан в правильном направлении. Но если присмотреться, становится очевидным: добавление арахисового масла помешало нам взять фасоль, а это позволило бы еще больше увеличить общую ценность ящика. Хорошая новость в том, что вас по крайней мере не заставят соблюдать диету героя из упомянутого «Марсианина» — на сей раз картошка на Марс не отправится.

После еще нескольких часов мозгового штурма вы почти готовы сдаться, согласившись, что единственный способ решить эту задачу — проверить, можно ли получить лучшее решение, включив его или исключив каждый элемент из списка. Единственный известный вам способ — перечислить все возможные решения, отфильтровать те, которые превышают пороговое значение по весу, и выбрать лучшее из оставшихся. Это так называемый *алгоритм грубой силы (полный перебор)*, и из математики известно, что он очень дорогой.

Поскольку каждый предмет можно либо упаковать, либо оставить, то число возможных решений составляет $2^7 = 128$. Конечно, безрадостная перспектива: придется перебрать около сотни решений. Но через несколько часов, обессилев и уяснив, почему это называется алгоритмом грубой силы, вы почти заканчиваете решение этой задачи.

И тут появляются новости: кто-то позвонил из центра управления полетом и сообщил, что после жалоб некоторых будущих поселенцев в список были добавлены 25 новых продуктов питания, включая сахар, апельсины, сою и мармайт (не спрашивайте меня...).

После первой прикидки все приходят в ужас: теперь вам нужно проверить примерно 4 миллиарда различных комбинаций.

1.3.3. Спасительные алгоритмы

В этот момент становится ясно, что нужна компьютерная программа, чтобы обработать все числа и принять наилучшее решение.

Вы пишете ее сами в течение следующих нескольких часов, но она выполняется довольно долго. А тут выясняется, что, как бы вы ни хотели посадить всех колонистов на одну и ту же диету, это невозможно, потому что у некоторых из них есть аллергия: четверть состава не переносят глютен, и многие клянутся, что у них аллергия на мармайт. Так что придется запускать алгоритм еще на несколько прогонов, каждый раз учитывая индивидуальную аллергию. Что еще хуже, центр управления полетами рассматривает возможность добавления в список еще 30 пунктов, чтобы компенсировать то, что не получат люди с аллергией. Если это произойдет, в перечне продуктов окажется 62 возможных элемента и программе придется перебрать около миллиарда миллиардов возможных комбинаций. Программу запускают, проходит день, а она все еще работает, ни на чуть-чуть не приблизившись к завершению.

Вся команда готова сдаться и вернуться к картофельной диете, когда кто-то вспоминает, что в команде запуска есть человек, у которого на столе лежит учебник по алгоритмизации.

Вы вызываете его, и он сразу видит задачу такой, какая она есть на самом деле: это задача о рюкзаке 0-1. Есть и плохая новость: это NP-полная задача[1], а это означает, что ее трудно решить, поскольку не существует «быстрого» (то есть полиномиального по количеству элементов) известного алгоритма, который вычисляет оптимальное решение.

Однако есть и хорошие новости: существует псевдополиномиальное[2] решение, использующее *динамическое программирование*[3], а оно требует времени, пропорционального максимальной вместимости рюкзака. К счастью, вместимость ящика

---

[1] NP-полные задачи — это множество задач, для которых любое данное решение может быть быстро проверено (за полиномиальное время), но не существует известного эффективного способа найти правильное решение в первую очередь. NP-полные задачи по определению в настоящее время не могут быть решены за полиномиальное время на классической детерминированной машине (например, модель RAM, которую мы рассмотрим в следующей главе).

[2] Для псевдополиномиального алгоритма время работы в наихудшем случае зависит (полиномиально) от значения некоторых входных данных, а не только от их количества. Например, для задачи о рюкзаке 0-1 входными данными являются n элементов (с весом и стоимостью) и вместимость ранца C: полиномиальный алгоритм зависит только от числа n, в то время как псевдополиномиальный зависит также (или только) от значения C.

[3] Динамическое программирование — это стратегия решения сложных задач, обладающих определенными характеристиками, а именно рекурсивной структурой подзадач, при которой результат каждой подзадачи требуется использовать несколько раз при вычислении окончательного решения. Окончательное решение вычисляется путем разбиения задачи на набор более простых подзадач, решения каждой из этих подзадач только один раз и сохранения этих решений.

ограниченна, поэтому для решения необходимо количество шагов, равное количеству возможных значений заполненности емкости, умноженному на количество предметов. Таким образом, если наименьшим шагом изменения заполненности будет 1 кг, потребуется всего 1000 × 62 шага. Ого! Это намного лучше, чем 2^{62}! Фактически, как только вы выбираете правильный алгоритм, он находит лучшее решение за считаные секунды.

В какой-то момент вы готовы принять алгоритм как черный ящик и подключить его без лишних вопросов. Но ведь результат расчетов будет иметь решающее значение для вашего выживания... похоже, в этой ситуации стоило бы использовать более глубокие знания о том, как работает алгоритм.

В первоначальном примере оказывается, что наилучшая возможная комбинация — это рис, пшеничная мука и бобы, в общей сложности 3 256 000 калорий. Неплохой результат по сравнению с нашей первой попыткой, не так ли?

Возможно, вы и так уже успели найти наилучшую комбинацию, но если для начального примера, где всего семь элементов, поиск решения мог показаться слишком простым, попробуйте то же самое с сотнями различных продуктов в более близкой к реальному сценарию конфигурации и посмотрите, сколько лет придется искать лучшее решение вручную!

Казалось бы, получено удовлетворительное решение, и это лучшее, что можно найти с учетом ограничений.

1.3.4. Выходим за рамки (в буквальном смысле)

Но неожиданно в повествовании появляется настоящий эксперт по алгоритмам. Например, представьте, что выдающийся ученый посещает ваши объекты во время подготовки полета и его приглашают помочь с расчетом наилучшего маршрута для экономии топлива. Во время обеденного перерыва кто-то с гордостью рассказывает ему о том, как вы блестяще решили проблему с упаковкой товаров. Выслушав, он задает кажущийся наивным вопрос: почему, собственно, вы не можете изменить размер коробок?

Скорее всего, ответ будет либо «так было всегда», либо «товары поставляются уже упакованными, и изменение этого потребует времени и денег».

И тогда эксперт объяснит, что, если удалить ограничение на размер коробки, задача о рюкзаке 0-1, являющаяся NP-полной, станет задачей неограниченного рюкзака, для которой существует жадное решение с линейным временем[1], а оно обычно лучше, чем лучшее решение для версии 0-1.

В переводе на понятный человеку язык: можно превратить эту задачу в такую, которую легче решить и которая позволит наполнять ящики продуктами с максимально

[1] Линейное время, если список товаров уже отсортирован. В противном случае линейно-логарифмическое.

возможным количеством калорий. Теперь формулировка задачи становится такой: для данного набора предметов выберите любое подмножество предметов *или частей* предметов из него, чтобы их общий вес не превышал 1000 кг и, таким образом, чтобы общее количество калорий было максимальным.

И да, стоит потратить время на переупаковку всего, потому что мы получим серьезное улучшение.

В частности, если можно взять любую долю от первоначального веса для каждого продукта, то следует просто упаковывать продукты, начиная с продукта с наибольшим соотношением калорий на килограмм (в этом примере арахисовое масло), и, когда дойдет очередь до коробки, которая не поместится в оставшееся доступное пространство, переупаковать эту коробку, взяв ее часть, достаточную для заполнения этого пространства. Так что в конце концов придется переупаковывать даже не все товары, только один.

Кстати, здесь лучшим решением будет арахисовое масло, рис, пшеничная мука, клубничное варенье и 230 кг бобов, что в сумме составляет 3 342 800 калорий.

1.3.5. Счастливый конец

Итак, в рассказанной истории у будущих поселенцев на Марсе будет больше шансов на выживание, и они не впадут в депрессию из-за диеты, состоящей только из картофеля с арахисовым маслом и клубничного варенья.

С вычислительной точки зрения мы проделали путь от неправильных алгоритмов (жадные решения, которые сначала используют наибольшее значение или наибольшее отношение) к правильному, но невыполнимому алгоритму (решение методом грубой силы, перечисляющее все возможные комбинации) и, наконец, к умному решению, которое помогло организовать вычисления более эффективным образом.

Следующий шаг, столь же важный или даже более важный, заставил нас искать нестандартный подход, чтобы упростить проблему. В результате удалось снять некоторые ограничения и таким образом найти более простой алгоритм и лучшее решение. На самом деле это еще одно золотое правило: всегда тщательно изучайте свои требования, подвергайте их сомнению и по возможности старайтесь убрать ограничения, если это позволит получить по крайней мере не худшее решение за счет гораздо меньших усилий. В некоторых случаях так стоит поступать даже тогда, когда решение может незначительно ухудшиться. Конечно, при этом необходимо учитывать и другие аспекты (например, законы и безопасность на всевозможных уровнях), поэтому некоторые ограничения невозможно устранить.

В процессе описания алгоритмов, как было сказано в предыдущем разделе, я буду подробно описывать предлагаемое решение и предоставлять рекомендации по его реализации.

Для алгоритма динамического программирования решения задачи рюкзака 0-1 эти шаги будут опущены, потому что, во-первых, это алгоритм, а не структура данных

и, во-вторых, он уже подробно описан в литературе. Не говоря уже о том, что в данной главе работа с ним была использована просто для иллюстрации:

- того, как важно избегать неправильного выбора применяемых нами алгоритмов и структур данных;
- процесса, которому я буду следовать в следующих главах книги, вникая в задачу и рассуждая о путях ее решения.

РЕЗЮМЕ

- Алгоритмы нужно описывать с точки зрения их ввода, вывода и последовательности инструкций, которые будут обрабатывать ввод и производить ожидаемый вывод.
- Структура данных — это конкретная реализация абстрактного типа данных, состоящая из структуры для хранения данных как таковой и набора алгоритмов, которые ими управляют.
- Абстрагирование задачи означает формулирование четкой постановки задачи и только потом обсуждение ее решения.
- Эффективно упаковать рюкзак может быть непросто (особенно если вы планируете отправиться на Марс!), но с алгоритмами и правильной структурой данных нет (почти) ничего невозможного!

Часть I

Улучшаем базовые структуры данных

В первой части этой книги закладываются основы для более сложных тем, их мы будем рассматривать позже. Она посвящена исследованию расширенных структур данных, которые обеспечивают улучшение по сравнению с другими, более простыми структурами. Например, рассматривается улучшение двоичных куч или балансировка деревьев, а также решение таких задач, как отслеживание объектов или групп объектов.

На подобных примерах я продемонстрирую существование множества способов выполнения операций с данными и тот факт, что выбор лучшего из способов зависит от контекста и требований — нам как разработчикам следует с этим свыкнуться. Таким образом, при решении конкретной задачи необходимо проанализировать требования, изучить контекст и научиться подвергать сомнению собственные знания, чтобы найти лучшее решение в той или иной специфической ситуации.

В главе 2 представлен расширенный вариант двоичных куч, *d-куча*. В ней также описывается, как структурирован материал в каждой из глав этой части.

В главе 3 продолжается исследование расширенного использования куч на примере *декартова дерева* — гибрида между двоичным деревом поиска и кучей, он может оказаться полезен во многих ситуациях.

Глава 4 переключается на рассмотрение *фильтров Блума* — расширенную форму хеш-таблицы, которая может помочь сэкономить память, сохраняя время поиска амортизационно постоянным.

В главе 5 представлены несколько альтернативных структур данных, которые используются для отслеживания *непересекающихся множеств* и формируют основу для создания бесчисленных сложных алгоритмов, имеющих реальное практическое применение в некоторых ситуациях.

В главе 6 представлены две структуры данных, которые проявляют себя лучше, чем универсальные контейнеры, с точки зрения хранения и поиска строк: *префиксное дерево* и *базисное дерево* (также известное как компактное префиксное дерево).

В главе 7 демонстрируется использование структур данных, представленных в первых шести главах, с целью построения *LRU-кэша* — составной структуры данных, служащей для реализации эффективного кэширования. Кроме того, подробно обсуждаются вариант применения *LFU-кэша* и возникающие проблемы с синхронизацией разделяемых контейнеров в многопоточных средах.

2 Улучшаем очереди с приоритетом: d-кучи

В этой главе

- ✓ Задача обработки заданий на основе приоритетов.
- ✓ Использование очередей с приоритетом для решения этой задачи.
- ✓ Реализация очереди с приоритетом с помощью кучи.
- ✓ Понятие *d*-кучи, ее анализ.
- ✓ Сценарии, в которых использование куч улучшает производительность.

В предыдущей главе были введены некоторые основные понятия, касающиеся структур данных и приемов программирования, описана система изложения материала этой книги. И это, я надеюсь, вызвало ваш интерес. Теперь вам должно быть понятно, *почему* разработчикам нужно иметь представление о структурах данных.

В текущей главе продолжим развитие этих идей и начнем углубляться в детали, то есть осуществим постепенное погружение в проблематику книги. Сейчас выбрана тема, которая должна быть хорошо знакома читателям, имеющим хоть какой-то опыт работы с алгоритмами. Но в дополнение к ней я проведу обзор куч и выскажу некоторые новые соображения, касающиеся коэффициентов ветвления.

Соответственно, я предполагаю, что читатель знаком с рядом базовых концепций, традиционно изучаемых на базовом курсе информатики в университетах: с нотацией «О большое», моделью RAM и простыми структурами данных, такими как массивы,

списки и деревья. Эти концепции будут использоваться на протяжении всей книги для построения все более сложных структур и алгоритмов, и важно, чтобы вы имели возможность применять их осознанно и успешно изучить следующие главы. Вот почему перечисленные фундаментальные темы кратко изложены в приложениях к книге; обратитесь к ним или хотя бы бегло просмотрите, чтобы убедиться, что хорошо понимаете материал.

Итак, заложив фундамент, перейдем к основной теме и в разделе 2.1 рассмотрим структуру, которой будем придерживаться в каждой из последующих глав.

В разделе 2.2 приведена задача, предлагаемая к рассмотрению в этой главе (как эффективно обрабатывать события с приоритетом), а в разделе 2.3 приводятся возможные ее решения, включая очереди с приоритетом, и объясняется, почему последние лучше, чем более простые структуры данных.

Затем в разделе 2.4 описывается API[1] очереди с приоритетом и демонстрируется пример использования ее как черного ящика. Позже, в разделах 2.5 и 2.6, мы с вами углубимся в ее внутреннюю структуру: в разделе 2.5 подробно проанализируем, как работает *d*-куча, рассмотрим функциональность ее методов, в разделе 2.6 разберем реализацию *d*-кучи.

В разделах 2.7 и 2.8 описаны сценарии, в которых применение куч и очередей с приоритетом принципиально важно и позволяет ускорить выполнение приложений или других алгоритмов.

Наконец, в разделе 2.9 основное внимание уделяется понятию оптимального коэффициента ветвления для кучи. Этот раздел можно считать необязательным, но я предлагаю вам хотя бы попытаться прочитать его, чтобы получить более глубокое представление о том, как работает куча и почему может быть полезно выбрать троичную кучу (ternary heap) вместо двоичной или наоборот.

Эта глава может показаться пугающе длинной и сложной, но крепитесь! Прочитав ее, вы изучите много вопросов и заложите основу для понимания всей книги.

2.1. СТРУКТУРА ЭТОЙ ГЛАВЫ

Начиная с текущей главы, примем следующую схему рассмотрения структур данных.

Основной темой каждой главы будет практический сценарий — реальная задача, показывающая, как структура данных используется на практике. Кроме того, тут же будет приведено пошаговое объяснение того, как выполняются операции, представлены образцы кода, показывающие способы использования алгоритмов, на которых в дальнейшем мы сосредоточим внимание.

Итак, сначала представляем задачу, обычно решаемую с использованием материала, изложенного в главе.

---

[1] Application Programming Interface — программный интерфейс приложения.

Затем приводим один или несколько способов ее решения. У одной и той же задачи может быть несколько возможных решений, и если это так, следуют объяснения, когда и почему использование определенной структуры данных предпочтительно. На этом этапе структура данных по-прежнему выступает как черный ящик: пока надо сосредоточиться на том, как ее использовать, игнорируя детали реализации.

Только в следующем разделе мы начнем обсуждать, как работает структура данных. Здесь сосредоточимся на описании механизма, использовании рисунков и примеров псевдокода, чтобы прояснить для себя его функционирование.

После раздела с кодом мы обсуждаем сложные теоретические вопросы, такие как производительность или математические обоснования алгоритмов.

Обычно я также предоставляю список дополнительных практических применений представленных алгоритмов, хотя для большинства этих примеров приходится опускать кодирование ради экономии места.

2.2. ЗАДАЧА: КАК РАБОТАТЬ С ПРИОРИТЕТАМИ?

Первой задачей, которую мы решим, будет обработка заданий на основе приоритета. Это то, с чем все мы в некотором роде знакомы.

Задачу можно описать так: рассматривая набор заданий с разными приоритетами, определите, какое задание должно быть выполнено следующим.

Можно найти множество примеров в реальном мире, где мы, сознательно или нет, применяем методы, которые помогают решить, что делать дальше. Наша повседневная жизнь полна заданий; обычно порядок, в котором мы их выполняем, определяется тем, какие ограничения по времени и какая важность назначены этим заданиям.

Распространенным примером среды, в которой задания выполняются по приоритету, является отделение неотложной помощи, где вместо того, чтобы заниматься пациентами в соответствии с порядком поступления, их принимают в зависимости от критичности состояния. Если же взять пример более близкий к нашей предметной области, ИТ, то можно обнаружить множество инструментов и систем, основанных на том же принципе. Подумайте, например, о планировщике задач в операционной системе. Или, может быть, об используемом вами мобильном приложении для составления списка дел.

2.2.1. Приоритеты на практике: отслеживание ошибок

Я же для этой главы хотел бы привести в качестве примера систему мониторинга ошибок. Вы, наверное, уже знакомы с таким инструментом. При работе в командах нужна возможность отслеживать ошибки и контролировать задания, чтобы два человека не работали над одной и той же задачей и не дублировали усилия, и при этом следить, чтобы эти задачи решались в правильном порядке (каким бы он ни был в зависимости от принятой бизнес-модели).

Чтобы упростить пример, ограничимся рассмотрением инструмента отслеживания ошибок. В нем каждой ошибке назначен приоритет, выраженный в количестве дней, в течение которых ее необходимо исправить (меньшие числа означают более высокий приоритет). Кроме того, предположим, что ошибки независимы друг от друга, поэтому исправление одной ошибки не является обязательным условием для исправления другой.

Для конкретизации примера рассмотрим следующий неупорядоченный список ошибок для одностраничного веб-приложения.

Каждая ошибка будет выглядеть как кортеж:

```
<описание задачи, важность несоблюдения срока>
```

Например, список может выглядеть следующим образом.

Описание ошибки	Серьезность (1–10)
Загрузка страницы занимает более 2 с	7
Пользовательский интерфейс не работает в браузере X	9
Необязательное поле формы заблокировано при использовании браузера X в пятницу 13-го	1
Стиль CSS нарушает выравнивание	8
Стиль CSS вызывает смещение 1 пикселя в браузере X	5

Когда ресурсы (например, количество разработчиков) ограничены, возникает необходимость расставить приоритеты ошибок. Таким образом, одни ошибки требуют более срочного исправления, чем другие, что и находит отражение в назначенных им приоритетах.

Теперь предположим, что разработчик в команде закончил работу над своим текущим заданием. Он ищет в системе следующую ошибку, требующую исправления. Если бы этот список был статическим, система могла бы просто отсортировать ошибки один раз и вернуть их по порядку[1].

Описание ошибки	Серьезность (1–10)
Пользовательский интерфейс не работает в браузере X	9
Стиль CSS нарушает выравнивание	8
Загрузка страницы занимает более 2 с	7
Стиль CSS вызывает смещение 1 пикселя в браузере X	5
Необязательное поле формы заблокировано при использовании браузера X в пятницу 13-го	1

[1] Часто системы отслеживания ошибок связывают более низкие числа с более высоким приоритетом. Чтобы упростить обсуждение, мы вместо этого предположим, что более высокие числа означают более высокий приоритет.

Однако, как вы понимаете, в действительности все происходит не так. Во-первых, все время обнаруживаются новые ошибки, и поэтому в список будут добавляться элементы. Скажем, обнаружена неприятная ошибка шифрования, и по-хорошему, ее нужно было исправить еще вчера! Во-вторых, приоритет ошибок может со временем меняться. Например, ваш генеральный директор может решить, что компания нацелена на сегмент рынка, в котором в основном используется браузер X, и планируется большой выпуск функционала в следующую пятницу, которая приходится как раз на 13-е число. Поэтому действительно нужно исправить ошибку в последней позиции списка в течение пары дней.

Описание ошибки	Серьезность (1–10)
Незашифрованный пароль в базе данных	10
Пользовательский интерфейс не работает в браузере X	9
Необязательное поле формы заблокировано при использовании браузера X в пятницу 13-го	8
Стиль CSS нарушает выравнивание	8
Загрузка страницы занимает более 2 с	7
Стиль CSS вызывает смещение 1 пикселя в браузере X	5

2.3. ПРОСТОЕ РЕШЕНИЕ: ХРАНИМ ОТСОРТИРОВАННЫЙ СПИСОК

Разумеется, мы могли бы просто обновлять отсортированный список каждый раз, когда элемент вставляется, удаляется или изменяется. Этот подход может хорошо сработать, если такие операции выполняются нечасто и размер нашего списка невелик.

Любое из этих действий, по сути, потребовало бы изменения положения линейного числа элементов как в худшем, так и в среднем случае[1].

В простом примере такое, вероятно, было бы оправданно. Но если в списке будут миллионы или миллиарды элементов, то при таком подходе, скорее всего, возникнут проблемы.

2.3.1. От отсортированных списков к очередям с приоритетом

К счастью, существует решение получше. Описанная ситуация — идеальный пример для демонстрации одной из основных структур данных. Очередь с приоритетом сохранит частичный порядок элементов, гарантируя, что следующий элемент, возвращенный из очереди, будет иметь наивысший приоритет.

[1] Используя реализацию на основе массива, можно было бы найти позицию для элемента за логарифмическое время с помощью двоичного поиска. Но тогда придется переместить все элементы справа от точки вставки, чтобы освободить для нее место, а это в среднем требует линейного времени.

Отказавшись от требования полного упорядочения (оно в данном случае не обязательно, потому что мы решаем задачи только одну за другой), мы выигрываем в производительности: каждая операция с очередью теперь может требовать не более чем логарифмического времени.

Здесь уместно сделать очередное напоминание о важности точного формулирования требований перед выбором любого решения. Всегда нужно убеждаться, что мы не переусложняем постановку задачи и выдвинутые требования. Например, хранение полностью отсортированного списка элементов, притом что нам нужно лишь частичное упорядочение, приводит к неоправданным тратам ресурсов и усложняет код, что, в свою очередь, затрудняет его поддержку и масштабирование.

2.4. ОПИСАНИЕ API СТРУКТУРЫ ДАННЫХ: ОЧЕРЕДИ С ПРИОРИТЕТОМ

Прежде чем углубиться в тему главы, сделаем шаг назад.

Как объясняется в приложении B, каждую структуру данных можно разбить на несколько компонент более низкого уровня.

- *API* — API представляет собой контракт, который структура данных (СД) заключает с внешними клиентами. Он включает в себя определения методов и некоторые гарантии поведения методов, указанные в спецификации СД. Например, очередь с приоритетами (ОП) (табл. 2.1) предоставляет следующие методы и гарантии:
  - `top()` — возвращает и извлекает элемент с наивысшим приоритетом;
  - `peek()` — как и `top`, возвращает элемент с наивысшим приоритетом, но не извлекает его из очереди;
  - `insert(e, p)` — добавляет новый элемент `e` с приоритетом `p` в ОП;
  - `remove(e)` — удаляет элемент `e` из очереди;
  - `update(e, p)` — изменяет приоритет элемента `e`, устанавливая его (приоритет) равным `p`.
- *Инварианты* — (необязательные) внутренние свойства, которые всегда остаются истинными на протяжении всего жизненного цикла структуры данных. Например, отсортированный список будет иметь один инвариант: каждый элемент не больше своего преемника. Цель инвариантов — убедиться, что условия, необходимые для выполнения контракта с внешними клиентами, всегда выполняются. Они являются внутренним аналогом гарантий в API.
- *Модель данных* — для размещения данных. Это может быть просто последовательность ячеек памяти, список, дерево и т. д.
- *Алгоритмы* — внутренняя логика, которая используется для обновления структуры данных при обязательном соблюдении гарантии, что инварианты не будут нарушены.

Таблица 2.1. API и контракт очереди с приоритетом

Абстрактная структура данных: очередь с приоритетом	

API	`class PriorityQueue {`
	`top() → element`
	`peek() → element`
	`insert(element, priority)`
	`remove(element)`
	`update(element, newPriority)`
	`size() → int`
	`}`

Контракт с клиентом	Верхний элемент, возвращаемый очередью, всегда является элементом с наивысшим приоритетом, хранящимся в данный момент в очереди

В приложении B разъясняется, в чем разница между *абстрактной структурой данных* и *конкретной структурой данных*. Первая включает в себя API и инварианты, описывающие на высоком уровне, как клиенты будут взаимодействовать с ней, а также результаты и производительность операций. Последняя строится на принципах и API, выраженных абстрактным описанием, с добавлением конкретной реализации структуры и алгоритмов (модель данных и алгоритмы).

Именно так связаны между собой *очереди с приоритетами* и *кучи*. Очередь с приоритетом — это абстрактная структура данных, которая может быть реализована разными способами (в том числе в виде отсортированного списка). Куча — это конкретная реализация очереди с приоритетом, использующая массив для хранения элементов и специальные алгоритмы для обеспечения соблюдения инвариантов.

2.4.1. Примеры работы очереди с приоритетом

Допустим, очередь с приоритетом уже создана. Она может находиться в сторонней или в стандартной библиотеке (многие языки, такие как C++ или Scala, предоставляют реализацию очередей с приоритетом в своей стандартной библиотеке для контейнеров).

На данный момент не нужно знать внутренности библиотеки; вам просто нужно следовать ее общедоступному API и использовать его, подразумевая, что он правильно реализован. Это подход «черного ящика» (рис. 2.1).

Например, предположим, что мы добавляем ошибки в нашу ОП в том же порядке, что и раньше.

Описание ошибки	**Серьезность (1–10)**
Загрузка страницы занимает более 2 с	7
Пользовательский интерфейс не работает в браузере X	9
Необязательное поле формы заблокировано при использовании браузера X в пятницу 13-го	1
Стиль CSS нарушает выравнивание	8
Стиль CSS вызывает смещение 1 пикселя в браузере X	5

Если возвращать задачи в том же порядке, в котором мы их вставляли, можно реализовать простую очередь (см. рис. 2.2, на котором кратко показано, как работает очередь, и приложение В, где дается описание основных контейнеров). Вместо этого предположим, что имеем очередь с приоритетом, содержащую те же пять элементов; мы до сих пор не знаем внутренностей ОП, но можем направлять ей запросы посредством API.

Рис. 2.1. Представление очереди с приоритетом в виде черного ящика. Если используется реализация очереди с приоритетом, предоставленная сторонней библиотекой (или взятая из стандартных библиотек), и мы уверены, что эта реализация корректна, то можно использовать ее как черный ящик. Другими словами, в этом случае мы можем игнорировать ее внутренности и просто взаимодействовать с ней через ее API

Например, можем проверить, сколько элементов она содержит, и даже взглянуть на тот, что находится вверху (см. рис. 2.1). Или напрямую попросить ее вернуть нам верхний элемент (имеющий наивысший приоритет) и удалить его из очереди.

Если после вставки пяти элементов на рис. 2.1 вызвать `top`, будет возвращен элемент UI breaks on browser X (Пользовательский интерфейс не работает в браузере X), а размер очереди станет равным 4. Если снова вызвать `top`, следующим возвратится CSS style causes misalignment (Стиль CSS нарушает выравнивание) и размер станет равным 3.

Можно с уверенностью утверждать, что эти два элемента будут возвращены первыми независимо от порядка их вставки, при условии, что очередь с приоритетом реализована правильно и приоритеты в наших примерах имеют именно такие значения.

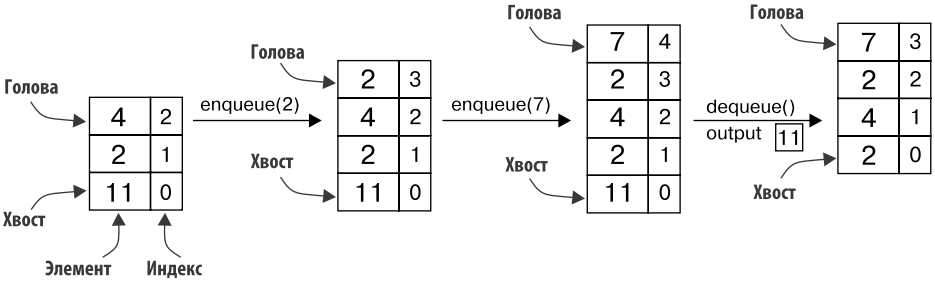


Рис. 2.2. Операции с очередью: элементы являются просто целыми числами, но они могли бы быть любыми значениями, потому что в обычных очередях приоритет определяется только порядком вставки (см. приложение Г). Вставка (enqueue) добавляет элемент в начало очереди. Удаление (dequeue) удаляет последний элемент в очереди и возвращает его. С некоторыми оговорками обе операции могут быть выполнены за постоянное время

2.4.2. Приоритет имеет значение: обобщаем FIFO

Теперь вопрос в том, как назначается приоритет элемента. Часто естественное упорядочение, определяемое временем, проведенным элементом в ожидании своей очереди, может считаться наиболее справедливым. Однако иногда в некоторых элементах есть какая-то особенная черта или характеристика, указывающая на то, что их следует обслуживать раньше, чем другие, ожидавшие дольше. Например, вы не всегда читаете свои электронные письма в том порядке, в котором вы их получили, пропуская информационные бюллетени или «забавные» сообщения от друзей, чтобы сначала прочитать корреспонденцию, связанную с работой. Точно так же в отделении неотложной помощи следующим, кем займутся врачи, не обязательно будет пациент, который ожидает дольше всего. Скорее тяжесть заболевания будет оцениваться при поступлении больного, и ему будет присвоен определенный приоритет. Тот, у кого приоритет окажется наивысшим, будет вызван первым, когда врач освободится.

В этом и заключается идея очередей с приоритетом: они ведут себя как обычные, простые очереди, за исключением того, что начало очереди определяется динамически на основе значения некоторого приоритета. Введение приоритета вызывает настолько глубокие изменения в реализации, что это влечет за собой создание особого вида структуры данных.

Но это еще не все: мы можем даже определить такие базовые контейнеры, как *сумка (bag)* или *стек*, как частные случаи очередей с приоритетом — в приложении Г показано, как это сделать. Рассмотрение материала поможет вам глубже понять, как работают очереди с приоритетом, хотя на практике подобные контейнеры обычно реализуются непосредственно при возникновении необходимости, потому что использование их специфики позволяет существенно улучшить производительность.

2.5. КОНКРЕТНЫЕ СТРУКТУРЫ ДАННЫХ

Теперь перейдем от абстрактных структур данных к конкретным. Знания того, как работает API очереди с приоритетом, достаточно для относительно успешного ее применения, но может быть недостаточно для того, чтобы работать с ней наилучшим образом. Часто при работе с компонентами, критичными к времени, или с приложениями, интенсивно использующими данные, необходимо понимать и внутренности структур данных, и детали их реализации, чтобы убедиться, что мы можем интегрировать их в наше решение, не создавая узких мест[1].

Каждая абстракция должна быть реализована с использованием конкретной структуры данных. Например, стек можно организовать с помощью списка, массива или

---

[1] Помимо производительности, в зависимости от контекста может потребоваться проверить и другие аспекты. Например, в распределенной среде необходимо убедиться, что наша реализация является потокобезопасной, иначе можно столкнуться с условиями гонки — самыми неприятными возможными ошибками, которые могут негативно повлиять на работу приложений.

теоретически даже кучи (хотя это было бы глупо, и в одном из последующих разделов я разъясню почему). Выбор базовой структуры данных, используемой для реализации контейнера, будет влиять только на производительность контейнера. Решение о признании реализации наилучшей обычно является компромиссом: некоторые структуры данных ускоряют одни операции, но замедляют другие.

2.5.1. Сравнение производительности

Для реализации очереди с приоритетом сначала рассмотрим три наивных варианта с использованием основных структур данных, описанных в приложении В: несортированный массив, который мы просто пополняем, добавляя элементы в его конец; отсортированный массив, где мы следим за тем, чтобы порядок восстанавливался каждый раз, когда добавляется новый элемент; и сбалансированные деревья, частным случаем которых являются кучи. Теперь сведем время выполнения[1] основных операций с помощью этих структур данных в таблицу (табл. 2.2) и проанализируем.

2.5.2. Какая конкретная структура данных наиболее верна?

Из табл. 2.2 видно, что наивный выбор привел бы к линейным требованиям по времени, по крайней мере для одной из основных операций, в то время как сбалансированное дерево всегда гарантировало бы логарифмическое время в наихудшем случае. Хотя линейное время обычно считается «выполнимым», все же существует огромная разница между логарифмическим и линейным: для миллиарда элементов это разница между 1 миллиардом операций и их количеством в несколько десятков. Если каждая операция занимает 1 мс, это означает разницу между длительностью 11 дней и промежутком менее чем в секунду.

Таблица 2.2. Производительность операций, обеспечиваемых ОП, в зависимости от базовой структуры данных

Операция	Несортированный массив	Отсортированный массив	Сбалансированное дерево
Вставка	O(1)	O(*n*)	O(log n)
Поиск минимума	O(1)*	O(1)	O(1)*
Удаление минимума	O(*n*)**	O(1)***	O(log n)

* Сохраняет дополнительное значение для минимума, что приводит к дополнительным тратам на сохранение его значения при вставке и удалении.

** Если для ускорения поиска минимума используется буфер, то нужно найти следующий минимум при удалении элемента. К сожалению, ничего не бывает бесплатно. В качестве альтернативы можно было бы иметь удаление с постоянным временем (отказавшись от буфера и заменяя удаленный элемент последним в массиве) и поиск минимума с линейным временем.

*** При хранении массива в обратном порядке удаление последнего элемента может просто вести к уменьшению размера массива или отслеживанию того, какой элемент в массиве идет последним.

[1] Введение в алгоритмический анализ и нотацию «О большое» можно найти в приложении Б.

Кроме того, учтите, что в большинстве случаев контейнеры и, в частности, очереди с приоритетом используются в качестве вспомогательных структур, то есть они являются частью более сложных алгоритмов/структур данных, и каждый цикл основного алгоритма может вызывать операции над ОП несколько раз. Например, для алгоритма сортировки это может означать переход от $O(n^2)$ (за этим обычно стоит невозможность уже для n, равного 1 миллиону или даже меньше) к $O(n \log(n))$ (что все еще допустимо для входных данных размером 1 миллиард или более)[1]. Однако ситуация компенсируется тем, что реализация сбалансированных двоичных деревьев обычно нетривиальна.

В следующем подразделе будет описан способ эффективной реализации универсальной очереди с приоритетом.

2.5.3. Куча

Двоичная куча — наиболее часто используемый вариант реализации очередей с приоритетом. Она позволяет вставлять и извлекать элементы в порядке возрастания или убывания, по одному за раз.

Хотя на самом деле для хранения элементов кучи используется базовый массив, концептуально его можно представить как особый вид двоичного дерева, соблюдая три инварианта.

- У каждого узла есть не более двух дочерних элементов.
- Дерево кучи полно и скорректировано слева. Полнота дерева (рис. 2.3) означает, что, если куча имеет высоту H, каждый листовой узел находится либо на уровне H, либо на уровне $H - 1$. Все уровни скорректированы слева, а это означает, что ни одно правое поддерево не имеет высоты больше, чем его левый сосед. Итак, если лист находится на той же высоте, что и внутренний узел[2], лист не может быть слева от этого узла. Инварианты 1 и 2 являются структурными инвариантами для кучи.
- Каждый узел имеет наивысший приоритет в поддереве, корнем которого является этот узел.

Свойства 1 и 2 допускают представление кучи в виде массива. Предположим, нужно хранить N элементов. Тогда можно компактно представить древовидную структуру непосредственно с использованием массива из N элементов без указателей на дочерние или родительские элементы. На рис. 2.4 показано, почему представления кучи в виде дерева и массива эквивалентны.

---

[1] Действительно ли разрешима задача для определенного размера входных данных, зависит от типа выполняемых операций и времени, которое они занимают. Но даже если каждая операция занимает всего 1 нс и если на вход поступает 1 миллион элементов, квадратичный алгоритм займет более 16 мин, а линейный алгоритм потребует менее 10 мс.

[2] Лист — это узел дерева, у которого нет дочерних элементов. Внутренний узел — узел, у которого есть хотя бы один дочерний элемент, или, что то же самое, узел, не являющийся листом.

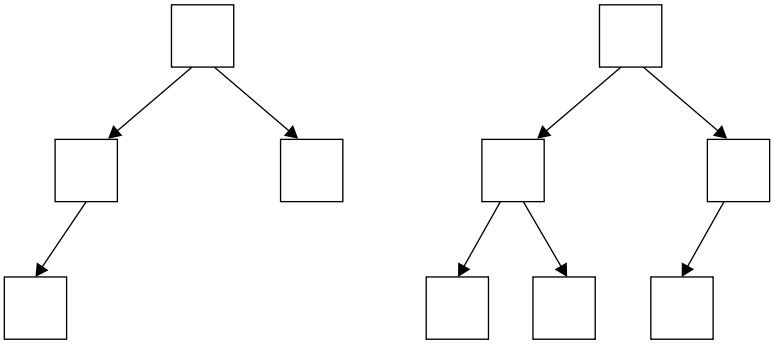


Рис. 2.3. Два примера полных двоичных деревьев. Все уровни в дереве имеют максимально возможное количество узлов, кроме (возможно) последнего. Все листья на последнем уровне выравниваются слева; на предыдущем уровне ни одно правое поддерево не имеет больше узлов, чем его левый сосед

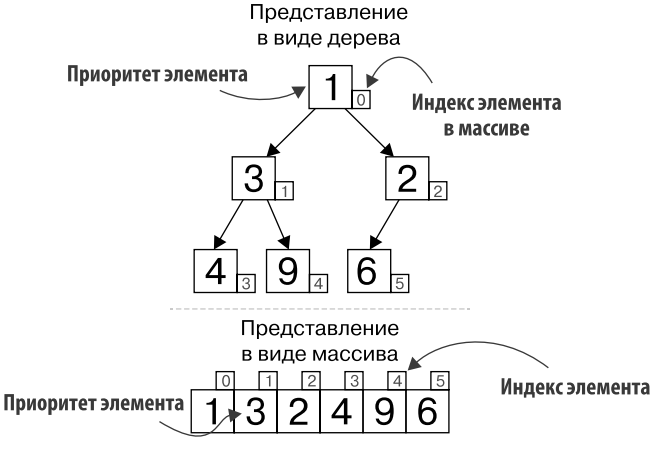


Рис. 2.4. Двоичная куча. Для узлов показаны только приоритеты (элементы в данном случае неважны). Цифры внутри маленьких квадратов показывают индексы элементов кучи в массиве. Узлы сопоставляются с элементами массива сверху вниз и слева направо. *Вверху:* представление кучи в виде дерева. Обратите внимание, что каждый родитель меньше (или в крайнем случае равен) его дочерних элементов и, в свою очередь, всех элементов в своем поддереве. *Внизу:* та же куча, представленная в виде массива

Если использовать представление через массив и считать индексы с 0, то дети i-го узла будут храниться в позициях[1] $(2 \times i) + 1$ и $2 \times (i + 1)$, а родитель узла i будет находиться по индексу $(i - 1) / 2$ (за исключением корня, у которого нет родителя). Например, на рис. 2.4 видно, что узел с индексом 1 (с приоритетом 3) имеет два дочерних узла

[1] Здесь в выражениях расставлены круглые скобки. В остальной части книги излишние скобки будут опущены, так что, например, будет написано: $2 \times i + 1$.

с индексами 3 и 4, а его родитель —индекс 0; для узла в позиции 5 его родителем будет элемент с индексом 2, а его дочерние элементы (не показаны на рисунке) будут иметь индексы 11 и 12.

Может показаться нелогичным, что мы используем массив для представления дерева. Ведь, в конце концов, деревья были изобретены для преодоления ограничений массива. В целом это верно, и деревья имеют ряд преимуществ: они более гибкие и, если они сбалансированы, обеспечивают лучшую производительность с логарифмическим поиском, вставкой и удалением в наихудшем случае.

Но улучшение, которое получается благодаря деревьям, конечно же, имеет свою цену. В первую очередь, как и в случае любой структуры данных, использующей указатели (списки, графики, деревья и т. д.), появляются дополнительные расходы памяти по сравнению с массивами. Ведь для массива просто нужно зарезервировать место для данных (плюс, возможно, в зависимости от деталей реализации, некоторое постоянное пространство для указателей и самой структуры узла), в то время как каждый узел дерева требует дополнительного места для указателей на его дочерние элементы и, возможно, родительский узел.

Не вдаваясь в подробности, также укажу, что массивы, как правило, лучше используют *локальность памяти*: все элементы массива в памяти являются смежными, а это означает меньшую задержку при их чтении.

В табл. 2.3 показано, что куча, с конкретной моделью данных и инвариантами, действительно соответствует абстрактному определению очереди с приоритетом.

Таблица 2.3. Базовые компоненты кучи

Конкретная структура данных: куча	

API	`Heap {`
	`  top() → element`
	`  peek() → element`
	`  insert(element, priority)`
	`  remove(element)`
	`  update(element, newPriority)`
	`}`

Контракт с клиентом	Верхний элемент, возвращаемый очередью, всегда является элементом с наивысшим приоритетом, хранящимся в данный момент в очереди
Модель данных	Массив, элементами которого являются элементы, хранящиеся в куче
Инварианты	Каждый элемент имеет двух «потомков». Для элемента в позиции `i` его дочерние элементы расположены по индексам `2*i+1` и `2*(i+1)`
	Каждый элемент имеет более высокий приоритет, чем его дочерние элементы

2.5.4. Приоритет, неубывающая и невозрастающая куча

Когда мы приводили три свойства кучи, использовали формулировку «наивысший приоритет». Говоря о куче, всегда можно утверждать, что элемент с наивысшим приоритетом будет вверху, не вызывая никакой неопределенности.

Когда же дело доходит до практики, необходимо задать метод определения приоритета. Если мы реализуем кучу общего назначения, можно просто параметризовать ее пользовательской функцией приоритета, принимающей элемент и возвращающей его приоритет. До тех пор пока эта реализация соответствует трем законам, указанным в подразделе 2.5.3, можно с уверенностью утверждать, что куча работает в соответствии с ожиданиями.

Однако иногда лучше иметь специализированные реализации, которые используют знания предметной области и избавлены от накладных расходов, связанных с пользовательской функцией приоритета. Например, если хранить задачи в куче, можно использовать кортежи `(приоритет, задача)` в качестве элементов и полагаться на естественное упорядочение кортежей[1].

Так или иначе, еще необходимо понять, как определять наивысший приоритет. Если предполагать, что наивысший приоритет обозначается большим числом, то есть если из $p_1 > p_2$ следует, что p_1 является наивысшим приоритетом, тогда куча называется невозрастающей (max-heap).

В других случаях бывает необходимо сначала возвращать наименьшие числа, тогда надо принять, что из $p_1 > p_2$ следует, что p_2 имеет наивысший приоритет. В этом случае используется так называемая неубывающая куча (min-heap). В оставшейся части книги и, в частности, в разделе кодирования предполагается реализация невозрастающей кучи.

Реализация неубывающей кучи лишь немного отличается от реализации невозрастающей. Код в значительной степени симметричен, и нужно просто поменять местами все вхождения $<$ и $\leq$ на $>$ и $\geq$ соответственно, а также поменять min на max.

Или еще проще — если уже имеется реализация одного типа кучи, то можно получить другой тип, просто используя обратные величины приоритетов (либо в функции приоритета, либо при явной передаче приоритета).

Чтобы привести конкретный пример, предположим, что имеется неубывающая куча, которую вы используете для хранения пар `(возраст, задача)`: неубывающая куча сначала вернет задачу с наименьшим возрастом. Однако в какой-то момент может потребоваться, чтобы первой возвращалась самая старая задача. Для этого больше подойдет невозрастающая куча. Оказывается, можно получить нужный результат без изменения кода для кучи! Вместо этого нужно просто хранить элементы в виде кортежей `(-возраст, задача)`. Фактически если `x.age` $<$ `y.age`, то `-x.age` $>$ `-y.age`, и, таким образом, неубывающая куча сначала вернет задачи, чей возраст имеет наибольшее абсолютное значение.

Предположим, дана задача `A` с возрастом 2 (дня) и задача `B` с возрастом 3. Создадим кортежи `(-2, A)` и `(-3, B)`. При извлечении их из неубывающей кучи можно быть уверенными, что `(-3, B)` $<$ `(-2, A)` и поэтому задача `B` будет возвращена раньше `A`.

---

[1] $(a_1, b_1, c_1...) < (a_2, b_2, c_2...)$ тогда и только тогда, когда $a_1 < a_2$ или ($a_1 == a_2$ и $(b_1, c_1...) < (b_2, c_2...)$).

2.5.5. Продвинутый вариант: d-куча

Можно подумать, что кучи могут быть двоичными деревьями и только. Действительно, двоичные деревья поиска — наиболее распространенный вид деревьев, неразрывно связанный с упорядочением. Но сохранять коэффициент ветвления[1] фиксированным и равным 2 вовсе не обязательно, чтобы использовать эту структуру данных. Можно присваивать ему значение больше 2 и применять то же, что и ранее, представление кучи в виде массива.

Для коэффициента ветвления 3 дочерние элементы i-го узла находятся в индексах $3 \times i + 1$, $3 \times i + 2$ и $3 \times (i + 1)$, а родитель узла i — в положении $(i - 1) / 3$.

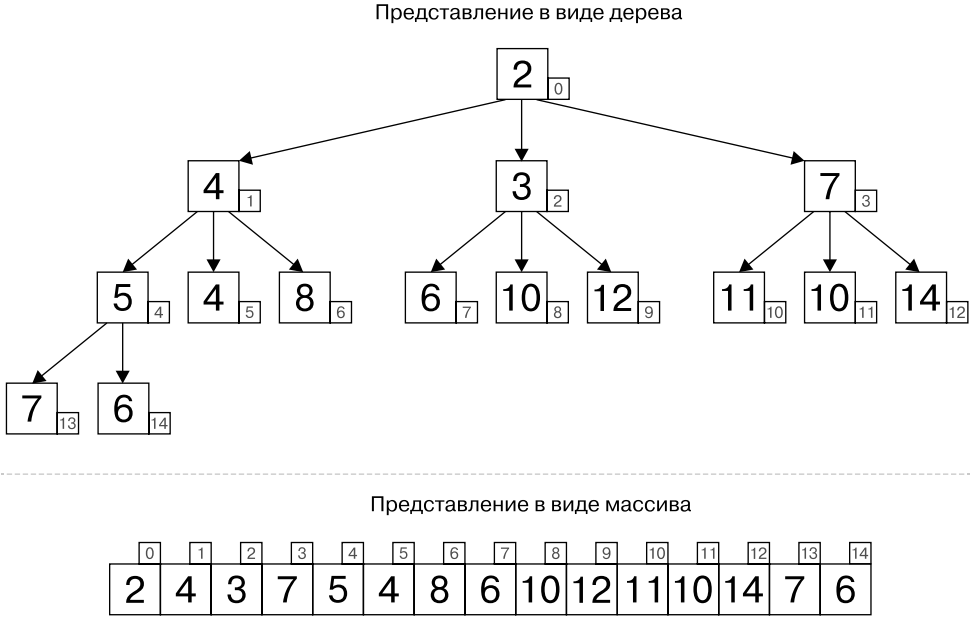


Рис. 2.5. Троичная куча. Для узлов показаны только приоритеты (значения в данном случае неважны). Меньшие квадраты рядом с элементами показывают индексы элементов кучи в массиве. *Вверху:* представление неубывающей кучи в виде дерева. Обратите внимание, что каждый родитель меньше (или в крайнем случае равен) его дочерних элементов и, в свою очередь, всех элементов в своем поддереве. *Внизу:* та же куча, представленная в виде массива

На рис. 2.5 показано представление троичной кучи как в виде дерева, так и в виде массива. Та же идея справедлива для факторов ветвления 4, 5 и т. д.

---

[1] Коэффициент ветвления дерева — это максимальное количество потомков, которое может иметь узел. Например, для двоичной кучи, которая, в свою очередь, является двоичным деревом, коэффициент ветвления равен 2. Дополнительные сведения об этом приведены в приложении В.

Для *d-кучи*, где коэффициент ветвления равен целому числу $D > 1$, приведенные выше три инварианта кучи превращаются в такие.

- Каждый узел имеет не более D потомков.
- Дерево кучи полно, все уровни скорректированы слева. То есть, каждое `i`-е поддерево имеет высоту, не превышающую высоту соответствующих ему родственных поддеревьев слева (от 0 до $i - 1$, $1 < i \leq D$).
- Каждый узел имеет наивысший приоритет в поддереве, корнем которого является этот узел.

ЛЮБОПЫТНЫЙ ФАКТ

Стоит отметить, что при $D = 1$ куча становится отсортированным массивом (или отсортированным двусвязным списком, если рассматривать ее представление в виде дерева). Создание кучи будет производиться алгоритмом сортировки вставками и потребует квадратичного времени. Любая другая операция потребует линейного времени.

2.6. КАК РЕАЛИЗОВАТЬ КУЧУ

На данный момент имеется хорошее понимание внутреннего представления очереди с приоритетами того, как ее использовать. Пришло время углубиться в детали реализации кучи.

Прежде чем мы начнем, еще раз взглянем на наш API:

```
class Heap {
    top()
    peek()
    insert(element, priority)
    remove(element)
    update(element, newPriority)
}
```

Это всего лишь публичные методы, определяющие API этого класса.

Но обо всем по порядку. В дальнейшем будем предполагать, что храним все кортежи (`элемент`, `приоритет`), добавленные в кучу, в массиве `pairs`, как показано в листинге 2.1.

Листинг 2.1. Свойства класса DHeap

```
class DHeap
    #type Array[Pair]
    pairs
    function DHeap(pairs=[])
```

Этот фрагмент кода — первый в нашей книге, так что сейчас самое время ознакомиться с приложением А, где объясняется используемый синтаксис. Например, если имеется

переменная p, содержащая такой кортеж, то используется специальный синтаксис для деструктурированного присвоения его полей двум переменным:

```
(element, priority) ← p
```

Также предположим, что поля кортежа именованы, так что можно, в свою очередь, получить доступ к `p.element` и `p.priority` или создать кортеж p, используя такой синтаксис:

```
p ← (element='x', priority=1)
```

На многих рисунках в этом разделе будут демонстрироваться только приоритеты элементов. Иначе говоря, мы предполагаем, что элементы — то же, что и приоритеты. Налицо упрощение, принятое для экономии места и увеличения понятности диаграмм, но оно также подчеркивает важную характеристику куч: во всех их методах нужно только получать доступ к приоритетам и перемещать их. Это может быть важно, когда элементы представляют собой большие объекты, особенно если они настолько велики, что не помещаются в кэш/память или по какой-либо причине хранятся на диске — тогда конкретная реализация может просто хранить и получать доступ к ссылке на элементы и их приоритет.

Теперь, прежде чем углубляться в методы API, определим две вспомогательные функции, которые будут использоваться для восстановления свойств кучи при каждом изменении. Перечислим все возможные изменения кучи:

- добавление нового элемента в кучу;
- удаление верхнего элемента кучи;
- обновление приоритета элемента.

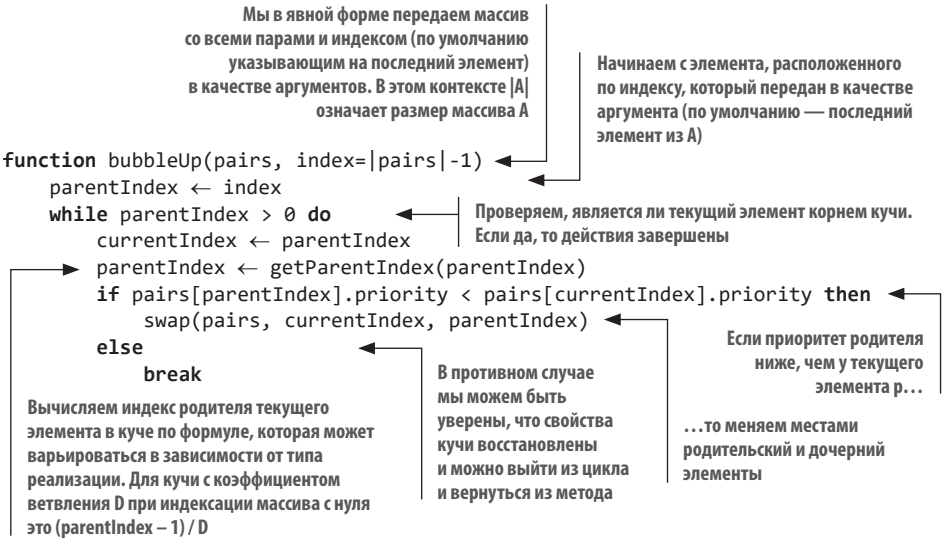
Любая из этих операций может привести к тому, что какой-то элемент кучи будет иметь более высокий приоритет, чем его родительский элемент, или более низкий приоритет, чем хотя бы один из его дочерних элементов.

2.6.1. BubbleUp

Когда элемент имеет более высокий приоритет, чем его родитель, необходимо вызвать метод `bubbleUp` (всплытие), показанный в листинге 2.2. На рис. 2.6 приведен пример, основанный на уже рассмотренном инструменте управления задачами: приоритет элемента с индексом 7 выше, чем у его родителя (с индексом 2), поэтому их необходимо поменять местами.

Как показано в листинге 2.2, мы продолжаем менять местами элементы до тех пор, пока либо текущий элемент не окажется корневым (строка 3), либо его приоритет не станет ниже, чем у его непосредственного предка (строки 6–9). То есть каждый вызов этого метода может включать не более $\log_D(n)$ сравнений и перестановок, поскольку он ограничен высотой кучи.

Листинг 2.2. Метод bubbleUp



Как вы помните, мы реализуем невозрастающую кучу, поэтому более высокие числа обозначают более высокий приоритет. Таким образом, элемент с индексом 7 на рис. 2.6, *А* находится не на своем месте, потому что его родитель «Необязательное поле формы заблокировано...» с индексом 2 имеет приоритет $8 < 9$.

В данном случае, чтобы исправить ситуацию, необходимо вызвать `bubbleUp(pairs, 7)`. Входим в цикл в строке 3, поскольку `parentIndex = 7 > 0`, вычисляем индекс нового родителя, который равен 2, и, поскольку условие в строке 6 также верно, меняем элементы местами в строке 7. После обновления массив кучи будет выглядеть так, как показано на рис. 2.6, *Б*.

На следующей итерации цикла (`parentIndex = 2 > 0`, поэтому мы войдем в цикл как минимум еще раз) получим новые значения для `currentIndex` и `parentIndex` в строках 4–5, они равны 2 и 0 соответственно.

Поскольку приоритеты дочернего и родительского элементов, в свою очередь, теперь равны 9 и 10, условие в строке 6 больше не выполняется и функция выйдет из цикла, вернув управление.

Обратите внимание, что, если высота дерева равна *H*, этот метод требует не более чем *H* перестановок, потому что каждый раз, меняя местами два элемента, мы перемещаемся на один уровень вверх по дереву, к его корню.

Это важный результат, как будет показано в разделе 2.7. Поскольку кучи представляют собой сбалансированные бинарные деревья, их высота логарифмична количеству элементов.

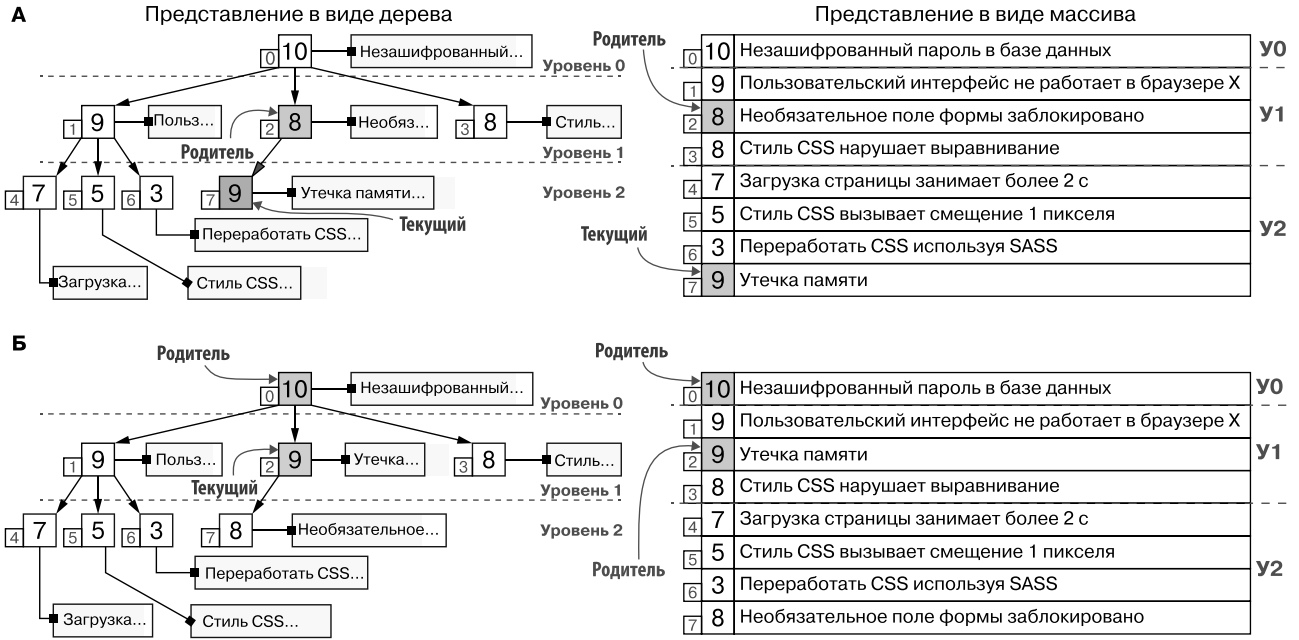


Рис. 2.6. Метод bubbleUp в действии для троичной невозрастающей кучи. *А*. Элемент с индексом 7 имеет более высокий приоритет 9, чем его родитель (чей приоритет равен 8). *Б*. Меняем местами элемент с индексом 7 на его родителя с индексом 2. Элемент, который теперь получил индекс 7, безусловно, встал на свою конечную позицию, в то время как всплывающий элемент необходимо сравнить с его новым родителем. В данном случае корень имеет более высокий приоритет, поэтому всплытие прекращается

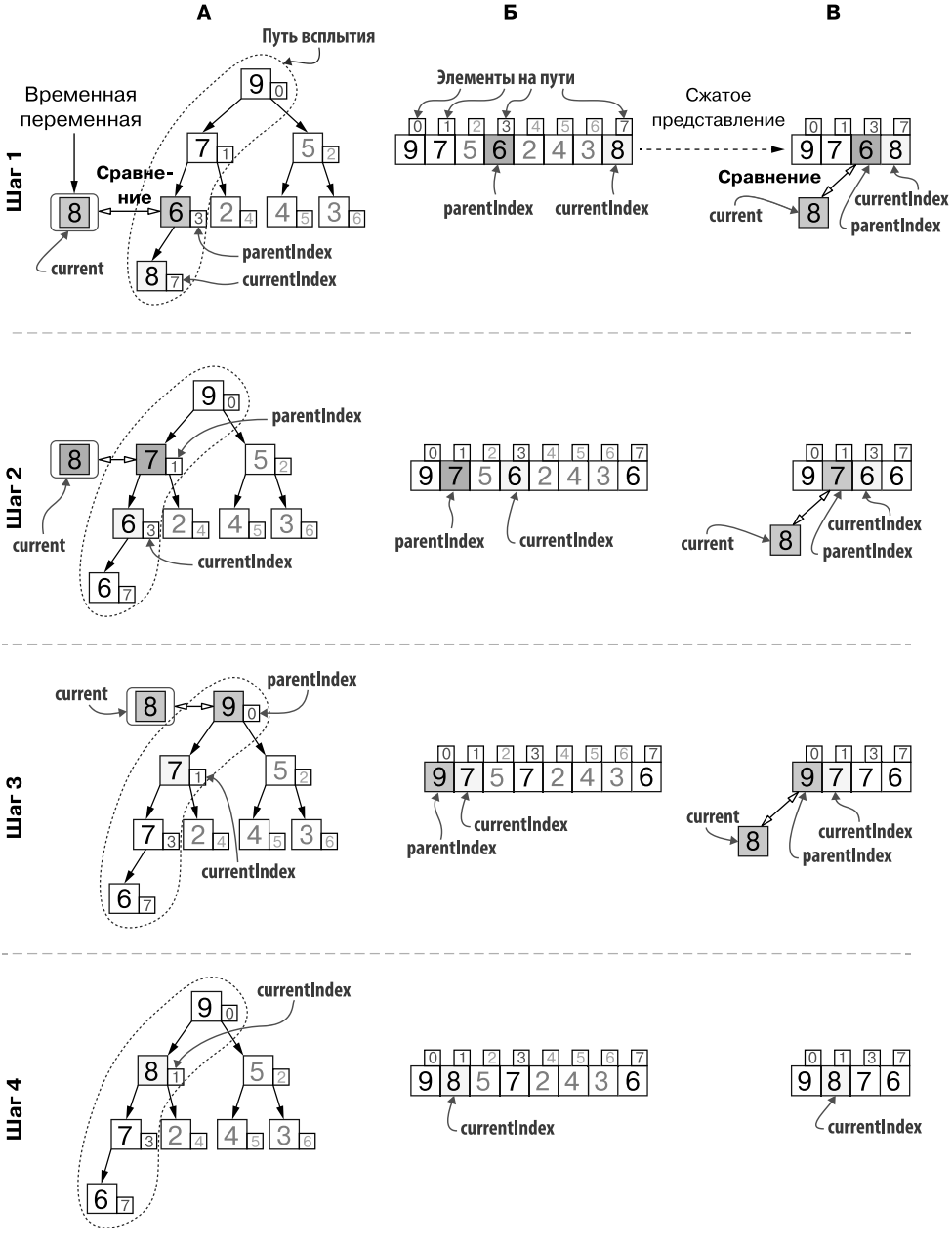


Рис. 2.7. Всплытие элемента в невозрастающей куче с использованием метода, уменьшающего количество установок значений до одной трети. На рисунке показаны вызов bubbleUp(pairs, 7), проиллюстрированный через представление в виде дерева *(А)* и массива *(Б)* для невозрастающей кучи, а также отфильтрованные элементы *(В)* на пути P от элемента к корню

Однако можно сделать еще одну небольшую доработку для дальнейшего улучшения алгоритма. Если вы заметили, мы неоднократно меняем местами один и тот же элемент с его текущим родителем. Это элемент, для которого изначально вызывается метод `bubbleUp`, и он продолжает двигаться к корню. На рис. 2.6 можно видеть, что он всплывает вверх, подобно мыльному пузырю, поднимаясь к вершине кучи. Для каждой перестановки требуются три присваивания (и использование временной переменной), поэтому наивная реализация в листинге 2.2 потребует (максимум) $3 \times H$ присваиваний.

И хотя индексы текущего и родительского узлов изменяются на каждой итерации, значение текущего элемента всегда остается одним и тем же.

На рис. 2.7 показан путь, который должен пройти такой элемент. В худшем случае он доходит до корня, как показано на рисунке, но всплытие могло бы остановиться и на промежуточном узле. Оказывается, всплытие элемента на пути *P* к корню кучи эквивалентно вставке этого элемента в (под)массив, полученный из элементов, входящих в *P* (см. рис. 2.7, *Б*).

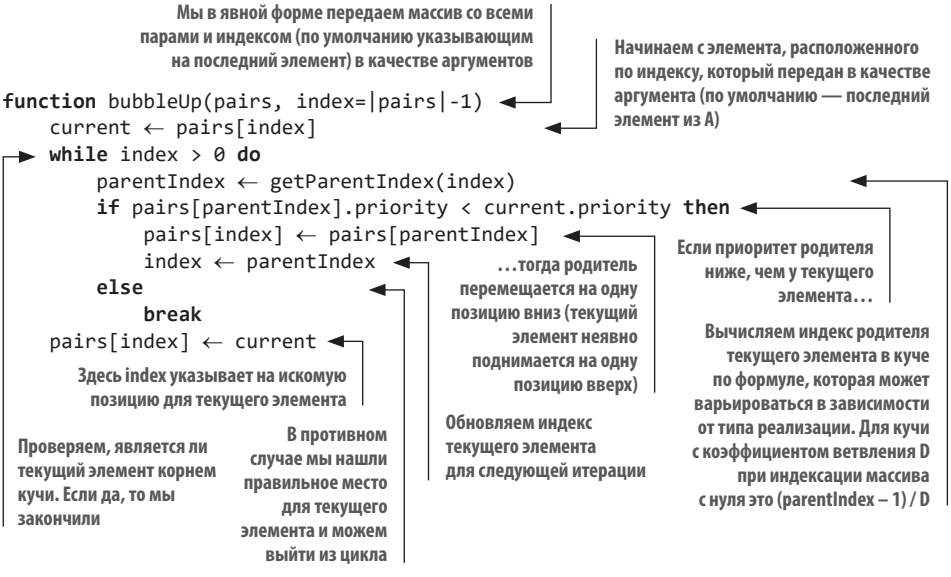
Продолжим рассмотрение рис. 2.7: наше первое действие — это сохранение элемента *X*, который будет двигаться вверх, во временной переменной (шаг 1). Затем, начиная с его родителя, сравним все элементы пути с временной переменной и скопируем их, если они имеют более низкий приоритет (шаги 1–3). Каждый раз копируем родительский элемент поверх того его дочернего элемента, который находится на пути *P*. Мы как бы отбрасываем все элементы в массиве кучи, которые не являются частью *P*: это отчетливо видно на рис. 2.7, *В*.

Наконец, когда мы вдоль *P* находим элемент *Y*, который имеет более высокий приоритет, чем наш временный, получаем возможность скопировать элемент *X* из временной переменной в дочерний элемент *Y*, принадлежащий *P* (шаг 4).

Итак, появилась возможность провести нашу операцию эффективно — точно так же, как работает каждая итерация алгоритма сортировки вставками. Сначала сохраняем во временной переменной копию элемента, который должен всплыть (назовем его *X*), а затем проверяем элементы слева от него в массиве. Мы «перемещаем» элементы вправо (см. рис. 2.7, *В*), копируя их на следующую от них позицию справа, пока не найдем элемент с приоритетом выше, чем у *X*. Это можно сделать, используя только (не более) $H + 1$ присваиваний для пути длиной *H*, таким образом экономя около 66 % присваиваний.

В листинге 2.3 показана улучшенная версия метода `bubbleUp`. Обратите внимание, как в какой-то момент на мгновение появятся две копии элемента с исходным индексом 3 (а позже и элемента с индексом 1). Так происходит потому, что вместо фактической замены элементов местами мы можем просто перезаписывать текущий элемент его родителем на каждой итерации (строка 6) и размещать всплывающий элемент только один раз в конце как последнее действие метода. Это работает, потому что всплытие элемента концептуально эквивалентно внутреннему циклу *сортировки вставками* — ищем правильную позицию для вставки текущего элемента в путь, соединяющий его с корнем кучи.

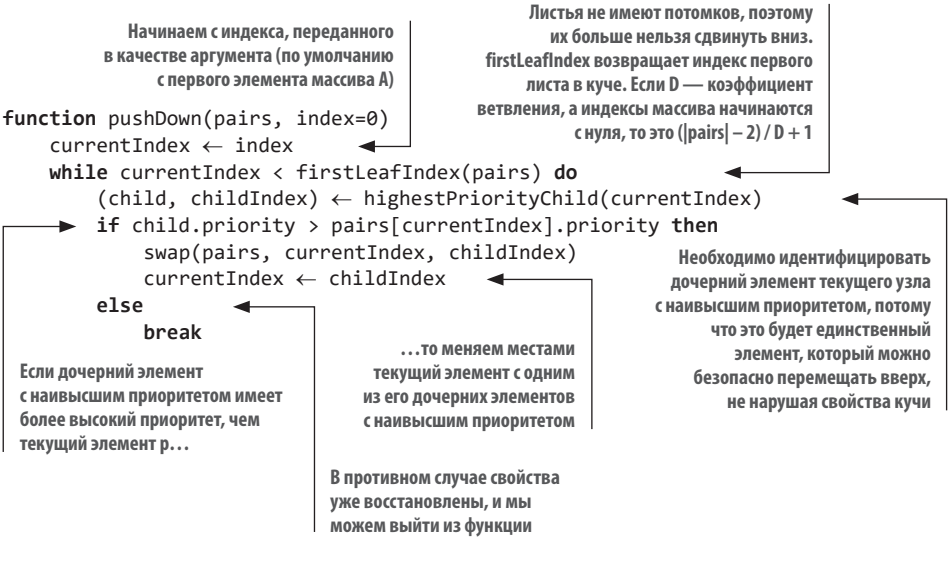
Листинг 2.3. Оптимизация метода bubbleUp



2.6.2. PushDown

Метод `pushDown` («проталкивание вниз») необходимо применять в симметричном случае, когда нужно переместить элемент вниз поближе к листьям кучи, потому что он может быть меньше, чем (по крайней мере) один из его дочерних элементов. Реализация показана в листинге 2.4.

Листинг 2.4. Метод pushDown



Здесь есть два существенных отличия от всплытия.

- *Условие срабатывания* — в этом случае измененный узел не нарушает инварианта 3 по отношению к своему родителю, но способен нарушить его по отношению к своим дочерним узлам. Так может произойти, например, если извлечь корень из кучи и заменить его последним листом массива или если присвоить более низкое значение приоритету узла.
- *Алгоритм* — для каждого уровня, через который проходит элемент, подвергаемый сталкиванию вниз, нужно найти его дочерний элемент с наивысшим приоритетом, чтобы определить, куда пристроить этот элемент, не нарушая никаких свойств.

Хотя время выполнения этого метода и асимптотически эквивалентно времени выполнения метода `bubbleUp`, метод PushDoown требует немного большего количества сравнений. В худшем случае их число равно $D \times \log_D(n)$ для кучи с коэффициентом ветвления D, содержащей n элементов. По этой причине бесконечное увеличение коэффициента ветвления (максимального количества дочерних элементов для каждого узла) — не очень хорошая идея (я приведу дополнительные иллюстрации в разделе 2.10).

Продолжая пример с задачами, рассмотрим случай, когда приоритет корня ниже, чем приоритет его потомков, как показано на рис. 2.8. У троичной кучи первый лист хранится по индексу 4.

Вызываем алгоритм `pushDown(pairs, 0)`, чтобы восстановить третье свойство кучи; вспомогательная функция `firstLeafIndex` в строке 3 возвращает 3 (поскольку элемент с индексом 7, последний в массиве, является дочерним элементом узла с индексом 2, который является последним внутренним узлом), так что мы заходим в цикл.

Дочерние элементы для элемента 0 находятся в позициях 1, 2 и 3, среди них в строке 4 алгоритма выберем дочерний элемент корня с наивысшим приоритетом. Им оказывается элемент `<Незашифрованный пароль в базе данных, 10>`. Его приоритет выше, чем у текущего элемента, поэтому в строке 6 мы поменяем его местами с текущим элементом.

В целом после одной итерации цикла `while` имеем ситуацию, показанную в части *Б* рис. 2.8, а `currentIndex` устанавливается равным 1.

Поэтому, когда мы входим в цикл во второй раз, в строке 3 `currentIndex` возвращает 1, что все еще меньше, чем 3 (индекс первого листа). Потомки элемента с индексом 1 находятся по индексам 4, 5 и 6. В строке 4 листинга 2.4 среди них отыскиваем узел с наивысшим для дочерних элементов текущего узла приоритетом; им оказывается элемент с индексом 4, `<Загрузка страницы занимает более 2 секунд, 7>`.

Таким образом, в строке 5 обнаруживаем, что на этот раз приоритеты всех дочерних элементов ниже, чем у текущего элемента, и выходим из цикла и из функции без дальнейших изменений.

Как и в случае с `bubbleUp`, функцию `pushDown` можно улучшить, просто сохранив текущий элемент во временной переменной и избегая перестановок на каждой итерации.

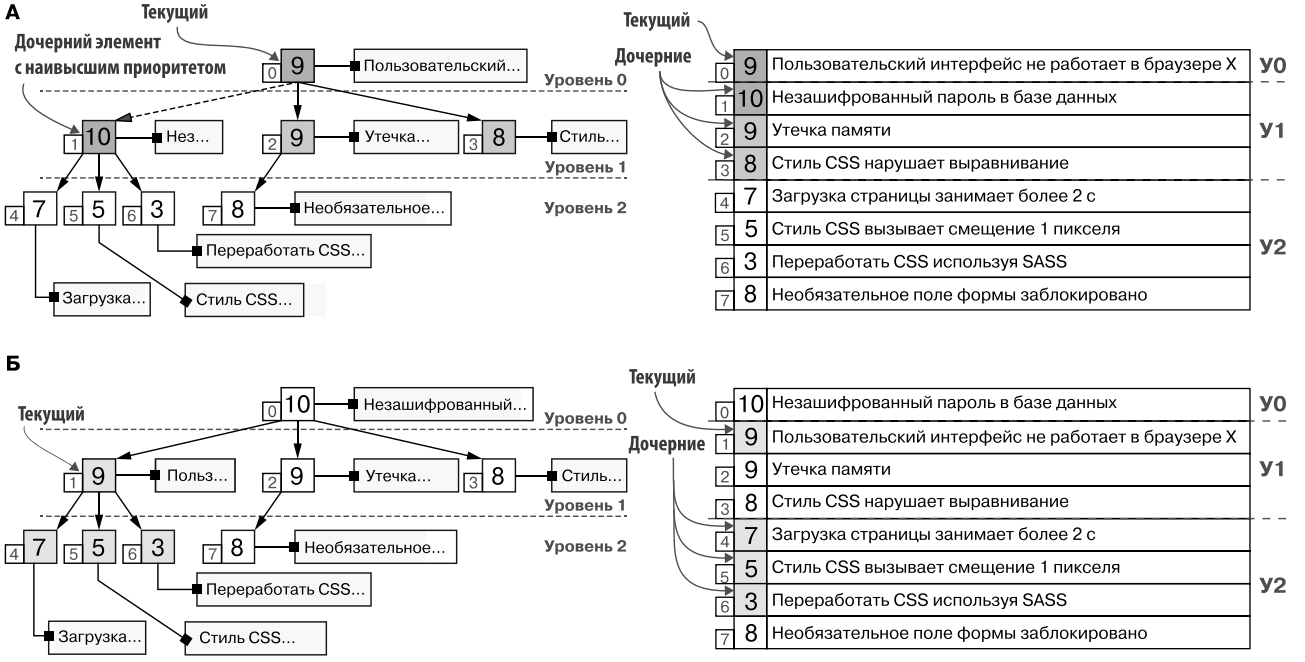
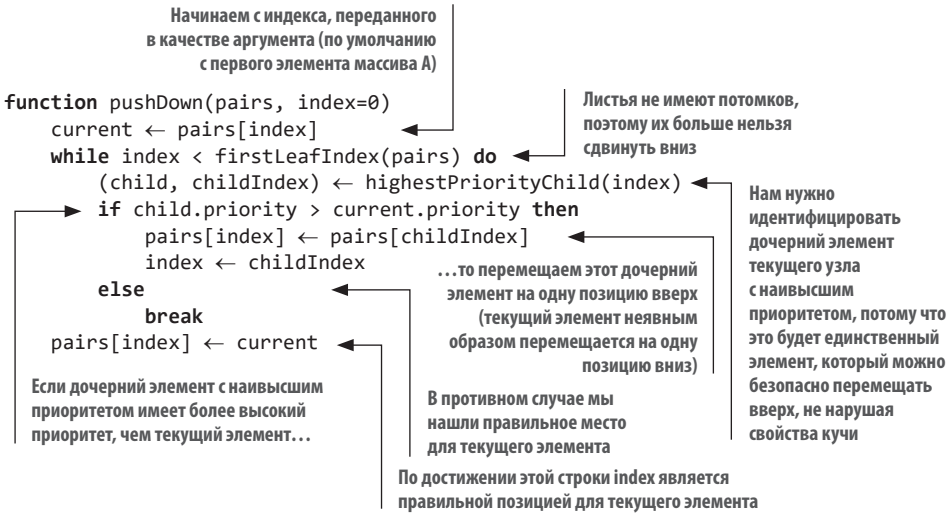


Рис. 2.8. Пример применения метода pushDown к корню троичной кучи. *А*. Корень дерева нарушает свойства кучи, будучи меньше одного из своих дочерних элементов. Алгоритм находит дочерний элемент с наивысшим приоритетом, а затем сравнивает его с текущим узлом (в данном примере — с корнем). *Б*. Мы поменяли местами корень с его дочерним элементом с наивысшим приоритетом: узлом с индексом 1. Теперь пришла очередь сравнить обновленный дочерний элемент с его дочерними элементами: поскольку ни один из них не имеет более высокого приоритета, чем текущий узел, можно остановиться

В листинге 2.5 показана окончательная улучшенная версия метода. Я рекомендую читателю разобрать этот пример построчно, чтобы лучше понять, как он работает, подобно тому как это было сделано в предыдущем разделе для `bubbleUp`.

Листинг 2.5. Оптимизация метода pushDown



Теперь, когда определены все необходимые вспомогательные методы, наконец становится возможно реализовать методы API. Слегка предвосхитив события, замечу, что вся логика приоритетов и специфики невозрастающих и неубывающих куч инкапсулирована в методах `bubbleUp` и `pushDown`, поэтому, разобравшись с этими двумя вспомогательными методами, мы значительно упростили себе жизнь в будущем, когда дело дойдет до адаптации нашего кода к различным ситуациям.

2.6.3. Вставка

Начнем со вставки. В листинге 2.6 описан псевдокод для вставки новой пары `(элемент, приоритет)` в кучу.

Как упоминалось в конце предыдущего раздела, метод вставки может быть реализован с использованием наших вспомогательных методов, и поэтому его код действителен независимо от того, используем мы невозрастающую или неубывающую кучу, и от определения приоритета.

Листинг 2.6. Метод insert

Первые два шага в листинге 2.6 объяснять не требуется: мы просто перестраиваем нашу модель данных, создавая пару из двух аргументов и добавляя ее в конец массива. В зависимости от языка программирования и типа контейнера, используемого для пар, может потребоваться вручную изменить его размер (для массивов со статическими размерами) или просто добавить элемент (для динамических массивов или списков)[1].

Последний шаг необходим, потому что новая пара может нарушить свойства кучи, которые были определены в подразделе 2.5.3. В частности, его приоритет может быть выше, чем у его родителя. Чтобы восстановить свойства кучи, нужно «поднимать» его к корню кучи, пока не будет найдено нужное ему место в дереве.

После операции вставки, как и удаления или обновления, или даже построения кучи необходимо убедиться, что свойства кучи сохраняются (рис. 2.9).

Перенаправлять указатели не придется, ведь мы используем компактную реализацию на основе массива. Значит, структура будет гарантированно соответствовать свойствам 1 (по построению) и 2 (если, конечно, в коде нет ошибок).

Итак, необходимо гарантировать свойство 3, или, другими словами, то, что каждый узел больше (для невозрастающей кучи), чем каждый из его D потомков. Для этого, как уже упоминалось в предыдущих разделах, возможно, придется сдвигать его вниз (при удалении и, возможно, обновлении) или поднимать его к корню (при вставке и обновлении) всякий раз, когда нарушатся инварианты кучи.

КАКОВО ВРЕМЯ РАБОТЫ ОПЕРАЦИИ ВСТАВКИ?

Сколько перестановок нужно будет выполнить, чтобы восстановить свойства кучи? Это зависит от конкретного случая, но, поскольку мы поднимаемся на один уровень вверх при каждом обмене, это число не может превышать высоту кучи. А поскольку куча — это сбалансированное полное дерево, то количество перестановок не превысит $\log_D(n)$ для D-ичной кучи с n элементами.

Однако тут есть подводный камень: в любой конкретной реализации придется расширять массив за пределы его текущего размера. Если используется статический массив фиксированного размера, нужно будет выделить его при создании кучи и установить максимальное количество элементов, которые он может содержать. В этой ситуации метод в худшем случае является логарифмическим.

Если же применять динамический массив, то логарифмическая граница становится амортизированной и не характеризует худший случай, потому что нам придется периодически изменять размер массива (см. приложение B, чтобы узнать больше о производительности динамических массивов).

[1] Как объясняется в приложении B, это не меняет выводов асимптотического анализа, поскольку можно доказать, что для вставки n элементов в динамический массив понадобится не более $2n$ присваиваний, поэтому для каждой вставки требуется амортизированное постоянное время.

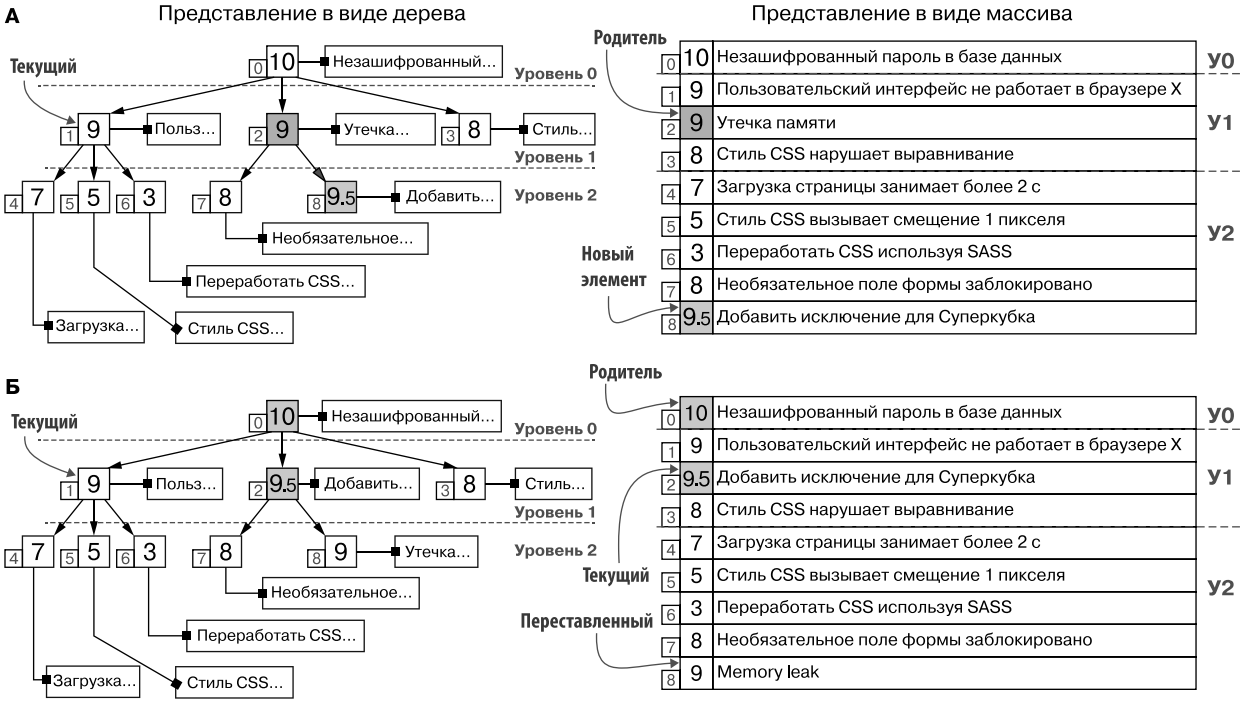


Рис. 2.9. Добавление нового элемента в тернарную невозрастающую кучу. *А*. Начальное состояние кучи; сначала добавляем новый элемент с приоритетом 9.5 в конец массива кучи. *Б*. Затем поднимаем его, пока не найдем для него подходящее место; то есть первое место, где он не нарушит свойства кучи

Вернемся к нашему примеру с управлением задачами, чтобы разобраться, как вставка работает с тернарной невозрастающей кучей. Предположим, у нас есть начальное состояние, показанное на рис. 2.9, *А*, описывающее кучу сразу после вызова `insert("Добавить исключение для Суперкубка", 9.5)`, добавляющего особенно срочную задачу. Маркетинг настаивает, что ее надо исполнить до конца дня. (И да, они хитрят, используя дробные числа для приоритета! Но, вероятно, вы не впервые видите, как кто-то меняет требования после того, как уже проделана большая работа, правда?) Фактически после выполнения строки 3 в листинге 2.6 список окажется в том промежуточном состоянии, где свойства кучи все еще нарушаются.

Затем выполняется строка 4 для восстановления свойства кучи номер 3, как уже демонстрировалось в подразделе 2.6.2. Окончательный результат показан на рис. 2.9, *Б*. Обратите внимание на порядок, в котором потомки корня появляются в массиве: элементы одного уровня не упорядочены — куча не поддерживает общий порядок своих элементов, как это происходит при использовании двоичного дерева поиска.

2.6.4. Метод top

Теперь, узнав, как вставить новый элемент, определим метод `top`, который будет извлекать и возвращать корень кучи.

На рис. 2.10 показан пример невозрастающей кучи и отмечены шаги, выполняемые в листинге 2.7.

Листинг 2.7. Метод top

```
function top()
  if pairs.isEmpty() then error()
  p ← pairs.removeLast()
  if pairs.isEmpty() then
    return p.element
  else
    (element, priority) ← pairs[0]
    pairs[0] ← p
    pushDown(pairs, 0)
    return element
```

Проверяем, не пустая ли куча. Если куча пустая — генерируем ошибку (или вернуть null)

Удаляем и сохраняем во временной переменной последний элемент из массива pairs

Еще раз проверяем, остались ли еще элементы; если не осталось, то просто возвращаем p

Иначе сохраняем верхнюю пару (первую в массиве pairs) во временных переменных...

...и перезаписываем первую пару в массиве ранее сохраненной парой p

Последний элемент был листом, поэтому, вероятно, имел низкий приоритет. Теперь, оказавшись в корне, он может нарушить свойства кучи, поэтому нужно перемещать его к листьям, пока оба его дочерних элемента не будут иметь более низкий приоритет, чем он

Первым делом проверяем наличие непустой кучи. Если куча пустая, то из нее нельзя извлечь верхний элемент, и в этом случае нужно сгенерировать ошибку.

Идея состоит в том, чтобы заменить другим элементом «дыру» в массиве, оставшуюся после удаления корня кучи.

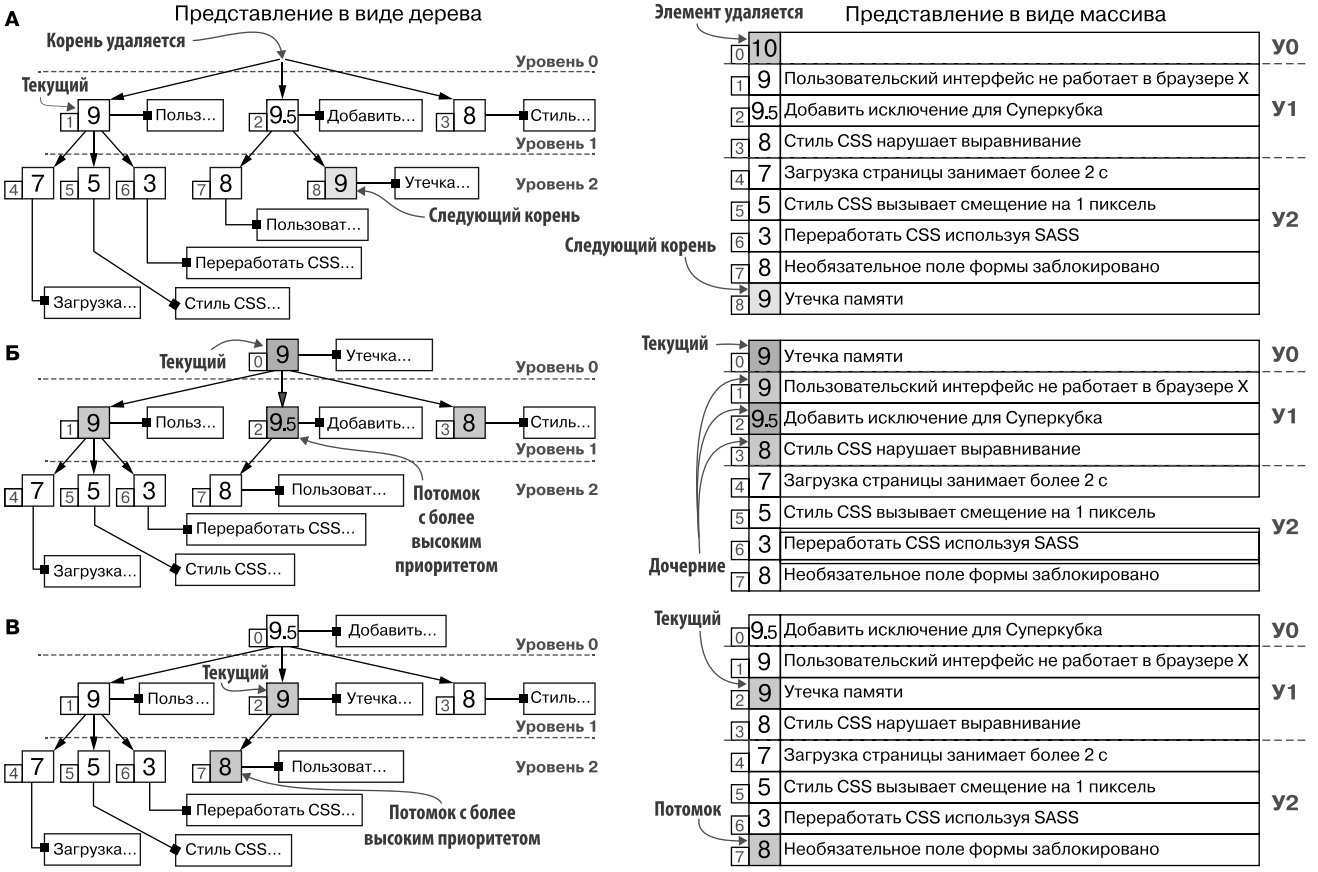


Рис. 2.10. Удаление верхнего элемента из двоичной кучи. *А*. Из допустимой тернарной кучи алгоритм удаляет ее корень (сохраняя его во временном месте, чтобы вернуть в конце). *Б*. Прежний корень замещается последним элементом массива (который также удаляется из его конца). Новый корень может нарушать свойства кучи (что и происходит в этом примере), поэтому нужно протолкнуть его вниз, сравнив с дочерними элементами, чтобы гарантировать соблюдение свойств кучи. *В*. Новый корень перемещается вниз к листьям кучи, пока для него не найдется допустимое место (первое не нарушающее свойств кучи)

Можно было бы «назначить» новым корнем одного из потомков, как показано в листинге 2.7. (Один из них уж точно будет следующим по уровню приоритета элементом; попробуйте доказать это самостоятельно.)

Однако таким способом мы просто переместим проблему в поддерево с корнем в узле, назначенном корнем, затем на следующий уровень и т. д.

Вместо этого можно просто извлечь последний элемент массива и использовать его для замены корня, поскольку нам также нужно сжать массив, и вообще добавлять или удалять элементы массива проще в его хвосте.

Этот новый корень может нарушить свойства кучи. На самом деле, учитывая, что элемент является листом, вероятность нарушения довольно высока. Такое нарушение нужно исправить с помощью метода `pushDown`.

ВРЕМЯ ВЫПОЛНЕНИЯ МЕТОДА TOP()

По аналогии со вставкой, учитывая природу кучи и тот факт, что при каждой перестановке мы перемещаемся на один шаг ближе к листьям, в худшем случае потребуется не более логарифмического числа перестановок.

Конкретные реализации также должны учитывать размер массива и, возможно, гарантировать только *амортизированную* логарифмическую верхнюю границу (если нужна наихудшая логарифмическая граница, то следует использовать статические массивы со всеми присущими им недостатками).

Вернемся к примеру управления задачами и посмотрим, что происходит при вызове `top()` для троичной кучи, изображенной на рис. 2.9, *Б*.

Во-первых, в строке 3 удаляется последний элемент кучи (с индексом 8) и сохраняется во временной переменной; в строке 4 проверяется наличие элементов в оставшемся массиве. Если массив пуст, то это означает, что последний элемент в массиве был вершиной кучи (ее единственным элементом!) и можно просто вернуть его.

Но в данном примере это не так, поэтому происходит переход к строке 7. Она сохраняет первый элемент кучи, `<Незашифрованный пароль в БД, 10>`, во временную переменную, которая будет возвращена методом. Теперь куча выглядит так, как показано на рис. 2.10, *А*, — три несвязанные ветви без корня. Чтобы «сшить» их, добавляется новый корень — элемент, который был сохранен в строке 3, прежде бывший последним в массиве (с индексом 8). На рис. 2.10, *Б* показана ситуация, сложившаяся к этому моменту.

Однако новый корень нарушает свойства кучи, поэтому нужно вызвать `pushDown` в строке 8, чтобы исправить нарушение, поменяв его местами со вторым дочерним элементом, чей приоритет равен 9,5, и создать кучу, показанную на рис. 2.10, *В*. Метод

pushDown выполнит этот этап и остановится, когда будет найдена правильная позиция для элемента <Утечка памяти, 9>.

В заключение стоит отметить, что метод peek возвращает верхний элемент кучи, но не удаляет его и не создает никаких побочных эффектов в структуре данных. Поэтому его логика тривиальна, и мы не будем рассматривать его реализацию.

2.6.5. Метод update

Метод update — самый интересный из общедоступных методов кучи, даже притом что он не всегда явно присутствует в реализациях[1].

При замене элемента его приоритет может не измениться, и в этом случае не нужны никакие дополнительные действия (листинг 2.8). Но он ведь может уменьшиться или увеличиться. Если приоритет элемента увеличился, то нужно проверить, не нарушает ли он третий инвариант относительно своих родителей; если уменьшился, то может нарушиться тот же инвариант относительно потомков.

Листинг 2.8. Метод update

```
function update(oldValue, newPriority)
  index ← pairs.find(oldValue)
  if index ≥ 0 then
    oldPriority ← pairs[index].priority
    pairs[index] ← Pair(oldValue, newPriority)
    if (newPriority < oldPriority) then
       bubbleUp(pairs, index)
    elsif (newPriority > oldPriority) then
      pushDown(pairs, index)
```

Получаем позицию элемента для обновления

Проверяем, действительно ли элемент хранится в куче, и обновляем его новым значением/приоритетом

Проверяем, выше ли новый приоритет старого

Если выше, поднимаем элемент выше к корню

Иначе опускаем ближе к листьям

В первом случае нужно продолжать поднимать измененный элемент, пока не обнаружится предок с более высоким приоритетом или пока не будет достигнут корень. Во втором случае надо продолжать опускать его вниз, пока все его дочерние элементы не будут иметь более низкий приоритет или не будет достигнут лист.

Как уже было показано, первый случай всегда можно реализовать более эффективно[2], и, к счастью, большинство алгоритмов требуют только уменьшения приоритета элементов.

---

[1] Например, в Java-классе PriorityQueue, представленном в стандартной библиотеке, чтобы выполнить ту же операцию, нужно удалить элемент, а затем вставить новый. Это неидеальное решение, тем более что удаление произвольного элемента реализовано здесь неэффективно и имеет линейное время выполнения.

[2] Используя кучи Фибоначчи, эту операцию можно реализовать даже с амортизированным постоянным временем выполнения, по крайней мере в теории.

ВРЕМЯ ВЫПОЛНЕНИЯ МЕТОДА UPDATE()

С точки зрения производительности проблема этого метода кроется в строке 2: в худшем случае поиск старого элемента может занять линейное время, потому что если элемент не находится в куче, то придется обойти всю кучу.

Чтобы повысить эффективность поиска в этом наихудшем сценарии, можно использовать вспомогательные структуры данных. Например, можно хранить ассоциативный массив `map`, связывающий каждый элемент в куче с его индексом. Если реализовать поиск с помощью хеш-таблицы, то для него потребуется амортизированное время O(1).

Более подробно этот вопрос мы обсудим, когда будем знакомиться с методом `contains`. А пока просто вспомним, что если операция поиска занимает время не более длительное, чем логарифмическое время, то и для выполнения самого метода требуется логарифмическое время.

2.6.6. Обработка дубликатов

До сих пор предполагалось, что куча не содержит дубликатов. Если это предположение не соответствует действительности, придется предусмотреть решение дополнительных проблем. В частности, нужно выяснить порядок, используемый в случае дублирования. Чтобы показать, почему это важно, возьмем в качестве примера более простую конфигурацию, изображенную на рис. 2.11. Для экономии места покажем только приоритеты в узлах.

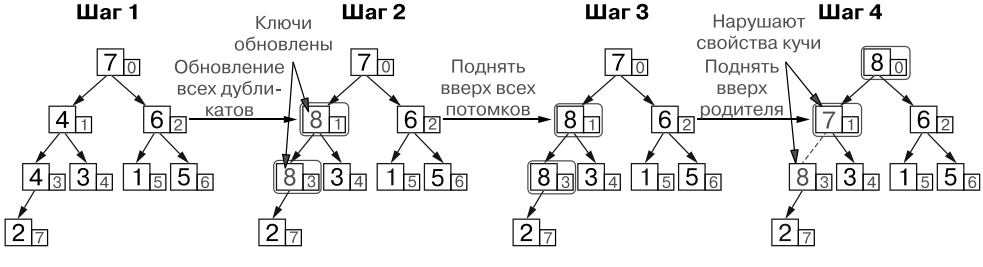


Рис. 2.11. Обновление элемента в двоичной невозрастающей куче с дубликатами

Здесь имеется два дубликата — назовем их `X` и `Y`, — и один из них является потомком другого. Рассмотрим случай, когда `X` является потомком `Y` и вызывается `update`, чтобы изменить их приоритет на более высокий. Внутри `update` после обновления приоритетов двух элементов дважды должен быть вызван метод `bubbleUp` — один раз для `X` и один для `Y`. Загвоздка в том, что, если выполнить вызовы в неправильном порядке, куча окажется в несогласованном состоянии, нарушающем ее свойства.

Предположим, что сначала поднимается `X`, потомок. Тогда для него сразу же будет найден родитель (то есть `Y`) и подъем остановится (поскольку он имеет тот же приоритет). Следующий вызов `bubbleUp(Y)` обнаружит, что родитель элемента `Y` имеет более низкий приоритет и поэтому `Y` нужно переместить ближе к корню. К сожалению, после этого элемент `X` больше не будет проверяться и, соответственно, не будет перемещен вверх вместе с `Y`, в результате чего окажется нарушенным свойство кучи.

На рис. 2.11 по шагам описывается, как метод `update` может нарушить ограничения кучи, если не предусматривает обработку дубликатов:

- на шаге 1 показано начальное состояние невозрастающей кучи;
- на шаге 2 все вхождения 4 меняются на 8;
- на шаге 3 производится подъем самого глубокого узла с индексом 3. Подъем немедленно прекращается, потому что родитель имеет такой же приоритет (он только что обновился вместе с приоритетом текущего узла). Узлы, передаваемые в вызов `bubbleUp`, выделены красным контуром;
- на шаге 4 поднимается другой узел с индексом 1. На этот раз действительно выполняются некоторые замены, потому что новый приоритет узла (8) выше, чем приоритет его родителя (7). Узел, созданный первым на шаге 3, больше никогда не будет обновляться, поэтому свойства кучи нарушатся.

Есть ли решение этой проблемы? Да, есть. Если вызовы `bubbleUp` будут следовать в направлении слева направо, свойства кучи будут соблюдены.

Как вариант можно изменить условия в `bubbleUp` и `pushDown` и останавливаться, только когда будут найдены родитель со строго более высоким приоритетом и потомки со строго более низким приоритетом соответственно. Еще одно решение — подъем (или опускание) узлов в процессе их обновления. Оба решения, однако, далеки от идеала по многим причинам, и не в последнюю очередь из-за низкой эффективности, потому что они увеличивают количество перестановок в худшем случае (доказать это достаточно просто, если посчитать количество перестановок в наихудшем случае, когда все элементы имеют одинаковый приоритет; разобраться с этим вы сможете самостоятельно).

2.6.7. Преобразование в кучу

Очереди с приоритетами можно создавать пустыми, но нередко они инициализируются набором некоторых элементов. Чтобы получить инициализированную кучу с n элементами, можно сначала создать пустую кучу, а затем последовательно добавить в нее элементы.

Чтобы выделить место для кучи, а затем вставить n элементов, потребуется времени не более $O(n)$. Вставка каждого элемента занимает логарифмическое время, поэтому верхняя граница равна $O(n \log n)$. Но является ли этот показатель лучшим? Как оказывается, инициализацию можно выполнить еще эффективнее. Предположим, что требуется инициализировать кучу множеством из n элементов, следующих в произвольном порядке.

Как мы уже видели, каждую позицию массива можно рассматривать как корень подкучи. Так, например, листья — это тривиальные подкучи, содержащие только один элемент, они являются действительными кучами.

Сколько листьев в куче? Это зависит от коэффициента ветвления. В двоичной куче половина узлов, а в четверичной три четверти — листья.

Остановимся для простоты на случае двоичной кучи. На рис. 2.12 показан пример неубывающей кучи, на котором можно шаг за шагом проследить, как работает метод `heapify` преобразования коллекции узлов в кучу. Если начать с последнего внутреннего узла кучи — назовем его `X`, — у него будет не более двух дочерних узлов, и оба они — листья. Следовательно, оба представляют допустимые кучи.

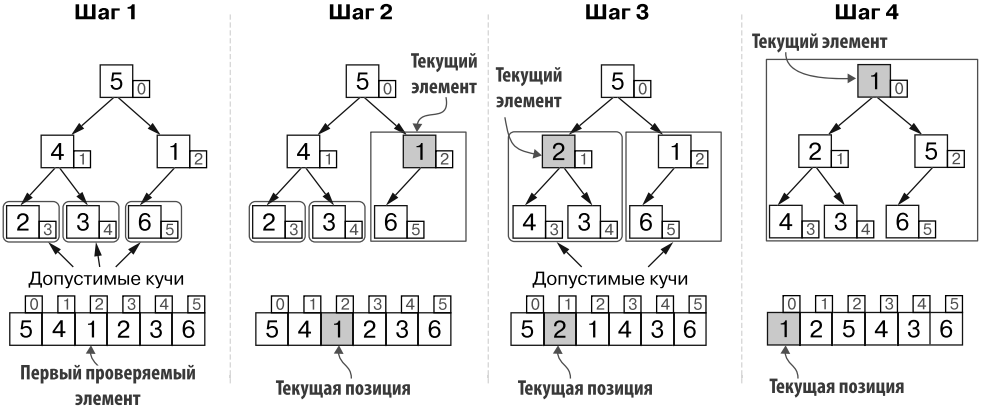


Рис. 2.12. Преобразование небольшого массива в кучу. Контурами с закругленными углами обведены действительные подкучи, то есть части дерева, в отношении которых соблюдаются свойства кучи. *Шаг 1*: изначально допустимыми неубывающими кучами являются только листья; здесь меньшие числа означают более высокий приоритет. *Шаг 2*: элемент, с которого начинается итерация, является первым внутренним узлом в куче с индексом 2 в массиве. Он отмечен стрелкой внизу на крайнем левом рисунке. Восстанавливаем свойства кучи для подкучи с корнем в этом узле, меняя местами содержимое этого узла и его потомка. *Шаг 3*: перемещаем указатель на текущий элемент на одну позицию влево и повторяем операцию для подкучи с корнем в индексе 1. *Шаг 4*: наконец весь массив преобразуется в кучу путем перемещения вниз временного корня (с приоритетом 5) до правильного местоположения

Пока нельзя сказать, является ли `X` корнем допустимой кучи, но можно попытаться сдвинуть его вниз и, вероятно, поменять местами с наименьшим из его дочерних элементов. После этого можно быть уверенными, что подкуча с корнем в `X` является допустимой кучей, потому что `pushDown` по своей природе восстанавливает свойства кучи в подкуче, где все дочерние элементы являются допустимыми кучами и только корень может быть не на своем месте.

Далее, переместившись в массиве на одну позицию влево, можно повторить процедуру и получить еще одну допустимую подкучу. Продолжая действовать таким образом, мы в какой-то момент обойдем все внутренние узлы, потомки которых являются

исключительно листьями. После этого можно перейти к первому внутреннему узлу, который является корнем подкучи высотой 2. Назовем его `Y`. Если обрабатывать элементы массива в порядке справа налево, то можно гарантировать, что потомки элемента `Y` — это подкучи, которые:

- имеют высоту не более 1;
- соблюдают свойства кучи.

И снова можно воспользоваться методом `pushDown`, чтобы сделать кучу с корнем в `Y` действительной кучей.

Если повторить эти шаги для всех узлов, то по достижении корня кучи мы сможем с уверенностью утверждать, что получили действительную кучу. Реализация метода `heapify` показана в листинге 2.9.

Листинг 2.9. Метод heapify

```
function heapify(pairs)
  for index in {(|pairs|-1)/D .. 0} do
    pushDown(pairs, index)
```

Обход начинается с первого внутреннего узла кучи (D — это коэффициент ветвления) и продолжается до достижения корня

Гарантирует, что подкуча с корнем в index является допустимой кучей

ВРЕМЯ ВЫПОЛНЕНИЯ HEAPIFY

Для реорганизации двоичной кучи метод `pushDown` будет вызван n / 2 раз, что в худшем случае потребует $O(n \times \log(n))$ перестановок.

Но обратите внимание, что подкучи, в которых все дочерние элементы являются листьями, имеют высоту 1, поэтому для их исправления требуется только одна перестановка. А таких элементов n / 2. Аналогично для подкуч высотой 2 требуется не более чем две перестановки, а их у нас n / 4.

С приближением к корню количество перестановок увеличивается, но количество вызовов `pushDown` уменьшается с той же скоростью. Наконец, будет сделано только два вызова `pushDown`, которым потребуется не более $\log_2(n) - 1$ перестановок, и только один вызов, для корня, которому в худшем случае потребуется $\log_2(n)$ перестановок.

Учитывая все это, получаем общее количество перестановок:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{h}{2^h} \right\rceil \right).$$

Так как последняя сумма ограничена геометрическим рядом с начальным числом 2, то:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{h}{2^h} \right\rceil \leq \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil = 2,$$

и, следовательно, общее количество перестановок в худшем случае линейно (не более $2n$).

Из сказанного следует, что сложность `heapify` составляет $O(n)$.

Вычисления для d-ичной кучи я оставляю вам в качестве упражнения. Они выполняются аналогично, меняется только коэффициент ветвления.

2.6.8. Методы вне API: contains

Одна из операций, крайне неэффективных при работе с кучами, — проверка присутствия некоторого элемента. Для ее исполнения нет другого выбора, кроме как обойти все элементы, пока не найдется искомый элемент или не будет достигнут конец массива. Это означает, что алгоритм поиска имеет линейное время выполнения. Сравните его с оптимальным средним постоянным временем для хеш-таблиц или даже с $O(\log(n))$ для двоичных деревьев поиска (в среднем) и для сбалансированных двоичных деревьев поиска (наихудший случай).

Кроме того, было бы желательно иметь возможность увеличивать/уменьшать приоритет. Как было показано в подразделе 2.6.5, и для той и для другой операции первостепенное значение имеет эффективное извлечение элемента, приоритет которого нужно изменить. Поэтому при реализации кучи можно добавить вспомогательное поле с хеш-таблицей, связывающей элементы с их позициями и позволяющей проверить присутствие элемента в куче (или получить его позицию) в среднем за постоянное время. Одна из возможных реализаций `contains` приводится в листинге 2.10.

Листинг 2.10. Метод contains

```
function contains(elem)
  index ← elementToIndex[elem]
  return index >= 0
```

Функция использует дополнительное поле `elementToIndex`, которое можно добавить в кучу. В своей работе она опирается на два предположения:

- `elementToIndex[elem]` по умолчанию возвращает `-1`, если `elem` отсутствует в куче;
- в куче отсутствуют повторяющиеся значения, иначе придется хранить список индексов для каждого ключа.

2.6.9. Еще раз о производительности

Описанием метода `contains` мы завершили обзор этой структуры данных — кучи.

Вы познакомились с несколькими операциями над ней, и сейчас самое время вспомнить длительности их выполнения (табл. 2.4) и, что особенно важно, определить необходимый им объем дополнительной памяти.

В информатике часто приходится чем-то жертвовать — временем или объемом памяти. В неформальном анализе принято пренебрегать фактором дополнительной памяти, однако с теми объемами, которыми приходится оперировать в эпоху больших данных, структуры данных должны хранить и обрабатывать миллиарды элементов или даже больше. Поэтому быстрый алгоритм, использующий квадратичный объем памяти, может оказаться лучшим выбором для небольших объемов данных, но непрактичным для некоторых реальных сценариев обработки гигантских объемов

данных. По этой причине более медленный алгоритм, использующий постоянный или логарифмический объем памяти, с ростом набора данных может оказаться предпочтительнее.

Таблица 2.4. Операции и длительности их выполнения для куч с n элементами

Операция	Время выполнения	Объем дополнительной памяти
`insert`	O(log(*n*))	O(1)
`top`	O(log(*n*))	O(1)
`remove`	O(log(*n*))*	O(1)*
`peek`	O(1)	O(1)
`contains` (простейшая реализация)	O(*n*)	O(1)
`contains`	O(1)*	O(*n*)*
`updatePriority`	O(log(*n*))	O(*n*)*
`heapify`	O(*n*)	O(1)

* При использовании улучшенной версии `contains` с поддержкой дополнительной хеш-таблицы, связывающей элементы с их позициями.

Кроме того, при проектировании крупномасштабных систем становится особенно важным начинать учитывать объемы расходуемой памяти как можно раньше.

Для работы с кучами, естественно, потребуется постоянное дополнительное пространство для каждого элемента и линейное дополнительное пространство в целом для размещения хеш-таблицы, отображающей элементы в их индексы.

Мы внимательно изучили все операции. Тем не менее стоит сделать еще пару замечаний.

Для методов `insert` и `top` гарантируется амортизированное время работы, а не худшее. Если для большей гибкости используются динамические массивы, то некоторые вызовы потребуют линейного времени для изменения размера массива. Можно доказать, что для заполнения динамического массива n элементами требуется не более $2n$ перестановок. Однако логарифмическая гарантия наихудшего случая может быть предложена, только если размер кучи задан с самого начала. По этой причине и для эффективного управления памятью в некоторых языках (например, Java) рекомендуется инициализировать кучи их ожидаемыми размерами, если, конечно, есть разумная оценка, которая остается верной в течение большего времени жизни контейнера.

Производительность `remove` и `updatePriority` зависит от эффективности реализации `contains`. Однако для реализации эффективного поиска необходимо хранить, кроме самого массива, еще одну структуру данных, обеспечивающую быструю косвенную адресацию. На выбор у нас есть хеш-таблицы и фильтр Блума (см. главу 4).

При использовании любой из этих структур время выполнения `contains` считается постоянным, с одной оговоркой: хеш для каждого элемента должен быть вычисляем за постоянное время. Иначе нужно учесть и эти дополнительные затраты.

2.6.10. От псевдокода к реализации

В предыдущих разделах показано, что работа d-ичной кучи не зависит от языка программирования. Псевдокод дает хороший способ наметить и объяснить структуру данных, не беспокоясь о деталях реализации, и сосредоточиться на ее поведении.

В то же время псевдокод почти не имеет практического применения. Чтобы перейти от теории к практике, нужно выбрать язык программирования и реализовать d-ичную кучу. Независимо от выбранной платформы неизбежно возникнут проблемы, связанные с конкретным языком, и появится необходимость учитывать различные особенности.

В этой книге представлены реализации алгоритмов, чтобы дать читателям возможность поэкспериментировать и опробовать эти структуры данных.

Полный код, включая тесты, можно найти в репозитории нашей книги на GitHub[1].

2.7. ПРИМЕР: ПОИСК K НАИБОЛЬШИХ ЭЛЕМЕНТОВ

В этом разделе будет показано, как можно использовать приоритетную очередь для поиска k наибольших элементов в множестве.

Для готового множества из n элементов у нас есть несколько альтернатив, которым не нужна никакая вспомогательная структура данных.

- Можно отсортировать входные данные и взять последние k элементов. Это простейшее решение требует $O(n \log(n))$ сравнений и перестановок и в зависимости от алгоритма может потребовать дополнительную память.
- Можно найти самый большой элемент в множестве и переместить его в конец массива, затем просмотреть оставшиеся $n - 1$ элементов, найти второй по величине элемент и переместить его в позицию $n - 2$ и т. д. По сути, этот алгоритм выполняет k раз внутренний цикл алгоритма `Selection Sort`, требуя $O(k)$ перестановок и $O(n \times k)$ сравнений. Дополнительная память не требуется.

В этом разделе вы увидите, что, используя кучу, можно достичь цели, применив $O(n + k \log(k))$ сравнений и перестановок, а также $O(k)$ дополнительной памяти. Это кардинальное улучшение для случая, когда k намного меньше n. В типичной ситуации n может быть порядка миллионов или миллиардов элементов, а k — от сотен до нескольких тысяч.

Кроме того, с помощью вспомогательной кучи алгоритм можно естественным образом адаптировать для работы с динамическими потоками данных, а также позволить потреблять элементы из кучи.

---

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#d-ary-heap.

2.7.1. Правильная структура данных...

Когда задача заключается в поиске подмножества наибольших/наименьших элементов, очереди с приоритетами кажутся естественным решением.

В программировании выбор правильной структуры данных может сыграть решающую роль[1]. Но этого недостаточно — ее нужно также правильно использовать.

Предположим, например, что имеется статический набор элементов, доступных с самого начала. Можно использовать невозрастающую кучу, добавить в нее все n элементов, а затем извлечь из этой кучи k наибольших.

Итого, для кучи потребуется $O(n)$ дополнительной памяти. Чтобы создать такую кучу из полного набора за линейное время $O(n)$, надо применить метод `heapify` и затем k раз вызвать метод `top` со стоимостью $O(\log(n))$ каждого вызова. Общая стоимость этого решения составит $O(n + k \log(n))$ сравнений и перестановок.

Это уже лучше, чем прямолинейное решение, но если подумать, то в случае $n >> k$[2] при выборе такого алгоритма работы будет создана огромная куча только для того, чтобы извлечь несколько элементов. А это выглядит уже как неоправданное расточительство.

2.7.2. ...И правильное ее использование

Итак, наша конечная цель — получить небольшую кучу с k элементами. Использование невозрастающей кучи не позволяет достичь ее без огромных затрат. Разберемся в причинах этого на примере.

Предположим, что нужно найти три самых больших числа в последовательности: 2, 4, 1, 3, 7, 6. Добавляем первые три и получаем такую невозрастающую кучу: [4, 2, 1]. Затем переходим к следующему числу в списке, это 3. Оно больше двух из трех элементов, находящихся в настоящее время в куче, но мы не знаем этого, потому что можем видеть только вершину кучи. После добавления числа 3 в кучу получаем [4, 3, 1, 2]. Теперь, чтобы сохранить размер кучи равным k элементам, нужно удалить минимальный. Как узнать, где он находится внутри кучи? Нам известно только, где находится максимум (на вершине невозрастающей кучи), а поиск минимума может занять линейное время (даже если учесть, что минимум находится в одном из листьев, к сожалению, их количество линейно определяется пропорцией n / D).

Вы можете проверить сами, что, даже вставляя элементы в другом порядке, окажетесь в похожей ситуации.

Загвоздка в том, что для получения k наибольших элементов на каждом шаге нужно определять, есть ли в куче из k уже имеющихся элементов число, которое меньше текущего. Соответственно, вместо невозрастающей кучи можно использовать неубывающую кучу, хранящую k наибольших из найденных на данный момент элементов.

---

[1] Впрочем, обычно наибольшую тревогу вызывает обратное — выбор неправильной структуры данных.

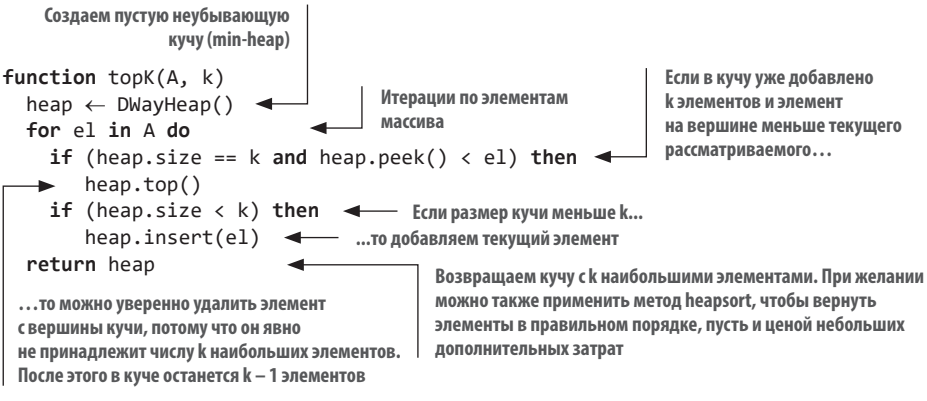
[2] $n >> k$ обычно интерпретируется как «n намного больше k».

В этом случае можно сравнивать каждый новый элемент с вершиной кучи, и если он меньше, то можно с уверенностью утверждать, что он не является одним из k наибольших элементов. Если новый элемент больше элемента на вершине кучи (то есть *наименьшего* из имеющихся k элементов), то остается только удалить вершину из кучи, а затем добавить в нее новое значение. Таким образом, обновление кучи в каждой итерации будет производиться за постоянное время[1] вместо линейного в невозрастающей куче.

2.7.3. Реализация

Интересное решение, верно? И простое в реализации. Поскольку код стоит тысячи слов, посмотрим на нашу кучу в действии, как показано в листинге 2.11.

Листинг 2.11. Выбор наибольших k элементов из списка



2.8. ДРУГИЕ СЛУЧАИ ИСПОЛЬЗОВАНИЯ

Куча — одна из самых универсальных структур данных. Вместе со стеком и очередью она составляет основу почти каждого алгоритма обработки входных данных в определенном порядке.

Замена двоичной кучи d-ичной кучей может ускорить практически любой код, использующий приоритетную очередь. Прежде чем углубляться в алгоритмы, которые могут оказаться наиболее выгодными при использовании куч, обязательно познакомьтесь с графами, потому что большинство подобных алгоритмов затрагивает эту структуру данных. Для вашего удобства краткое введение в графы дается в главе 14.

Теперь обсудим некоторые алгоритмы, которые могут выиграть от использования куч.

---

[1] Время $O(\log(k))$, если быть точным, но поскольку k в этом контексте является константой (намного меньшей, чем n, и не зависит от n), то можно заключить, что $O(\log(k)) == O(1)$.

2.8.1. Поиск кратчайшего пути в графе: алгоритм Дейкстры

Очереди с приоритетами имеют решающее значение в реализациях алгоритмов Дейкстры и A*, подробно описанных в главе 14, в разделах 14.4 и 14.5. На рис. 2.13 показан пример простого графа, иллюстрирующий концепцию кратчайшего пути между двумя вершинами. Как вы узнаете в главе 14, время работы этих фундаментальных алгоритмов на графах, определяющих кратчайшее расстояние до цели, сильно зависит от реализации приоритетной очереди и переход от двоичной кучи к *d*-ичной может обеспечить стабильное ускорение.

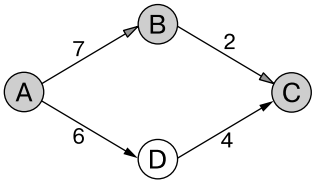


Рис. 2.13. Кратчайший путь между вершинами A и C в ориентированном графе

2.8.2. Еще о графах: алгоритм Прима

Алгоритм Прима вычисляет минимальное остовное дерево (Minimum Spanning Tree, MST) неориентированного связного графа `G`.

Предположим, что `G` имеет *n* вершин. Тогда минимальное остовное дерево `G` — это:

- дерево (связный, неориентированный, ациклический граф);
- подграф графа `G` с *n* вершинами, имеющий минимальную сумму весов ребер среди всех подграфов графа `G`, которые также являются деревьями и охватывают все *n* вершин.

Рассмотрим граф из примера в подразделе 2.8.1. Его минимальным остовным деревом будет дерево, изображенное на рис. 2.14.

Алгоритм Прима работает точно так же, как алгоритм Дейкстры, но:

- не учитывает расстояние от источника;
- сохраняет ребро, соединяющее прежде посещенную вершину со следующей ближайшей вершиной;
- вершина, используемая алгоритмом Прима как «источник», будет корнем MST.

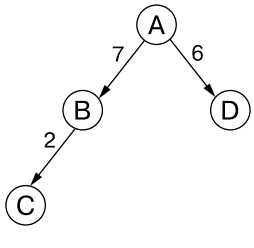


Рис. 2.14. Остовное дерево для графа на рис. 2.13

Неудивительно, что он имеет такое же время работы, как и алгоритм Дейкстры:

- $O(V^2)$ при использовании массивов (сортированных или несортированных) для очереди с приоритетами;
- $O(V \log(V) + E \log(V))$ при использовании двоичной или d-ичной кучи;
- $O(V \log(V) + E)$ при использовании кучи Фибоначчи.

2.8.3. Сжатие данных: коды Хаффмана

Алгоритм Хаффмана является, пожалуй, самым известным алгоритмом сжатия данных, и вы, возможно, уже слышали о нем, если изучали курс «Введение в информатику». Этот простой, блестящий, *жадный* алгоритм, даже притом что больше не считается современным, в 50-х годах был значительным прорывом.

Код Хаффмана — это дерево, строящееся снизу вверх, начиная со списка различных символов, встречающихся в тексте, и их частот. Алгоритм итеративно:

1) выбирает и удаляет два элемента в списке с наименьшей частотой;
2) затем создает новый узел, объединяя их (суммируя две частоты);
3) и наконец, добавляет новый узел в список.

Само дерево не является кучей, но ключевой шаг алгоритма основан на эффективном извлечении наименьших элементов из списка и добавлении новых элементов в список. Вы, наверное, уже догадались, что здесь на помощь приходят кучи.

Взгляните на сам алгоритм, показанный в листинге 2.12.

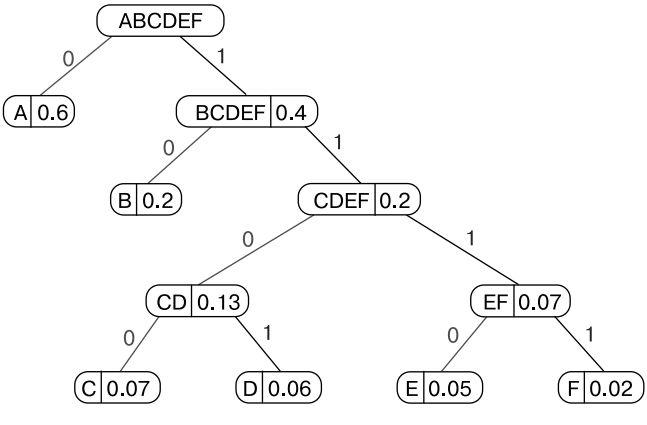


Рис. 2.15. Дерево кодирования Хаффмана, построенное из таблицы частот символов: A = 0.6, B = 0.2, C = 0.07, D = 0.06, E = 0.05 и F = 0.02

Предполагается, что входными данными для алгоритма является текст, хранящийся в строке (конечно, на самом деле текст может храниться в файле или потоке, но мы

всегда можем преобразовать его в строку[1]), а выходными данными — отображение символов в двоичные последовательности.

Первая подзадача, которую нужно решить, — преобразование текста: необходимо просчитать некоторую статистику, чтобы определить наиболее и наименее часто используемые символы в нем. С этой целью вычислим частоту появления символов в тексте[2].

Детали метода `ComputeFrequencies`, вызываемого в строке 2, выходят за рамки обсуждения и достаточно просты (по крайней мере, в его базовой версии), поэтому мы не будем углубляться в этот вспомогательный метод.

После вычисления карты частот создаем новую очередь с приоритетами, а затем в строках 4 и 5 перебираем содержимое карты частот, создаем новый `TreeNode` для каждого символа, добавляя его в очередь с приоритетами, как показано в листинге 2.12. Очевидно, учитывая тему этой главы, что для представления очереди стоит использовать кучу, в частности неубывающую кучу (min-heap), верхний элемент которой — это элемент с наименьшим значением поля приоритета. И в этом случае поле приоритета (что неудивительно) является полем частоты в `TreeNode`.

Листинг 2.12. Алгоритм кодирования Хаффмана

```
function huffman(text)
  charFrequenciesMap ← ComputeFrequencies(text)
  priorityQueue ← MinHeap()
  for (char, frequency) in charFrequenciesMap do
    priorityQueue.insert(TreeNode([char], frequency))
  while priorityQueue.size > 1 do
    left ← priorityQueue.top()
    right ← priorityQueue.top()
    parent ← TreeNode(left.chars + right.chars,
                      left.frequency + right.frequency)
    parent.left ← left
    parent.right ← right
    priorityQueue.insert(parent)
  return buildTable(priorityQueue.top(), [], Map())
```

Каждый `TreeNode` фактически содержит два поля (помимо указателей на его дочерние элементы): набор символов и частоту их появления в тексте, вычисленную как сумму частот отдельных символов.

Если посмотреть на рис. 2.15, можно заметить, что корень получившегося дерева — это набор всех символов в примере текста, и, следовательно, общая частота равна 1.

---

[1] Дополнительные сложности могут возникнуть, если текст не умещается в памяти или рекомендуется использовать подход MapReduce: мы инкапсулируем всю эту сложность в методе ComputeFrequencies.

[2] Вместо фактической частоты, возможно, будет проще подсчитать количество вхождений каждого символа.

Этот набор разбивается на две группы, образующие дочерние элементы корня. Аналогично разбивается каждый внутренний узел, пока не будут достигнуты листья, каждый из которых содержит только один символ.

Но вернемся к алгоритму. На рис. 2.15 можно видеть, как дерево строится снизу вверх, а строки 2–5 в листинге 2.12 выполняют первый шаг, создавая листья дерева и добавляя их в очередь с приоритетами.

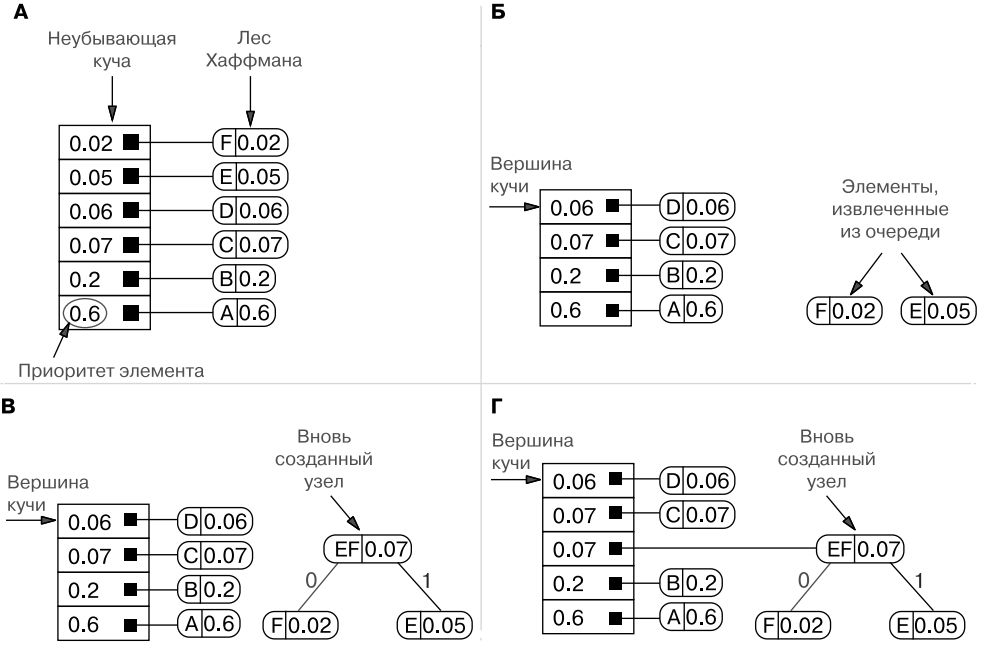


Рис. 2.16. Первый шаг алгоритма кодирования Хаффмана. Как упоминалось в тексте, алгоритм использует две вспомогательные структуры данных: очередь с приоритетами и двоичное дерево. Каждый узел дерева имеет значение — набор символов, встречающихся в тексте, и приоритет — сумму частот этих символов. *А*. Первоначально для каждого символа создается свой узел дерева, хранящий сам символ и его частоту. Каждый узел добавляется также в очередь с приоритетами, где роль приоритета играет частота (чем меньше частота, тем выше приоритет, поэтому здесь используется неубывающая куча (min-heap)). *Б*. Из начала очереди с приоритетами извлекаются два элемента. *В*. Создается новый узел дерева, куда в качестве потомков добавляются два узла, извлеченные на шаге *Б*. По соглашению меньший узел добавляется как левый дочерний узел, а другой — как правый (впрочем, здесь подойдет любое непротиворечивое соглашение). Значением вновь созданного узла становится объединенный набор символов из дочерних элементов, а приоритетом — сумма их приоритетов. *Г*. Новый корень этого поддерева можно добавить обратно в очередь с приоритетами. Обратите внимание, что узлы в куче показаны в отсортированном порядке, но в общем случае порядок хранения узлов в очереди с приоритетами является деталью реализации. Контракт API очередей с приоритетами гарантирует только, что при удалении двух элементов из очереди удалены будут элементы с наименьшими частотами

Далее, начиная со строки 6, располагается ядро алгоритма: мы извлекаем верхние записи `TreeNode` в строках 7 и 8, пока в очереди не останется только один элемент. Как можно видеть на рис. 2.16, *Б*, эти два элемента становятся поддеревьями с наименьшими на данный момент частотами.

Назовем эти поддеревья `L` и `R` (причина выбора таких имен скоро станет ясна).

На рис. 2.16, *В*, показаны действия, выполняемые в строках псевдокода с 9-й по 11-ю: создается новый `TreeNode` (назовем его `P`) путем слияния наборов символов из двух элементов и установки его частоты как суммы частот старых поддеревьев. Затем новый узел и два поддерева объединяются в новое поддерево, где новый узел `P` является корнем, а поддеревья `L` и `R` — его потомками.

Наконец, в строке 12 новое поддерево добавляется обратно в очередь. Как показано на рис. 2.16, *Г*, иногда его можно поместить в начало очереди, даже если это место он займет не по праву. Очередь с приоритетами автоматически позаботится о перемещении узла в нужное место (обратите внимание, что здесь очередь с приоритетами используется как черный ящик, это обсуждалось в разделе 2.4).

Эти шаги повторяются до тех пор, пока в очереди не останется только один элемент (на рис. 2.17 показано еще несколько шагов), и этот последний элемент станет корневым узлом `TreeNode` окончательного дерева.

Затем, в строке 13, он используется для создания таблицы сжатия, которая является конечным результатом работы метода Хаффмана. Таблицу сжатия, в свою очередь, можно использовать для сжатия текста путем преобразования каждого символа в последовательность битов.

Мы не будем рассматривать этот последний шаг[1] и исследуем только шаги, необходимые для создания таблицы сжатия (листинг 2.13) из дерева на рис. 2.15. Этот материал выходит за рамки данной главы, потому что метод не использует очередь с приоритетами. Но краткое объяснение может пригодиться читателям, заинтересованным в реализации алгоритма кодирования Хаффмана.

Листинг 2.13. Алгоритм кодирования Хаффмана (построение таблицы из дерева)

```
function buildTable(node, sequence, charactersToSequenceMap)
  if node.characters.size == 1 then
    charactersToSequenceMap[node.characters[0]] ← sequence
  else
    if node.left <> null then
      buildTable(node.left, 0 + sequence, charactersToSequenceMap)
    if node.right <> null then
      buildTable(node.right, 1 + sequence, charactersToSequenceMap)
  return charactersToSequenceMap
```

[1] Это будет простое отображение 1:1 символов в тексте, надо будет только обратить внимание на эффективное кодирование битов в выводе.

Метод `buildTable` реализован с использованием рекурсии. Как поясняется в приложении Д, рекурсия позволяет создавать более чистый и понятный код, но в некоторых языках лучшую производительность может показать реализация, основанная на явных итерациях.

Метод принимает три аргумента: `TreeNode` — текущий рассматриваемый узел при обходе дерева, последовательность `sequence`, представляющая путь от корня к текущему узлу (где мы добавляем 0 для обозначения «поворота налево» и 1 для «поворота направо»), и карта `Map` с ассоциациями между символами и битовыми последовательностями.

В строке 2 мы проверяем длину набора символов узла. Если он содержит только один символ, это означает, что достигнут лист, и рекурсия прекращается. Битовая последовательность, связанная с символом в узле, представляет путь от корня к текущему узлу, хранящийся в переменной `sequence`.

Если набор содержит несколько символов, то проверяется, есть ли у узла левый и правый потомки (узел будет иметь хотя бы одного потомка, потому что это не лист) и выполняется переход к ним. Важным моментом является конструирование аргумента `sequence` в рекурсивных вызовах: если выполняется переход к левому потомку текущего узла, то в начало `sequence` добавляется 0, а если выполняется переход к правому потомку, то добавляется 1.

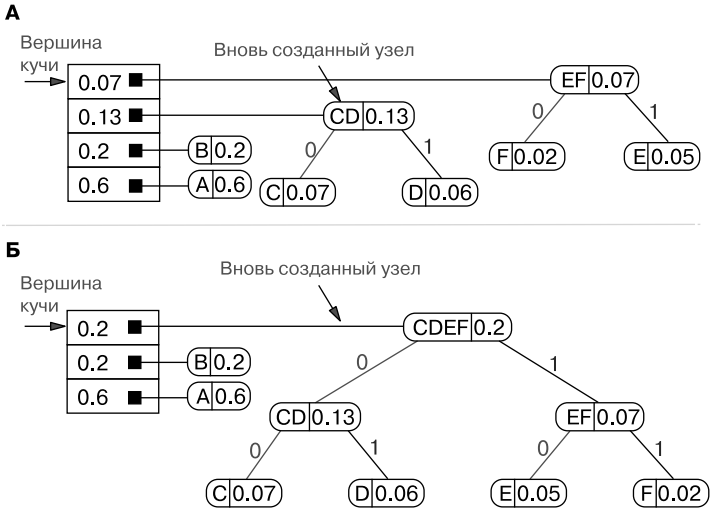


Рис. 2.17. Результат следующих двух шагов алгоритма кодирования Хаффмана. *А*. Из очереди удаляются и объединяются два верхних узла в куче, C и D. В конце этого шага EF и CD становятся двумя наименьшими узлами в куче. *Б*. Эти два узла объединяются в CDEF, и вновь созданный узел добавляется обратно в кучу. Какой именно узел между CDEF и B окажется в начале очереди с приоритетами, является деталью реализации и не имеет значения для алгоритма кодирования Хаффмана (код, конечно, немного изменится, в зависимости от того, какой узел будет извлечен первым, но степень сжатия останется неизменной). Следующие шаги легко представить, используя рис. 2.15 в качестве справки

Таблица 2.5 — не что иное, как таблица сжатия, созданная на основе дерева на рис. 2.15; последний столбец не является частью фактической таблицы сжатия, но полезно понимать, как наиболее часто используемые символы в итоге преобразуются в более короткие последовательности (это и является ключом к эффективному сжатию).

Таблица 2.5. Таблица сжатия, созданная на основе дерева Хаффмана на рис. 2.15

Символ	Последовательность битов	Частота
A	0	0,6
B	10	0,2
C	1100	0,07
D	1101	0,06
E	1110	0,05
F	1111	0,02

Наиболее важное свойство последовательностей состоит в том, что они образуют префиксный код: ни одна последовательность не является префиксом никакой другой последовательности.

Это свойство является ключевым для декодирования: при анализе сжатого текста его легко разбить на символы.

Например, при анализе последовательности `1001101` в сжатом тексте сразу видно, что `10` соответствует символу `B`, следующий `0` — `A` и, наконец, `1101` соответствует `D`, то есть эта сжатая последовательность битов преобразуется в `"BAD"`.

2.9. АНАЛИЗ КОЭФФИЦИЕНТА ВЕТВЛЕНИЯ[1]

Теперь, после знакомства с *d*-ичными кучами, правомерно задать вопрос: не будет ли достаточно обычной двоичной кучи? Дает ли какие-то преимущества более высокий коэффициент ветвления?

2.9.1. Нужны ли d-ичные кучи?

Двоичных куч вполне достаточно для решения всех задач программирования. Основное преимущество этой структуры данных в том, что она гарантирует логарифмическое время выполнения каждой из типичных операций. В частности, основные операции с двоичным сбалансированным деревом гарантированно потребуют количество сравнений, пропорциональное $\log_2(N)$ в худшем случае. Как обсуждается в приложении Б, такая особенность обеспечивает возможность применения этих методов к контейнерам гораздо большего размера, чем в случае, если бы время выполнения было линейным. Учтите, что даже для миллиарда элементов $\log_2(N)$ дает в результате примерно 30.

[1] Этот раздел включает дополнительные понятия.

Как было показано во введении, постоянные множители не имеют значения при оценке времени выполнения, то есть $O(c \times N) = O(N)$, а мы знаем из алгебры, что два логарифма с разными основаниями отличаются только постоянным множителем, в частности:

$$\log_b(N) = \log_2(N) / \log_2(b).$$

В итоге мы имеем:

$$O(\log_2(N)) = O(\log_3(N)) = O(\log(N)).$$

Однако для реализации постоянные множители имеют значение. Они настолько важны, что в некоторых пограничных случаях алгоритмы, которые, согласно оценке времени выполнения, должны быть эффективнее, на самом деле оказываются медленнее более простых алгоритмов с худшими оценками. По крайней мере, это верно для произвольных практических входных данных (например, при сравнении 2^n и $n \times 100^{100\,000}$, чтобы нивелировать влияние постоянного множителя, объем входных данных должен быть огромным).

Ярким примером такого поведения является куча Фибоначчи[1]: теоретически она обеспечивает амортизированное постоянное время выполнения некоторых важных операций, таких как вставка или обновление приоритета, но на практике реализации того и другого получаются сложными и медленными для любого приемлемого объема входных данных.

Присутствие постоянных множителей, как правило, обусловлено несколькими причинами, в том числе такими, как:

- задержка чтения/записи в память (чтение из ограниченной области памяти и из разных мест, далеко отстоящих друг от друга);
- стоимость поддержки счетчиков или итераций циклов;
- стоимость рекурсии;
- детали реализации, которые в асимптотическом анализе не учитываются (например, как уже было показано, использование статических и динамических массивов).

Итак, на данный момент должно быть ясно, что в любой реализации следует стремиться делать постоянные множители как можно меньшей величины.

Рассмотрим следующую формулу:

$$\log_b(N) = \log_2(N) / \log_2(b).$$

---

[1] Куча Фибоначчи — это усовершенствованная версия очереди с приоритетами, реализованная с помощью набора куч. Операции поиска минимума, вставки и обновления приоритета в куче Фибоначчи занимают постоянное амортизированное время O(1), тогда как удаление элемента (включая минимальный) требует амортизированного времени O(log n), где n — размер кучи. То есть теоретически кучи Фибоначчи быстрее любых других куч, но на практике их реализации из-за чрезмерной сложности оказываются медленнее простых двоичных куч.

Если $b > 2$, очевидно, что $\log_b(N) < \log_2(N)$, и, следовательно, если во время выполнения алгоритма проявляется логарифмический множитель и есть возможность представить реализацию, которая потребует $\log_b(N)$ шагов вместо $\log_2(N)$, то очевидно, что при неизменности всех остальных факторов время ее выполнения будет постоянно уменьшаться.

В разделе 2.10 мы дополнительно исследуем, как это правило применимо к *d*-ичным кучам.

2.9.2. Время выполнения

Скажем сразу, что настройка коэффициента ветвления кучи действительно дает преимущества, но ключевым моментом является компромисс.

Чем больше коэффициент ветвления, тем быстрее выполняется вставка, так как в лучшем случае новый элемент будет добавляться в корень с не более чем $O(\log_D(n))$ сравнениями и перестановками.

С другой стороны, увеличение коэффициента ветвления ухудшает производительность удаления элемента и обновления приоритета. Если вспомнить алгоритмы выталкивания элемента, то для каждого узла потребуется сначала найти наивысший приоритет среди всех его дочерних элементов, а затем сравнить его с элементом, который проталкивается вниз.

Чем больше коэффициент ветвления, тем меньше высота дерева (она уменьшается логарифмически с ростом коэффициента ветвления). С другой стороны, количество сравниваемых потомков на каждом уровне растет линейно вместе с коэффициентом ветвления. Нетрудно представить, что с коэффициентом ветвления 1000 алгоритм будет работать неэффективно (и сопоставимо с линейным поиском в массиве с числом элементов менее 1001!).

На практике путем профилирования и тестирования производительности был сделан вывод, что лучшим компромиссом в большинстве ситуаций является выбор $D = 4$.

2.9.3. Поиск оптимального коэффициента ветвления

Если вы ищете значение D, оптимальное для любой ситуации, то вас ждет разочарование. В некоторой степени может помочь теория, показывающая диапазон оптимальных значений. Можно показать, например, что оптимальное значение не может быть больше 5. Или, другими словами, можно математически доказать, что:

- компромисс между вставкой и удалением лучше всего сбалансирован при $2 \le D \le 5$;
- теоретически троичная куча быстрее двоичной;
- четверичная и троичная кучи имеют одинаковую производительность;
- пятеричная куча немного медленнее.

На практике наилучшее значение *D* зависит от деталей реализации и данных, которые будут храниться в куче. Оптимальный коэффициент ветвления можно определить только эмпирически в каждом конкретном случае. Универсального оптимального коэффициента ветвления не существует, он зависит от фактических данных и соотношения вставок/удалений или, например, от дороговизны вычисления приоритета по сравнению с копированием элементов.

По опыту, двоичные кучи никогда не бывают самыми быстрыми, пятеричные кучи редко дают выигрыш в скорости (за исключением узких областей применения), а лучший выбор обычно представляют коэффициенты ветвления 3 и 4, в зависимости от нюансов.

Поэтому я с уверенностью могу предложить начать с коэффициента ветвления 4. А если рассматриваемая структура данных используется в ключевом разделе вашего приложения и небольшое улучшение производительности может иметь существенное значение, то следует организовать возможность настройки коэффициента ветвления и передавать его в виде параметра.

2.9.4. Коэффициент ветвления и память

Обратите внимание, что я не просто так предложил выбрать наибольший коэффициент ветвления из двух самых эффективных. Как оказывается, выбирая оптимальный коэффициент ветвления для кучи, нужно учитывать еще одно обстоятельство: локальность ссылок.

Когда размер кучи больше доступного кэша или памяти и, вообще, когда задействовано кэширование и несколько уровней хранения, операции с двоичной кучей в среднем дают больше промахов кэша или сбоев страниц, чем с *d*-ичной кучей. Это связано с тем, что в ней дочерние элементы хранятся группами и при обновлении или удалении любого узла необходимо проверить все его дочерние элементы. Чем выше коэффициент ветвления, тем ниже и шире становится куча и тем сильнее проявляется принцип локальности.

D-ичная куча является, пожалуй, лучшей традиционной структурой данных, способствующей уменьшению сбоев страниц[1]. За последние годы были предложены новые альтернативы, сокращающие количество промахов кэша и сбоев страниц; одна из них — *косые деревья* (splay trees).

В общем случае эти альтернативы не могут обеспечить такого же баланса между практической и теоретической производительностью, как кучи, но в случаях, когда большой вклад в снижение производительности вносят сбои страниц или обращения к диску, лучше применить линейно-логарифмический[2] алгоритм с более высокой локальностью, чем линейный с плохой локальностью.

---

[1] http://comjnl.oxfordjournals.org/content/34/5/428.

[2] $O(n \log(n))$ для входных данных с n элементами.

2.10. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ: ПОИСК НАИЛУЧШЕГО КОЭФФИЦИЕНТА ВЕТВЛЕНИЯ

Мы обсудили теорию. Теперь попробуем применить ее на практике и посмотрим, как правильно профилировать реализацию структуры данных и приложения.

Уже было показано, что очереди с приоритетами являются ключевым компонентом алгоритма сжатия Хаффмана. Если оценивать производительность методов кучи количеством выполненных перестановок, то можно обнаружить, что на каждый вызов метода приходится не более h перестановок, где h — высота кучи. В разделе 2.5 также продемонстрировано, что высота d-ичной кучи как полного сбалансированного дерева точно равна $\log_D(n)$[1].

Казалось бы, производительность методов `insert` и `top` должна расти с увеличением коэффициента ветвления и уменьшением высоты кучи.

Но простого ограничения количества перестановок недостаточно. В разделе 2.9 мы проанализировали производительность этих двух методов и приняли во внимание также количество обращений к массиву или, что то же самое, количество сравнений элементов кучи, которое должен выполнить каждый метод. Если `insert` обращается только к одному элементу на каждом уровне кучи, то метод `top` выполняет обход дерева от корня к листьям и на каждом уровне должен обойти список дочерних элементов узла. Соответственно, ему потребуется примерно до $D \log_D(n)$ обращений к куче с коэффициентом ветвления D, содержащей n элементов.

В табл. 2.6 и 2.7 приводятся обобщенные результаты анализа производительности трех основных методов куч.

Таблица 2.6. Основные операции, поддерживаемые d-ичными кучами, и количество выполняемых ими перестановок (в куче с n элементами)

Операция	Количество перестановок	Дополнительная память
`Insert`	$\sim\log_D(n)$	O(1)
`Top`	$\sim\log_D(n)$	O(1)
`Heapify`	$\sim n$	O(1)

В случае с методом `top` увеличение коэффициента ветвления не всегда приводит к улучшению производительности, потому что, несмотря на уменьшение $\log_D(n)$, величина D становится больше. Доведя коэффициент ветвления до крайности, $D > n - 1$, мы получим кучу, в которой корень имеет список из $n - 1$ дочерних элементов, поэтому методу `insert` потребуется выполнить только одно сравнение и одну перестановку, а методу `top` — n сравнений и одну перестановку (что ничуть не лучше, чем при работе с несортированным списком элементов).

[1] Где D — коэффициент ветвления анализируемой d-ичной кучи.

Не существует простого[1] способа найти значение D, минимизирующее функцию $f(n) = D \log_D(n)$ для всех значений n, и, кроме того, эта формула дает лишь оценку максимального числа операций доступа и перестановок. Точное количество фактически выполненных сравнений и перестановок зависит от последовательности операций и порядка добавления элементов.

Таблица 2.7. Стоимость основных операций в виде количества сравнений (в куче с n элементами)

Операция	Количество перестановок	Дополнительная память
`Insert`	$\sim\log_D(n)$	O(1)
`Top`	$\sim D*\log_D(n)$	O(1)
`Heapify`	$\sim n$	O(1)

Тогда возникает вопрос: как выбрать оптимальный коэффициент ветвления?

Лучшее, что можно сделать, — выполнить профилирование приложения с разными значениями. Теоретически чем чаще приложение вызывает `insert` по сравнению с `top`, тем больший выигрыш оно получит от увеличения коэффициента ветвления. Когда соотношение вызовов между этими двумя методами приближается к 1,0, лучшим можно считать более сбалансированный выбор.

2.10.1. Добро пожаловать в профилирование

Итак, мы столкнулись с необходимостью *профилирования*. Но что такое профилирование? И с чего начать? Вот несколько советов тем, у кого возникли такие вопросы.

- Под профилированием понимается измерение времени выполнения и, возможно, потребления памяти различными частями кода.
- Профилирование можно выполнить на высоком уровне (измерить время вызовов высокоуровневых функций) или на более низком, вплоть до отдельных инструкций. Измерения можно выполнять вручную (засечь время до и после вызова и вычислить разность), однако существуют отличные инструменты, способные помочь в этом и обычно гарантирующие безошибочность измерений.
- Профилирование не может дать универсальные ответы: оно лишь измеряет производительность *конкретного* кода на *конкретных* данных.
- С другой стороны, это означает, что результат профилирования в точности соответствует предложенным входным данным, то есть в случаях использования очень специфических входных данных можно настроить код так, что он будет показывать высокую производительность в пограничных случаях и низкую на других входных данных. Еще один ключевой фактор — объем данных: для стати-

[1] Очевидно, что можно найти формулу минимума f(D) с помощью матанализа, в частности вычислением производных первого и второго порядка функции.

стической значимости полезно бывает собрать результаты профилирования на множестве прогонов с (псевдо-) случайными входными данными.

- Это также означает, что результаты нельзя обобщать на другие языки программирования или даже на разные реализации одного и того же языка.
- Профилирование требует времени. Чем более глубоко вы занимаетесь настройкой, тем больше времени требуется на профилирование. Помните совет Дональда Кнута: «Преждевременная оптимизация — корень всех зол»? Это означает, что профилирование следует применять только для оптимизации критических путей в коде. Тратить два дня, чтобы сократить время выполнения приложения на 5 мс, если оно занимает 1 минуту, — это, откровенно говоря, пустая потеря времени (и, возможно, если вы в конечном итоге, пытаясь оптимизировать, усложните код, есть риск ухудшить его).

Если, несмотря на все сказанное, вы решите, что приложение действительно нуждается в некоторой настройке, то подготовьтесь и выберите хороший инструмент профилирования для вашего фреймворка.

Очевидно, что при демонстрации приемов профилирования нам придется отказаться от псевдокода и обратиться к реальному коду; в примере ниже профилируется программа кодирования Хаффмана на Python с использованием динамической кучи. Этот материал можно найти в репозитории книги на GitHub[1].

Сам код и тесты написаны на Python 3, в частности с использованием версии 3.7.4 — последней стабильной версии на момент написания этих строк. Для анализа результатов профилирования будем использовать несколько библиотек и инструментов:

- Pandas;
- Matplotlib;
- Jupyter Notebook.

Чтобы облегчить жизнь желающим опробовать код, я предлагаю установить дистрибутив Anaconda, включающий последний дистрибутив Python и все перечисленные пакеты.

Для фактического профилирования используем пакет `cProfile`, который уже включен в базовый дистрибутив Python.

Я не буду подробно объяснять, как использовать `cProfile` (об этом рассказывается во многих онлайн-материалах, начиная с документации Python, ссылка на которую приведена выше), только скажу, что `cProfile` позволяет профилировать определенный метод или функцию и зафиксировать время выполнения, общее для каждого вызова и накопленное.

Используя `pStats.Stats`, можно получить и вывести (или обработать) эти статистики; результат профилирования выглядит так, как показано на рис. 2.18.

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#huffman-compression.

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
      1    0.000    0.000    0.002    0.002 <string>:1(<module>)
      1    0.000    0.000    0.002    0.002 huffman_profile.py:24(run_test)
      1    0.000    0.000    0.002    0.002 huffman.py:116(create_encoding)
      1    0.000    0.000    0.001    0.001 huffman.py:92(_heap_to_tree)
     37    0.000    0.000    0.001    0.000 dway_heap.py:190(top)
     44    0.000    0.000    0.001    0.000 dway_heap.py:71(_push_down)
     95    0.000    0.000    0.001    0.000 dway_heap.py:141(_highest_priority_child_index)
780/452    0.000    0.000    0.000    0.000 {built-in method builtins.len}
      1    0.000    0.000    0.000    0.000 huffman_profile.py:11(read_text)
      1    0.000    0.000    0.000    0.000 huffman.py:72(_frequency_table_to_heap)
   37/1    0.000    0.000    0.000    0.000 huffman.py:49(tree_encoding)
      1    0.000    0.000    0.000    0.000 {built-in method io.open}
      1    0.000    0.000    0.000    0.000 dway_heap.py:18(__init__)
      1    0.000    0.000    0.000    0.000 dway_heap.py:169(_heapify)
    328    0.000    0.000    0.000    0.000 dway_heap.py:44(__len__)
     18    0.000    0.000    0.000    0.000 dway_heap.py:217(insert)
     45    0.000    0.000    0.000    0.000 dway_heap.py:166(first_leaf_index)
      1    0.000    0.000    0.000    0.000 {method 'read' of '_io.TextIOWrapper' objects}
      1    0.000    0.000    0.000    0.000 huffman.py:67(_create_frequency_table)
```

Рис. 2.18. Вывод статистик, накопленных в Stats после профилирования функции кодирования Хаффмана

Чтобы не распыляться, сконцентрируемся на нескольких методах, а именно на функциях, использующих кучу:

- `_frequency_table_to_heap`, которая принимает словарь с частотами (количествами вхождений) символов во входном тексте и создает кучу с одним элементом для каждого символа;
- `_heap_to_tree`, которая принимает кучу, созданную предыдущей функцией, и на ее основе создает дерево кодирования Хаффмана.

Также отследим вызовы методов кучи: `_heapify`, `top` и `insert`, но не просто выведем статистики, а прочитаем их как словарь и отфильтруем записи, соответствующие этим пяти функциям.

Чтобы получить содержательные и надежные результаты, необходимо также профилировать несколько вызовов метода `huffman.create_encoding`, поэтому решение обработать статистику и сохранить результаты в CSV-файл кажется удачным.

Чтобы увидеть профилирование в действии, загрузите пример[1] из репозитория GitHub. В примере показано несколько вызовов метода, создающего код Хаффмана

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction/blob/master/Python/mlarocca/tests/huffman_profile.py.

для группы больших текстовых файлов[1] и растрового изображения. Растровое изображение должно быть предварительно преобразовано, чтобы его можно было обрабатывать как текст. Детали этого предварительного преобразования не особенно интересны, но для любознательных отметим, что байты изображения кодируются в формат base64, чтобы их можно было представить допустимыми символами.

2.10.2. Интерпретация результатов

Сохранив результаты профилирования в CSV-файл, остается только их интерпретировать и понять, какой коэффициент ветвления лучше всего подходит для приложения кодирования Хаффмана. Кажется, в этом нет ничего сложного, верно?

Интерпретировать результаты можно разными способами, но лично я предпочитаю нарисовать несколько графиков в Jupyter Notebook: посмотрите, как это делается в примере[2]. Разве не замечательно, что GitHub уже позволяет отображать блокноты Jupyter без необходимости запускать эту среду локально?

Однако, прежде чем углубиться в результаты, сделаем шаг назад. Поскольку для отображения мы собираемся использовать коробчатые диаграммы (boxplot)[3], необходимо уметь интерпретировать эти графики. Тем из вас, кто не обладает этим умением, поможет рис. 2.19!

Для анализа данных в нашем примере блокнота используется библиотека *Pandas*, с помощью которой CSV-файл со статистиками читается и преобразуется в `DataFrame` — внутреннее представление наборов данных в Pandas. Это представление можно рассматривать как таблицу SQL «на стероидах». На самом деле, используя `DataFrame`, можно разделить данные по контрольным примерам (изображение или текст), сгруппировать их по методам и, наконец, обработать каждый метод отдельно и отобразить результаты для каждого метода, сгруппированные по фактору ветвления. На рис. 2.20 показаны результаты для случая кодирования большого текста по двум основным функциям, использующим кучу.

Эти результаты подтверждают некоторые заявления, сделанные в разделе 2.9:

- коэффициент ветвления 2 никогда не является лучшим выбором;
- оптимальным компромиссом выглядит коэффициент ветвления 4 или 5;
- неизменно наблюдается разница (а для `_frequency_table_to_heap` она составляет целых –50 %) между производительностью двоичной и *d*-ичной кучи с оптимальным коэффициентом ветвления.

---

[1] Мы использовали романы, свободные от авторских прав, загруженные с сайта Project Gutenberg, http://www.gutenberg.org (обязательно загляните туда!).

[2] Имейте в виду, что есть и другие, наверное более удачные, способы сделать то же самое. Это лишь один из возможных. https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction/blob/master/Python/mlarocca/notebooks/Huffman_profiling.ipynb.

[3] https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.boxplot.html.

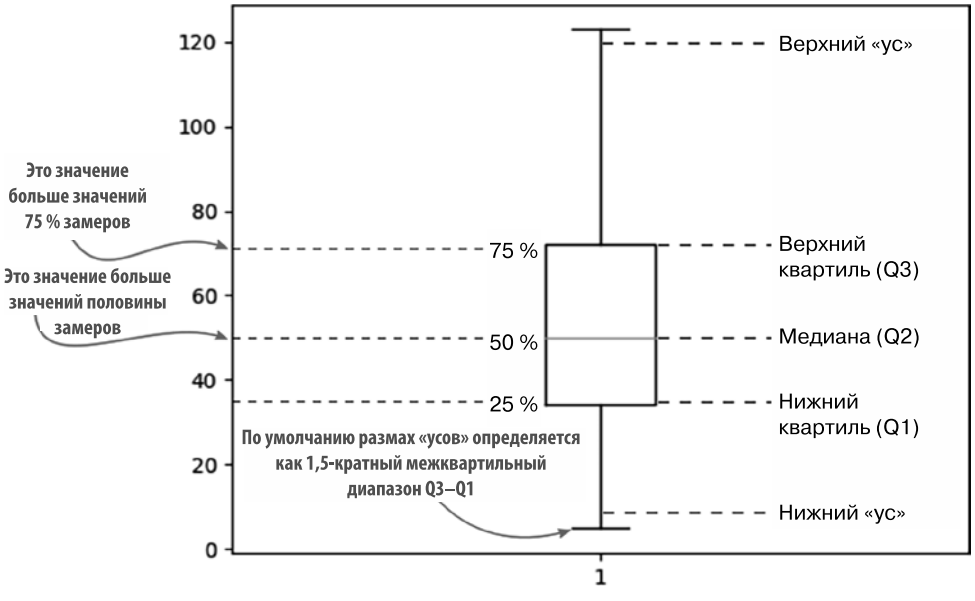


Рис. 2.19. Как читать коробчатые диаграммы. Коробчатые диаграммы в красивом и информативном виде показывают распределение замеров. Основная идея заключается в том, что наиболее релевантными являются значения, лежащие между первым и третьим квартилями. То есть мы находим три значения, Q1, Q2 (также называемое медианой) и Q3, такие, что 25 % замеров меньше Q1; 50 % замеров меньше Q2 (кстати, это само определение медианы); и 75 % замеров меньше Q3. Прямоугольники на диаграмме четко отражают значения Q1 и Q3, а «усы» показывают, насколько широко от медианы разбросаны данные. С этой целью можно также отобразить выбросы, иными словами, замеры, находящиеся за пределами диапазона «усов». Однако иногда выбросы больше запутывают, чем помогают, поэтому мы не будем рассматривать их в этом примере

Однако кое-что может показаться удивительным: похоже, что производительность `_frequency_table_to_heap` растет с увеличением коэффициента ветвления, тогда как `_heap_to_tree` показывает наивысшую производительность при коэффициенте ветвления около 9 и 10.

Как объяснялось в разделе 2.6 на примере псевдокода методов кучи (и как можно видеть в примерах кода на GitHub), первая функция вызывает только вспомогательный метод `_push_down`, а вторая использует `top` (который, в свою очередь, вызывает `_push_down`) и `insert` (вызывающий `_bubble_up`), поэтому ожидался противоположный результат. Во всяком случае, `_heap_to_tree` вызывает `top` и `insert` в соотношении 2:1.

Теперь углубимся внутрь этих высокоуровневых функций и посмотрим, как меняется время выполнения методов кучи `_heapify` (рис. 2.22), `top` и `insert` и их вспомогательных методов `_push_down` и `_bubble_up` (рис. 2.21 и 2.23).

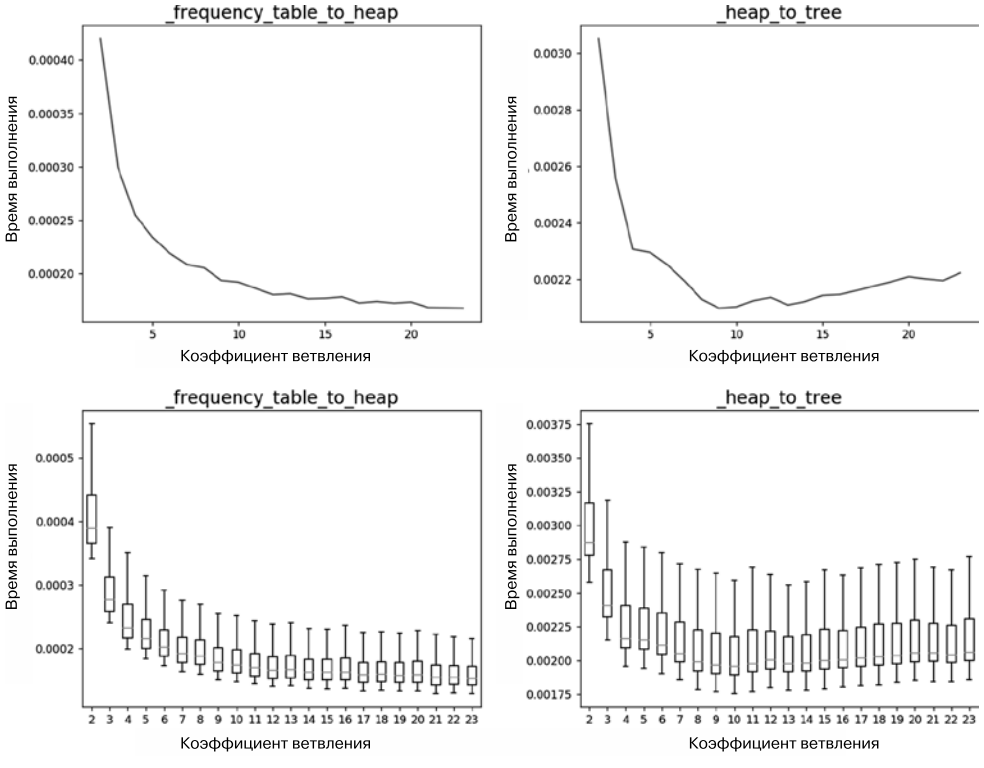


Рис. 2.20. Распределение времени выполнения двух высокоуровневых функций в Huffman.py с реализацией алгоритма Хаффмана на основе куч. *Вверху*: среднее время выполнения в зависимости от коэффициента ветвления. *Внизу*: коробчатые диаграммы распределения времени выполнения в зависимости от коэффициента ветвления. Все диаграммы созданы с использованием результатов хронометража единственного вызова для сжатия группы текстовых файлов

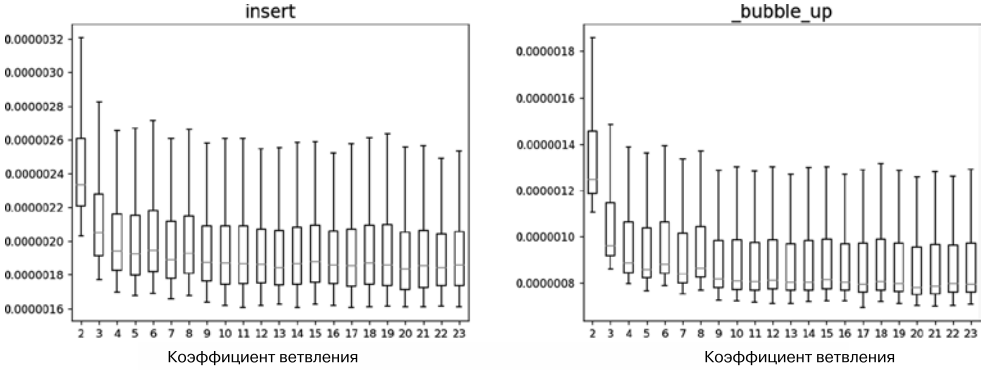


Рис. 2.21. Распределение времени выполнения методов insert и _bubble_up

Прежде чем углубиться в самое интересное, пройдемся по результатам хронометража метода `insert`. На рис. 2.21 видно, что этот метод не таит никаких сюрпризов; его производительность постепенно улучшается с увеличением коэффициента ветвления, как и производительность метода `_bubble_up` — его главного вспомогательного метода.

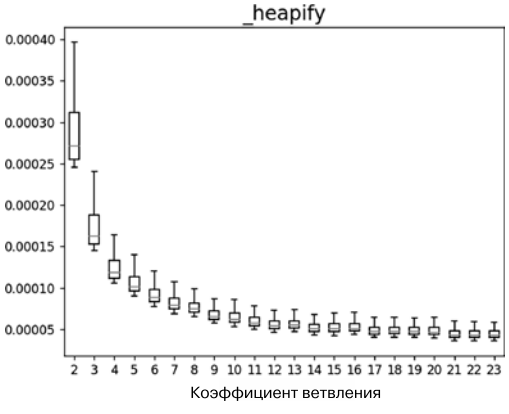


Рис. 2.22. Распределение времени выполнения метода heapify

Метод `_heapify`, как и ожидалось, показывает тенденцию, аналогичную функции `_frequency_table_to_heap`, потому что он просто создает кучу из таблицы частот. И все же кажется немного удивительным, что производительность `_heapify` не ухудшается при больших коэффициентах ветвления.

А теперь самое интересное. Взгляните, как меняется производительность метода `top` (рис. 2.23). На графике ясно виден локальный минимум (около $D == 9$), что вполне ожидаемо, учитывая время выполнения метода $O(D \log_D(n))$. Полученные эмпирические результаты подтверждают теоретические рассуждения и выкладки, приведенные в табл. 2.7. Неудивительно, что идентичное распределение имеет и вспомогательный метод `_push_down`.

Если внимательно рассмотреть листинги 2.7 и 2.4, а также вспомнить, о чем рассказывалось в подразделах 2.6.2 и 2.6.4, то станет ясно, что `pushDown` является основой метода `top`, а основная работа в `pushDown` — это метод, который на каждом уровне кучи извлекает дочерний элемент текущего узла. Мы дали ему имя `highestPriorityChild`[1].

График времени выполнения `_highest_priority_child` (рис. 2.24, *слева*) подтверждает, что профилирование действительно не лишено смысла, так как выявлено, что оно увеличивается с коэффициентом ветвления. Чем больше D, тем длиннее для каждого узла список дочерних элементов, которые этот метод должен обойти, чтобы выбрать следующую ветвь дерева для обхода.

[1] В нашей реализации на Python используются имена _push_down и _highest_priority_child соответственно, как принято соглашениями об именах в языке Python.

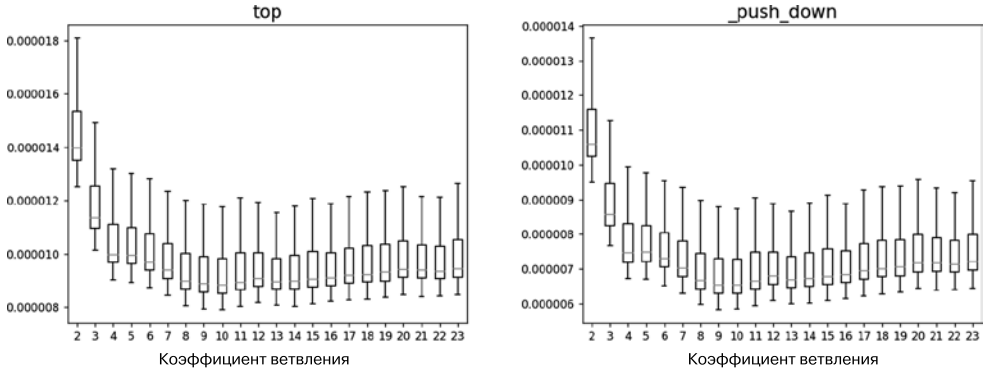


Рис. 2.23. Распределение времени выполнения методов top и _push_down

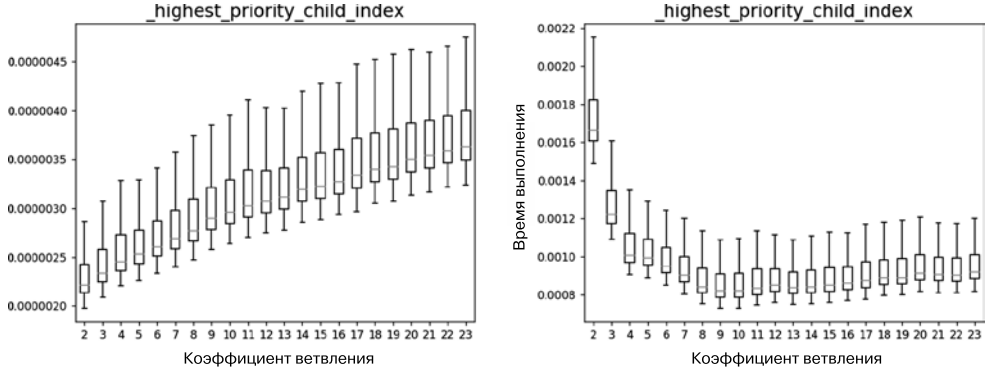


Рис. 2.24. *Слева:* распределение времени выполнения _highest_priority_child.
Справа: распределение накопленного времени выполнения _highest_priority_child

Возникает вопрос, почему `_push_down` не проявляет такой же тенденции? Да, время выполнения `_highest_priority_child` равно O(D), в частности, он выполняет D сравнений, но `_push_down` выполняет (не более) $\log_D(n)$ перестановок и $D \log_D(n)$ сравнений, потому что вызывает `_highest_priority_child` не более $\log_D(n)$ раз.

Чем больше коэффициент ветвления D, тем меньше обращений к `_highest_priority_child`. Это становится очевидным, если вместо графика времени выполнения `_highest_priority_child` построить график накопленного времени (сумму всех вызовов в каждом методе), как показано на рис. 2.24, *справа*. Здесь тоже заметно, что эта составная функция $f(D) = D \log_D(n)$ имеет минимум при $D == 9$.

2.10.3. Загадка _heapify

Как было показано выше, производительность `_heapify` продолжает улучшаться даже при больших коэффициентах ветвления, хотя можно также сказать, что она выходит на плато после $D == 13$, но такое поведение противоречит поведению методов `top` и `_push_down`, соответствующему ожиданиям.

Рост производительности `_heapify` можно объяснить несколькими причинами.

1. Возможно, опробовав бóльшие коэффициенты ветвления, мы обнаружим минимум (просто мы его еще не нашли).
2. Производительность этого метода сильно зависит от порядка вставки: с отсортированными данными он показывает лучшую производительность, чем со случайными.
3. Особенности реализации делают менее значимым вклад вызовов `_push_down` в общее время выполнения.

Но... можно ли утверждать, что наблюдаемые результаты противоречат теории? Вы уже знаете, что я люблю риторические вопросы, а это означает, что пришло время заняться математикой.

И действительно, давайте вспомним подраздел 2.6.7, в котором отмечалось, что количество перестановок, выполняемых методом `_heapify`, ограничено величиной:

$$\frac{n}{D}\sum_{h=0}^{\lfloor \log_D(n) \rfloor}\left\lceil \frac{h}{D^h} \right\rceil.$$

Анализ этой функции — непростая задача. Но, самое интересное, теперь можно построить функцию, используя свои (возможно, недавно приобретенные) навыки работы с Jupyter Notebook. На рис. 2.25 вы видите построенные графики зависимостей этой функции от *D* для нескольких значений *n*. Они показывают, что, несмотря на замедление выполнения `_push_down` с увеличением коэффициента ветвления, суммарное количество перестановок ожидаемо уменьшится.

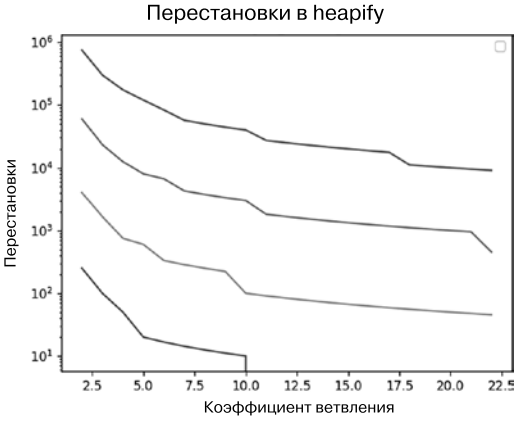


Рис. 2.25. Верхняя граница количества перестановок, выполняемых методом heapify, как функция коэффициента ветвления D. Линии представляют разные значения n. Обратите внимание, что ось Y использует логарифмическую шкалу

Разгадка найдена; метод `_heapify` ведет себя в точности так, как ожидалось. И самое приятное, что его производительность увеличивается с ростом коэффициента ветвления.

2.10.4. Выбор лучшего коэффициента ветвления

Перед тем как поставить точку, нужно ответить на оставшийся важный вопрос: какой коэффициент ветвления выбрать, чтобы добиться от метода кодирования Хаффмана максимальной производительности? Мы отвлеклись, углубившись в анализ кучи, и забыли про этот самый главный вопрос.

Ответить на него можно только одним способом: посмотреть на график времени выполнения основного метода библиотеки кодирования Хаффмана на рис. 2.26.

На нем можно наблюдать три интересных факта:

- лучшим выбором кажется $D == 9$;
- выбор любого значения больше 7, вероятно, будет ничем не хуже;
- в отличие от `_frequency_table_to_heap` и `_heapify`, для которых максимальный прирост составляет 50 и 80 % соответственно, здесь мы получаем только 5 %.

Обратите внимание, что график изменения производительности для `create_encoding` больше похож на график метода `_heap_to_tree`, чем на `_frequency_table_to_heap`. Кроме того, учитывая третий пункт, напрашивается объяснение, что операции с кучей вносят лишь малую часть в общее время выполнения и, соответственно, методы с бóльшим вкладом в него нужно искать в другом месте. (Подсказка: время выполнения `create_frequency_table` зависит от длины входного файла, а других методов — только от размера алфавита.)

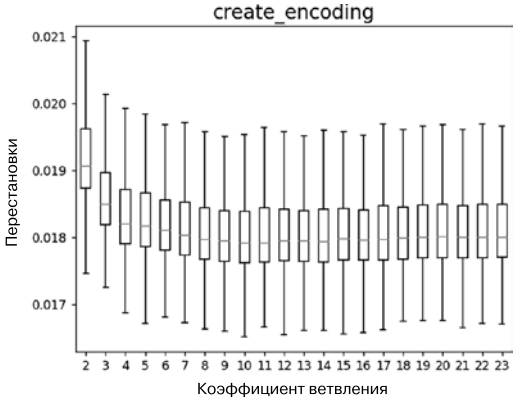


Рис. 2.26. Распределение времени выполнения метода huffman.create_encoding

Полные результаты, включая другие примеры, можно найти в блокноте Jupyter на GitHub. Но имейте в виду, что в этом анализе используются ограниченные данные, поэтому он может быть необъективным. Это всего лишь пример, и я настоятельно рекомендую попробовать выполнить более тщательное профилирование, используя больше прогонов для разных входных данных.

Там же, на GitHub, можно найти полный код профилирования и визуализации, дающий почву для углубленного анализа.

В завершение хотелось бы сделать несколько выводов:

- *D*-ичные кучи быстрее двоичных;
- если есть код, зависящий от параметра, то профилирование поможет настроить его, а лучший выбор будет зависеть от реализации и от входных данных;
- старайтесь выполнять профилирование на репрезентативной выборке данных: при использовании ограниченного подмножества вы, скорее всего, выполните оптимизацию для пограничного случая, и полученные вами результаты будут необъективными.

Оптимизируйте правильно: профилируйте только критический код и сначала выполняйте высокоуровневое профилирование, чтобы выявить такие критические участки кода, оптимизация которых даст наибольшие выгоды. Убедитесь, что ожидаемое улучшение будет стоить потраченного времени.

РЕЗЮМЕ

- Теория, лежащая в основе функционирования и анализа *d*-ичной кучи, основана на базовых структурах и инструментах, описываемых в приложениях от A до E.
- Существует разница между конкретными и абстрактными структурами данных. Последние лучше использовать в виде черных ящиков при разработке приложений или алгоритмов.
- Переходя от абстрактных структур данных к их конкретным реализациям на каком-то языке программирования, следует обратить внимание на мелкие детали, характерные для выбранного языка, и убедиться, что методы не замедляются из-за неправильной реализации.
- Концептуально куча — это дерево, но для повышения эффективности она реализована с использованием массива.
- Изменение коэффициента ветвления кучи не влияет на асимптотическое время выполнения ее методов, но обеспечивает плавное улучшение производительности при переходе от чистой теории к приложениям, управляющим огромными объемами данных.
- В некоторых продвинутых алгоритмах используются очереди с приоритетами, помогающие увеличить производительность. В их числе можно назвать BFS, алгоритм Дейкстры и алгоритм Хаффмана.
- Когда высокая производительность имеет первостепенное значение, а используемые структуры данных позволяют параметризовать их (например, изменять коэффициент ветвления для куч), единственный способ найти наилучшие параметры для реализации — выполнить профилирование кода.

3 Декартовы деревья: применение рандомизации для получения сбалансированных двоичных деревьев поиска

В этой главе

- ✓ Индексация данных по нескольким критериям.
- ✓ Структура данных в декартовом дереве.
- ✓ Балансировка двоичного дерева поиска.
- ✓ Применение декартовых деревьев для реализации сбалансированных двоичных деревьев поиска (BST).
- ✓ Работа с рандомизированными декартовыми деревьями (RT).
- ✓ Сравнение простых сбалансированных двоичных и декартовых деревьев.

В главе 2 вы видели, как хранить элементы и извлекать их с учетом приоритетов, используя кучи, и как можно усовершенствовать двоичные кучи, увеличив коэффициент ветвления.

Очереди с приоритетом особенно полезны, когда элементы должны извлекаться из динамически изменяющегося списка (например, списка задач, выполняемых на процессоре) в определенном порядке, чтобы в любой момент можно было получить следующий элемент (в соответствии с определенными критериями), удалить его из списка и (обычно) перестать беспокоиться об исправлении чего-либо в других элементах. В отличие от сортированного списка элементы в приоритетной очереди

просматриваются только один раз, а элементы, удаленные из списка, не имеют значения для упорядочения.

Когда требуется обеспечивать определенный порядок следования элементов и, возможно, выполнять обход их перечня несколько раз (например, списка объектов для отображения на веб-странице), приоритетные очереди могут оказаться не лучшим выбором. Более того, есть и другие виды операций, которые могут понадобиться; например эффективное извлечение минимального или максимального элемента коллекции, доступ к *i*-му элементу (без удаления элементов перед ним) или поиск предшествующего или последующего элемента в упорядоченном наборе.

В приложении B рассказывается, что деревья являются лучшим компромиссом, когда возникает потребность в дополнительных операциях, помимо вставки и удаления. Если дерево сбалансировано, то любое из этих действий можно выполнить за логарифмическое время.

Проблема в том, что деревья в целом и двоичные деревья в частности не гарантируют сбалансированности. На рис. 3.1 показано, как в зависимости от порядка вставки могут получаться весьма сбалансированные или сильно перекошенные деревья.

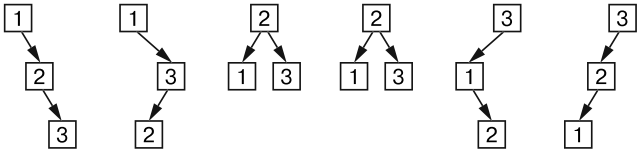


Рис. 3.1. Все возможные варианты двоичного дерева поиска (BST) с тремя элементами. Структура дерева зависит от порядка вставки элементов. Обратите внимание, что две последовательности создают деревья с одинаковой структурой: для [2, 1, 3] и [2, 3, 1] мы получаем один и тот же конечный результат

В этой главе мы рассмотрим приемы использования свойств кучи, гарантирующие (в разумных пределах) сбалансированность двоичных деревьев. Для объяснения механики работы введем понятие декартовых деревьев (англ. treaps) — это структура данных, объединяющая деревья и кучи. Но мы здесь будем использовать иной подход, несколько необычный и непохожий на описываемый в литературе по этой структуре данных.

Для начала, как всегда, начнем с описания задачи, которую будем решать.

3.1. ЗАДАЧА: МНОЖЕСТВЕННАЯ ИНДЕКСАЦИЯ

Ваша семья владеет небольшим продуктовым магазином, и вы хотели бы помочь своим родителям следить за ассортиментом. Чтобы произвести впечатление и показать всем, что уроки информатики стоили затраченных усилий, вы приступаете к разработке цифрового инструмента управления ассортиментом — хранилища со списком имеющихся товаров, к которому предъявляются два требования:

- возможность эффективного поиска товаров по названиям, позволяющая обновлять ассортимент;
- возможность в любой момент отыскать товар, количество которого на складе минимально, что позволит спланировать следующий заказ поставщику.

Конечно, можно просто купить готовую электронную таблицу, но какой в этом смысл? Более того, разве кого-то это впечатлит? Итак, приступим к разработке структуры данных, поддерживающей запросы с двумя различными критериями.

Совершенно понятно, что реальные сценарии намного сложнее. Например, для доставки каждого товара требуется разное время, а некоторые товары заказываются у одних и тех же поставщиков (и, следовательно, их логично свести в один заказ, чтобы сэкономить на доставке). Цена товара со временем может меняться (значит, резонно выбирать самую дешевую марку, скажем, муки или масла на момент заказа), а некоторые товары иногда вообще оказываются недоступны.

Однако все эти сложности и нюансы бизнеса можно учесть и отразить в виде эвристической функции. Концептуально описанный анализ можно проводить точно так же, как простой натуральный учет, поэтому в нашем примере не стоит усложнять, лучше просто использовать описанный подход.

Один из способов удовлетворить перечисленные требования — использовать две разные структуры данных: одну для эффективного поиска по названию, например хеш-таблицу, и другую для поиска товаров, нуждающихся в срочном пополнении, например приоритетную очередь.

В главе 7 я покажу, что иногда объединение двух структур данных является лучшим выбором для достижения цели. Но не будем торопиться; не забывайте о проблемах координации двух таких контейнеров, а также о том, что для их хранения может потребоваться в два раза больше памяти.

Оба эти соображения вызывают беспокойство; было бы неплохо, если бы существовала единая структура данных, эффективно обрабатывающая оба аспекта.

3.1.1. Суть решения

Проясним, что нам нужно: это не просто вопрос оптимизации операций с контейнером, как обсуждается в приложении В. Здесь каждый бит данных, каждая запись в контейнере состоит из двух отдельных частей, и обе можно «измерить» некоторым образом. Есть названия товаров, которые можно сортировать в алфавитном порядке; с каждым товаром связано его количество на складе. Количества можно сравнивать, чтобы определить, какие товары заканчиваются и нуждаются в пополнении.

Теперь если отсортировать список по одному критерию, например по имени, то придется просмотреть весь список, чтобы найти значение, заданное другим критерием, в данном случае количество на складе.

А если использовать неубывающую кучу с товарами в небольшом количестве наверху, то (как рассказывалось в главе 2) нам потребуется линейное время для сканирования всей кучи в поисках товара для пополнения.

Проще говоря, ни одна из базовых структур данных, описанных в приложении B, ни очередь с приоритетами не способны в одиночку решить эту задачу.

3.2. РЕШЕНИЕ: ОПИСАНИЕ И API

Теперь, получив представление о том, что должен делать идеальный контейнер (хотя мы пока не знаем, как он должен это делать), можно определить абстрактную структуру данных (Abstract Data Structure, ADT) с соответствующим API. Пока реализации будут соответствовать этому API, можно будет беспрепятственно использовать любую из них в программе или в более сложном алгоритме, не опасаясь получить сбой (табл. 3.1).

Таблица 3.1. API и контракт для SortedPriorityQueue

Абстрактная структура данных	

API	`class SortedPriorityQueue {`
	`  top() → element`
	`  peek() → element`
	`  insert(element, priority)`
	`  remove(element)`
	`  update(element, newPriority)`
	`  contains(element)`
	`  min()`
	`  max()`
	`}`

Контракт с клиентом	Записи сортируются по элементам (ключам), но в любой момент методы `top()` и `peek()` могут вернуть элемент с наивысшим приоритетом

Исходя из этого описания, можно представить себе расширенный вариант очереди с приоритетами, которая дополнительно поддерживает сортировку элементов. В оставшейся части главы используется термин *«ключ»* для обозначения элементов, и присваивается приоритет каждому элементу.

В сравнении с абстрактной структурой `PriorityQueue`, представленной в главе 2, класс `SortedPriorityQueue` имеет несколько новых методов: метод поиска заданного ключа в контейнере и два метода, возвращающих минимальный и максимальный ключи.

Заглянув в приложение B, вы увидите, что эти три операции часто реализуются базовыми структурами данных, такими как связанные списки, массивы или деревья (некоторые из них выполняются за линейное время, а некоторые, обладающие большей производительностью, — за логарифмическое).

Таким образом, структуру `SortedPriorityQueue` можно рассматривать как сплав двух разных контейнеров, объединяющий характеристики и предоставляющий методы обеих структур данных. Например, ее можно рассматривать как комбинацию кучи и связанного списка или...

3.3. ДЕКАРТОВО ДЕРЕВО

...Дерева и кучи!

Декартово дерево[1] имеет еще одно название, хотя и редко употребляемое, — *дуча*[2], составленное из слов «дерево» и «куча». На самом деле двоичные деревья поиска обеспечивают наилучшую среднюю производительность для всех стандартных операций вставки, удаления и поиска (а также для получения минимального и максимального значений).

Кучи, с другой стороны, эффективно поддерживают приоритеты, используя древовидную структуру. Поскольку двоичные кучи одновременно являются двоичными деревьями, эти две структуры кажутся совместимыми; нужно только найти способ заставить их сосуществовать в одной объединяющей структуре, и тогда можно будет получить все самое лучшее, предлагаемое каждой из них.

Однако легко сказать, но сложно сделать! К одномерному набору данных не получится одновременно применить инварианты двоичного дерева поиска и кучи. Мы можем добавить:

- либо «горизонтальное» ограничение (данный узел `N` с двумя потомками, где `L` — левый и `R` — правый потомок, и тогда все ключи в левом поддереве с корнем в `L` должны быть меньше ключа `N`, а все ключи в правом поддереве с корнем `R` должны быть больше ключа `N`);
- либо «вертикальное» ограничение: ключ в корне любого поддерева должен быть наименьшим в этом поддереве.

В любом случае нам повезло, потому что каждая наша запись имеет два значения: имя и количество. Соответственно, идея состоит в том, чтобы применить ограничения двоичного дерева поиска к именам, а ограничения кучи — к количествам, и получить примерно то, что изображено на рис. 3.2.

В этом примере названия товаров играют роль ключей двоичного дерева поиска, поэтому они определяют общий порядок (на рисунке слева направо).

Количества товаров, в свою очередь, рассматриваются как приоритеты в куче и определяют частичное упорядочение сверху вниз. Для приоритетов, как и для всех куч, действует частичное упорядочение, когда только узлы, находящиеся на одном и том же пути от корня к листьям, упорядочены в соответствии с приоритетами.

---

[1] Декартовы деревья были представлены в докладе Randomized search trees Сесилией Р. Арагон (Cecilia R. Aragon) и Раймундом С. Зейделем (Raimund C. Seidel) на 30-м ежегодном симпозиуме по основам информатики. IEEE, 1989. Название статьи может показаться не связанным с обсуждаемой темой, но далее в этой главе вы увидите, как декартовы деревья связаны с рандомизированными деревьями поиска.

[2] Слово «дуча» образовано из первой буквы в слове «дерево» и последних трех букв в слове «куча». Точно так же сформирован англоязычный термин treap, обозначающий декартово дерево — из первых букв в слове tree («дерево») и последних букв в слове heap («куча»).

На рис. 3.2 можно видеть, что дочерние узлы всегда представлены в большем количестве, чем их родители, но такое отношение не гарантируется между узлами-братьями (находящимися на одном уровне).

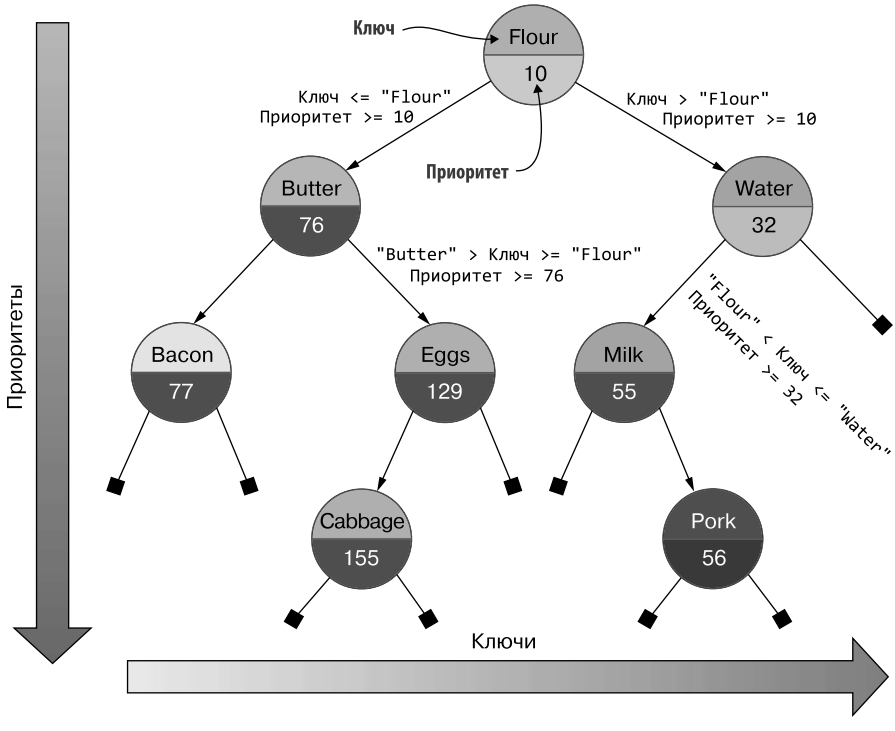


Рис. 3.2. Пример декартова дерева со строками в роли ключей двоичного дерева поиска и целыми числами в роли приоритетов кучи. Обратите внимание, что куча в этом случае является неубывающей, поэтому меньшие числа находятся сверху. Для некоторых стрелок, близких к корню, также показан диапазон ключей, которые могут быть помещены в ветви дерева с корнем в узле, на который они указывают

Этот тип дерева предлагает простой способ поиска записей по ключу (в примере — по названиям товаров). Выполнить поиск по приоритетам в нем сложнее, зато легко найти элемент с наивысшим приоритетом[1]. Он всегда будет в корне дерева!

Однако извлечь верхний элемент... сложнее, чем в случае с кучей! Его нельзя просто заменить листом кучи и сдвинуть вниз, потому что нужно учесть и ограничения двоичного дерева поиска.

[1] Как обсуждалось в главе 2, «более высокий приоритет» — это абстрактное понятие, которое может означать и более низкое, и более высокое значение в зависимости от типа кучи. Здесь мы имеем дело с неубывающей кучей, и более высокий приоритет означает «меньшее количество товара», поэтому меньшие значения находятся выше. Код тоже предполагает, что реализуется неубывающая куча.

Точно так же для вставки (или удаления) узла нельзя использовать простой алгоритм двоичного дерева. Если просто найти позицию, которую будет занимать новый ключ в дереве, и добавить его в качестве листа, как показано на рис. 3.3, то ограничение BST будет соблюдаться, но приоритет нового узла может нарушить инварианты кучи.

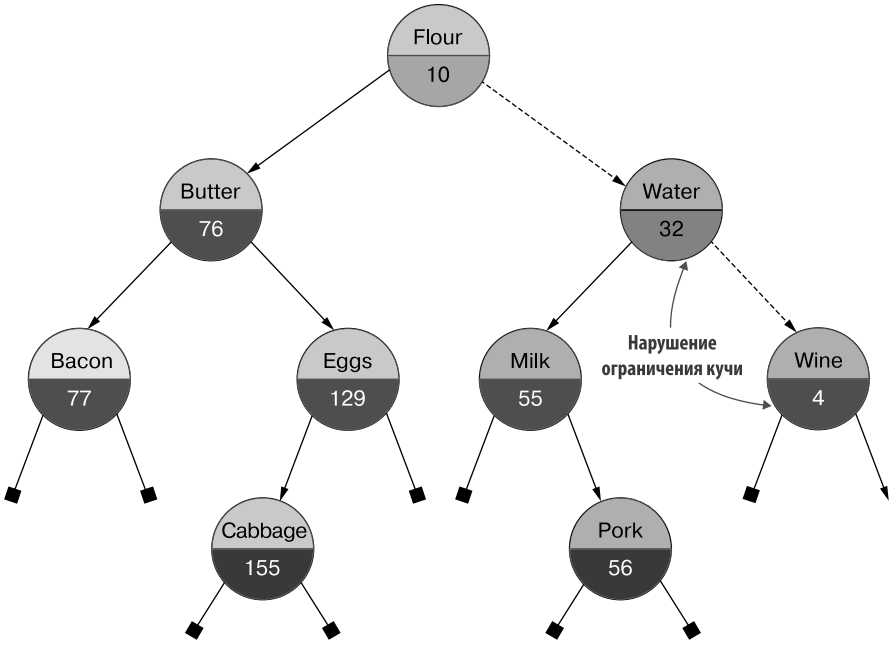


Рис. 3.3. Пример вставки узла в декартово дерево только на основе величины ключа. Однако приоритет нового узла нарушает ограничения кучи

В листинге 3.1 представлена возможная реализация основной структуры декартова дерева. В ней для моделирования узлов дерева используется вспомогательный класс. Обратите внимание, что в отличие от куч, обсуждавшихся в главе 2, здесь применяются явные ссылки на дочерние узлы. Мы еще вернемся к обсуждению этого выбора в подразделе 3.3.2.

В этой реализации класс `Treap` по сути является оберткой вокруг корня фактического дерева; каждый узел дерева содержит два атрибута: `key`, который может быть любого типа, поддерживающего возможность упорядочения, и `priority`, который будем считать числом с плавающей точкой двойной точности. Целочисленные и любые другие типы с возможностью упорядочения тоже можно использовать, но, как будет показано в следующем разделе, числа с плавающей точкой двойной точности лучше подходят на эту роль.

Более того, каждый узел содержит указатели (ссылки) на два дочерних элемента, левый и правый, и на родителя.

Листинг 3.1. Класс Treap

```
class Node
  key
  #type double
  priority

  #type Node
  left
  #type Node
  right
  #type Node
  parent

  function Node(key, priority)
    (this.key, this.priority) ← (key, priority)
    this.left ← null
    this.right ← null
    this.parent ← null

  function setLeft(node)
    this.left ← node
    if node != null then
      node.parent ← this

class Treap
  #type Node
  root

  function Treap()
    root ← null
```

Конструктор класса Node устанавливает ключ и приоритет в соответствии с аргументами и записывает null во все ссылки

Обновляются левый дочерний элемент Node и ссылки на родителя в дочерних элементах. Метод setRight (здесь не показан для экономии места) работает симметрично

Конструктор узла просто инициализирует атрибуты `key` и `priority` значениями аргументов, а указатели `left` и `right` — значением `null`, фактически создавая лист. Затем, после создания экземпляра узла, в указатели на ветви могут быть записаны действительные ссылки. Как вариант, можно реализовать перегруженную версию конструктора, принимающую ссылки на дочерние элементы.

3.3.1. Поворот

Есть ли решение у этой проблемы? Есть. И таким решением является одна из операций над двоичными деревьями поиска: операция поворота. На рис. 3.4 показано, как поворот может исправить нарушение (или, наоборот, привести к нарушению!) ограничений декартова дерева. Поворот — обычная операция для многих версий двоичных деревьев поиска, таких как красно-черные деревья или деревья 2-3[1].

Поворот двоичного дерева поиска — это преобразование, инвертирующее отношения «родитель — потомок» между узлами дерева `Y` и `X` (см. рис. 3.4). Требуется, чтобы

[1] Красно-черные деревья и деревья 2-3 — это несколько необычные версии сбалансированных двоичных деревьев поиска. Мы еще вернемся к ним далее в этой главе.

дочерний узел стал родительским и наоборот, но мы не можем просто поменять местами эти два узла: иначе в общем случае, когда ключи двух узлов различны, в итоге нарушится порядок ключей.

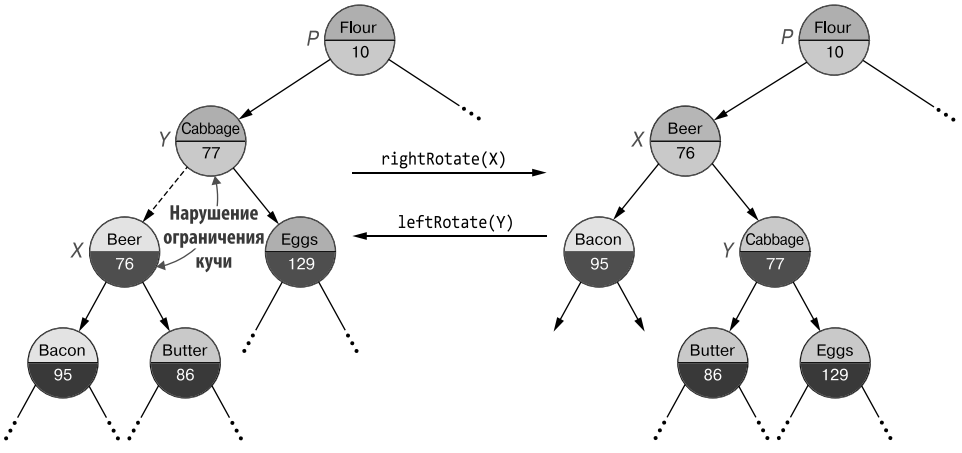


Рис. 3.4. Повороты влево и вправо на примере. Декартово дерево слева нарушает инварианты кучи, но поворот вправо относительно узла, отмеченного как X, исправляет это нарушение. И наоборот, если применить поворот влево к куче справа относительно узла, отмеченного как Y, то результат нарушит ограничения кучи. Обратите внимание, что поворот вправо всегда применяется к левым дочерним элементам, а влево — к правым

Вместо этого следует удалить все поддерево с корнем в родительском элементе, заменить его (меньшим) поддеревом с корнем в дочернем элементе, а затем вставить удаленные узлы в это новое поддерево.

Как это сделать? Как показано на рис. 3.4, сначала следует различать два случая, в зависимости от того, где находится дочерний узел по отношению к родителю — слева или справа. Эти положения симметричны, следовательно, далее сосредоточимся только на первом случае.

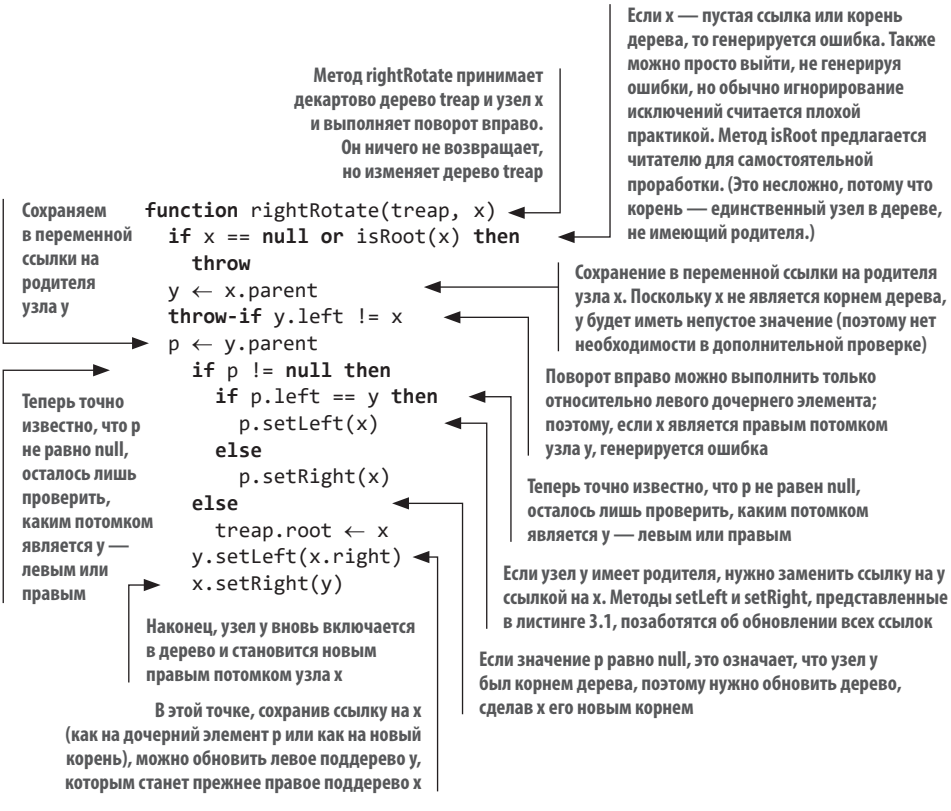
В листингах 3.2 и 3.3 показан псевдокод поворотов вправо и влево, поясняющий детали операций, описанных несколькими строками выше. На рис. 3.5 изображены шаги, выполняемые при повороте вправо, где дочерний узел `X`, точка поворота, является левым дочерним элементом своего родителя `Y`.

Нужно удалить `Y` из дерева, обновить `P` — родителя узла `Y` (строки 4–11), заменив ссылку на `Y` ссылкой на `X` (строки 8–11); в этот момент `Y` исключается из дерева, а вместе с ним и все его правое поддерево.

Левое поддерево узла `Y` опустело, потому что узел `X` был отсоединен и перемещен. Теперь можно переместить правое поддерево узла `X` в левый дочерний элемент узла `Y` (строка 14), как показано внизу слева на рис. 3.5. Это не нарушает упорядоченности ключей, потому что (при условии, что *перед* поворотом нарушений не было)

key[Y]>=key[Y.left] и key[Y]>=key[Y.left.right]. Другими словами, поскольку X был левым потомком узла Y, правое поддерево узла X по-прежнему находится в левом поддереве узла Y и все ключи в левом поддереве узла меньше или в лучшем случае равны собственному ключу узла. Для справки можно использовать рис. 3.2.

Листинг 3.2. Поворот вправо



Осталось подключить Y к основному дереву: этот узел можно вставить на место правого потомка узла X (строка 15), что не создаст никаких нарушений. На самом деле мы уже знаем, что Y и его правое поддерево имеют ключи, которые больше, чем ключ X, а левое поддерево Y было построено с использованием прежнего правого поддерева X и по определению все ключи в нем тоже больше, чем ключ X.

Теперь вы знаете, как выполнить поворот. В этом нет ничего необычного, нужно просто обновить несколько ссылок в дереве. Единственная загадка на данный момент: почему такую операцию называют поворотом?

Ответ на этот вопрос можно найти на рис. 3.6, интерпретирующем шаги, которые мы видели в листинге 3.2 и на рис. 3.5, но рассматривали с другой точки зрения. Хочу отметить, что это всего лишь неформальный способ показать, как работает

поворот. Но если вы соберетесь реализовать метод сами, то лучше будет обратиться к листингу 3.2 и рис. 3.5.

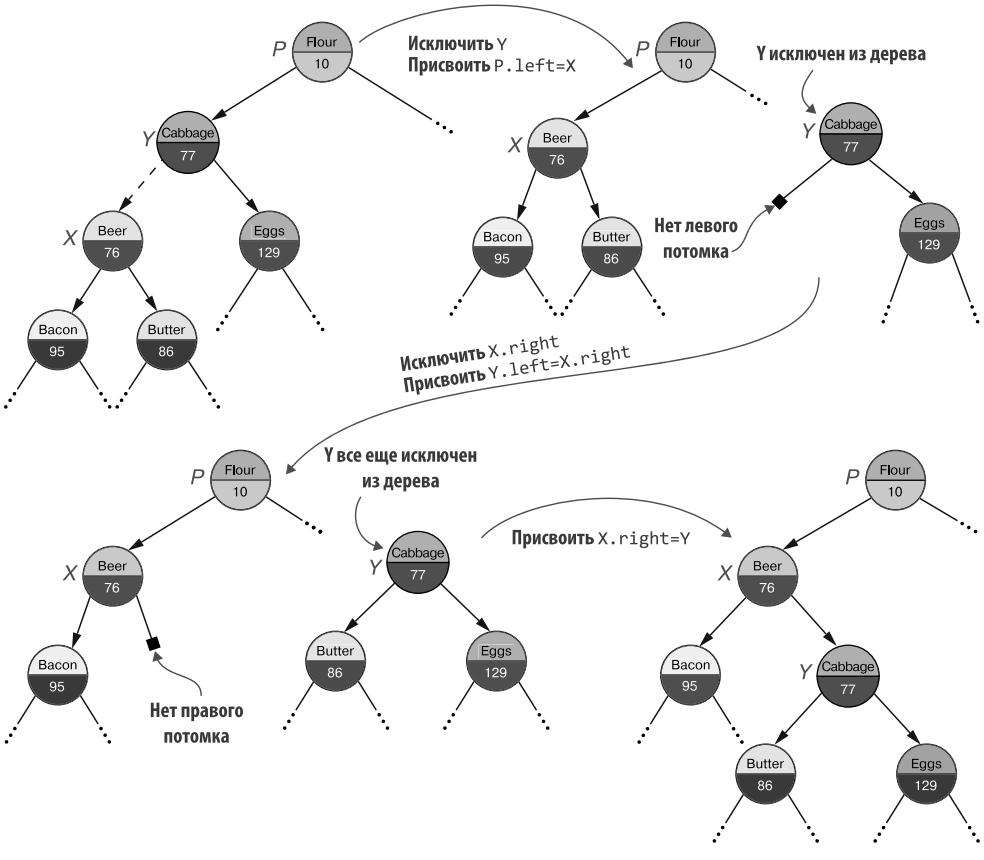


Рис. 3.5. Алгоритм правого поворота двоичного дерева поиска

Листинг 3.3. Поворот влево

```
function lefttRotate(treap, x)
  if x == null or isRoot(x) then
    throw
  y ← x.parent

  throw-if y.right != x
  p ← y.parent
```

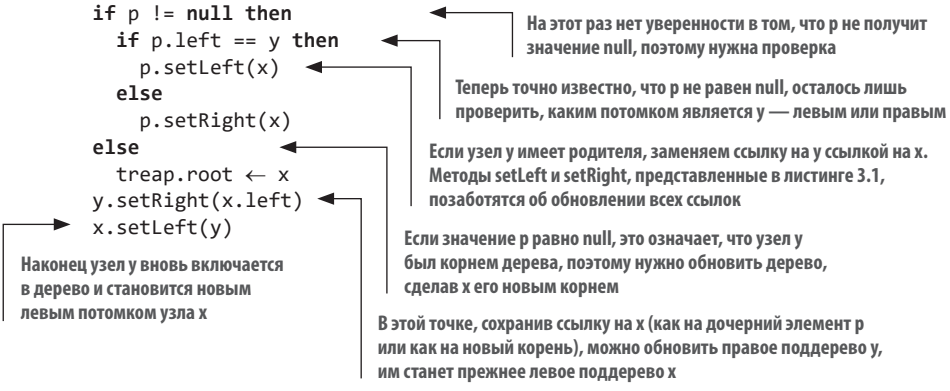
Метод leftRotate принимает декартово дерево treap и узел x и выполняет поворот влево. Он ничего не возвращает, но изменяет дерево treap

Сохраняем в переменной ссылки на родителя узла y

Если x — пустая ссылка или корень дерева, то генерируется ошибка

Сохранение в переменной ссылки на родителя узла x. Поскольку x не является корнем дерева, y будет иметь непустое значение

Поворот влево можно выполнить только относительно правого дочернего элемента; поэтому, если x является левым потомком узла y, генерируется ошибка



Предположим, что функция поворота вызывается для узла X, а узел Y является его родителем. Напомню, что мы анализируем поворот вправо, соответственно, X является левым потомком Y.

Рассматривая поддерево с корнем в Y, можно визуально «повернуть» его по часовой стрелке (отсюда и название «поворот вправо») относительно оси, проходящей через узел X, пока X не займет положение корня дерева, а все остальные узлы окажутся ниже X, как показано на рис. 3.6.

Результат должен выглядеть так, как показано в правой верхней четверти рис. 3.6. Конечно, чтобы получилось действительное двоичное дерево поиска с корнем в узле X, нужно внести несколько изменений. Например, имеется ребро, направленное от дочернего элемента к родителю — от Y к X, а это запрещено в деревьях, поэтому нужно изменить направление ребра. Однако если просто поменять направление ребра, то у узла X появятся три дочерних элемента, что недопустимо в двоичном дереве. Для решения проблемы можно перенести связь между X и его правым дочерним элементом в Y. Оба изменения показаны внизу слева на рис. 3.6.

На этом этапе поддерево имеет допустимую структуру. Но для полноты преобразования в качестве последнего шага просто улучшим его визуальное представление.

Древовидную структуру можно представить как некую конструкцию из болтов и веревок, и тогда операцию поворота можно описать как захват этой конструкции за узел X и предоставление возможности остальным узлам повиснуть под действием силы тяжести, с единственной оговоркой, что правый потомок X перемещается в узел Y.

Прежде чем завершить обсуждение операций поворота, хотелось бы отметить, что любой поворот всегда сохраняет ограничения двоичного дерева поиска, но не сохраняет инварианты кучи. На самом деле повороты можно использовать для исправления нарушений в дереве, но при применении к правильному дереву они нарушат ограничения приоритета для узла, относительно которого выполняется поворот.

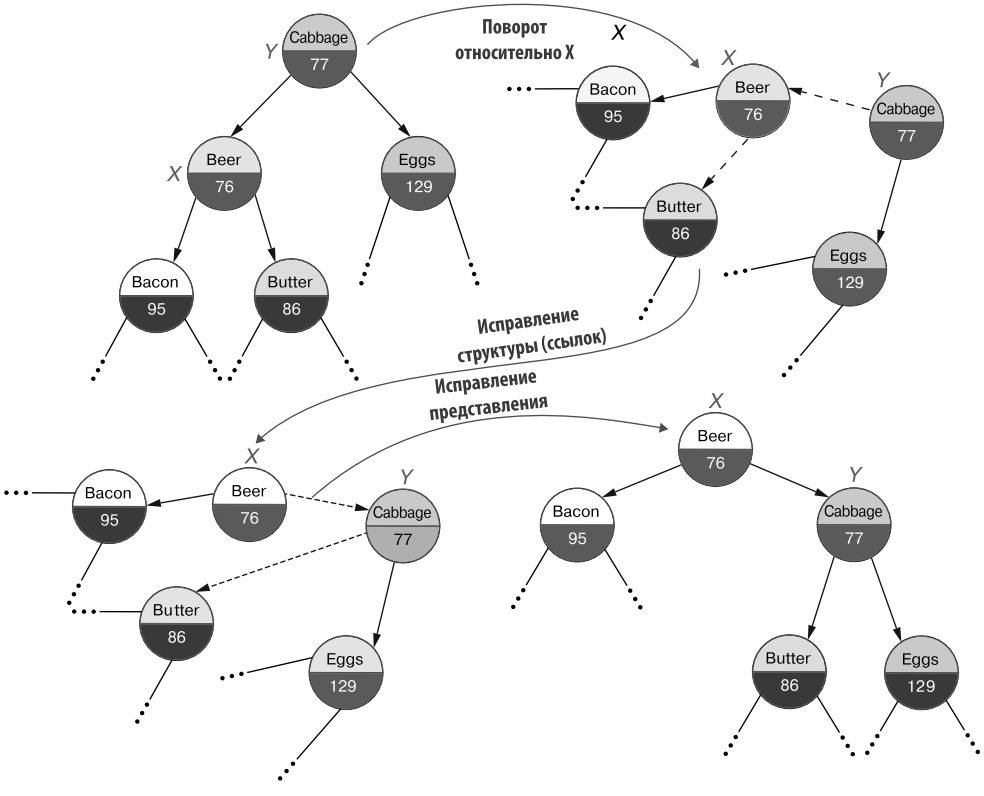


Рис. 3.6. Более наглядная интерпретация операции rightRotate, помогающая объяснить такое название, а также запомнить выполняемые шаги

3.3.2. Несколько вопросов по организации

Декартовы деревья — это кучи, имеющие представление в виде двойного массива и одновременно являющиеся специальными деревьями. Как было показано в главе 2, кучу можно реализовать на основе массива — более эффективного по размеру представления, обладающего также преимуществом локальности ссылок.

Можно ли таким же способом, на основе массива, реализовать декартово дерево? Приостановитесь на минутку и поразмышляйте над этим вопросом, прежде чем двинуться дальше и прочитать ответ. Какие плюсы и минусы дает использование массива по сравнению с деревом и какие неудобства может принести использование дерева?

Проблема представления в виде массива в том, что оно не особенно гибкое. Такое представление хорошо подходит для случаев, когда местами меняются только случайные элементы, а удаление/добавление производится лишь в конец массива. В случаях, когда требуется перемещать элементы, использование массива может привести к катастрофе! Например, вставка нового элемента в середину массива потребует перемещения всех элементов после него, в среднем $O(n)$ перестановок (рис. 3.7).

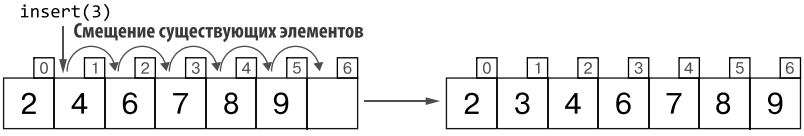


Рис. 3.7. Чтобы вставить новый элемент в упорядоченный массив, нужно все элементы, которые больше нового, сместить на одну позицию в сторону хвоста массива (при условии, что в нем есть место). Это означает, что если захотеть в массив с n элементами вставить новое значение в элемент с индексом k (то есть это будет (k + 1)-й элемент в массиве), то для завершения вставки потребуется выполнить n – k присваиваний

Ключевая особенность куч в том, что они являются полными, сбалансированными и выровненными по левому краю деревьями. А так как кучи не сохраняют общего порядка ключей, можно добавлять и удалять элементы из хвоста массива, затем поднимать вверх или опускать вниз только один элемент, чтобы восстановить свойства кучи (как рассказывалось в главе 2).

Декартовы деревья, напротив, являются двоичными деревьями поиска, сохраняющими порядок ключей. Вот почему нужно выполнять операции поворота при вставке или удалении новых элементов из дерева. Как отмечалось в подразделе 3.2.1, поворот подразумевает перемещение целого поддерева из правого поддерева узла `X` в левое поддерево его родителя `Y` (или наоборот). Нетрудно представить, что эта операция, легко выполняемая за постоянное время при использовании указателей в узлах дерева, может превратиться в мучительно неэффективную при использовании массивов (например, потребовать линейного времени).

Именно поэтому массивы не используются для представления декартовых деревьев (и двоичных деревьев поиска).

Еще один вопрос, который можно и *нужно* задать перед тем, как приступить к реализации: выбор коэффициента ветвления для кучи. В главе 2 я продемонстрировал, что кучи могут иметь коэффициент ветвления, отличный от 2, а в разделе 2.10 даже показал, что куча с коэффициентом ветвления 4 или выше заметно превосходит по производительности двоичную кучу (по крайней мере, в нашем примере приложения). Можно ли точно таким же образом реализовать декартово дерево с коэффициентом ветвления больше 2?

К сожалению, это не так просто. Во-первых, мы используем двоичные деревья поиска, то есть деревья с коэффициентом ветвления 2: если коэффициент ветвления в куче не будет совпадать с коэффициентом ветвления в двоичном дереве поиска, наступит хаос!

Во-вторых, можно, конечно, предложить использовать тернарные деревья поиска или их обобщение, но это сделает операции поворота намного более сложными, а код запутанным (и, вероятно, более медленным!). Кроме того, поддерживать баланс такого дерева будет труднее, если не использовать чего-то вроде дерева 2-3, но это уже изначально гарантированно сбалансированное дерево.

3.3.3. Реализация поиска

Теперь, имея более полное понимание, как декартово дерево должно быть представлено в памяти и как работают повороты, можно переходить к реализации основных методов. Реализацию декартова дерева на Java можно также найти в репозитории книги на GitHub[1].

Начнем с самого простого — метода поиска. Это самый обычный метод поиска, широко используемый в двоичных деревьях поиска: он просматривает дерево, начиная с корня, пока найдет искомый ключ или дойдет до листа, не найдя ничего.

Так же как и в случае с обычными двоичными деревьями поиска, просматривается только одна ветвь каждого поддерева в направлении слева направо в зависимости от способа сравнения искомого ключа с ключом текущего узла.

В листинге 3.4 показана реализация внутреннего метода. Он принимает узел и обходит его поддерево. В этой версии используется рекурсия (прием, описанный в приложении Д). Следует отметить еще раз, что рекурсия часто позволяет писать более понятный и ясный код, обрабатывающий итеративные структуры данных, такие как деревья, однако при значительной глубине рекурсии рекурсивные методы могут вызвать переполнение стека. В этом конкретном случае компиляторы некоторых языков программирования смогут применить оптимизацию хвостовых вызовов и преобразовать рекурсию в явный цикл, одновременно переводя код на машинный язык[2]. Однако в общем случае стоит подумать об использовании явного цикла даже в реализации на языке высокого уровня, особенно если нет уверенности в поддержке компилятором оптимизации хвостовой рекурсии или условий, в которых она может применяться.

Листинг 3.4. Метод search

```
function search(node, targetKey)
  if node == null then
    return null
  if node.key == targetKey then
    return node
  elsif targetKey < node.key then
    return search(node.left, targetKey)
  else
    return search(node.right, targetKey)
```

Метод search принимает узел декартова дерева и искомый ключ. Возвращает узел, хранящий этот ключ, если тот будет найден, или null в противном случае

Если node равен null, то возвращаем null, чтобы показать, что ключ не найден

Если ключ узла совпадает с искомым, то просто возвращаем найденный узел

Проверяем, как соотносятся искомый ключ и ключ текущего узла

Если искомый ключ меньше, то переходим к левой ветви

Иначе искомый ключ, если он имеется, может находиться только в правой ветви, и нужно перейти к ней

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#treap.

[2] Подробнее о проблеме переполнения стека и оптимизации хвостовых вызовов можно прочитать в приложении Д.

Метод `contains` класса `Treap` просто вызывает метод `search`, передавая ему корень дерева, и возвращает `false` или `true` в зависимости от полученного результата — `null` или нет.

3.3.4. Вставка

Поиск ключа в декартовом дереве реализуется относительно просто, но вставка нового элемента — это совсем другая история. Как упоминалось в подразделе 3.3.1, функция вставки для двоичного дерева поиска здесь не подходит, потому что, даже если ключ нового элемента станет на свое место в дереве, его приоритет может нарушить инварианты кучи: например, возникнет ситуация, что он окажется больше приоритета родителя (рис. 3.8).

Но нет причин отчаиваться! У нас есть способ исправления нарушений инвариантов кучи: поворот относительно узла, нарушающего ограничения приоритета.

В общем случае метод вставки `insert` выполняет всего два шага: вставляет новый узел как лист, а затем сравнивает его приоритет с приоритетом родителя. Если приоритет нового узла окажется выше, то новый узел нужно поднять вверх, но мы не можем просто поменять его местами с родителем, как в куче.

Как показано на рис. 3.6, нужно взять поддерево с корнем в родителе нового узла и повернуть его так, чтобы новый узел стал корнем этого поддерева (потому что этот узел определенно будет иметь наивысший приоритет).

В листинге 3.5 показан псевдокод метода вставки, а на рис. 3.8 и 3.9 — пример вставки в инвентарный перечень склада 20 единиц нового элемента Beer («пиво»).

Прежде всего нужно найти правильное место для вставки нового элемента в существующий инвентарный перечень. Для этого производится обход дерева точно так же, как при поиске, только на каждом шаге запоминается родитель текущего узла, чтобы иметь возможность добавить новый лист. Обратите внимание, что обход реализован с явным использованием цикла вместо рекурсии, чтобы было лучше видно, как работает этот метод.

Как показано в верхней части рис. 3.8, первый шаг — это обход дерева и поиск места для добавления нового узла в качестве листа. Мы идем налево, когда пересекаем Flour и Butter, затем направо, на Bacon (строки 5–10 в листинге 3.5).

Обратите также внимание, что для краткости на рисунке используется сокращенная нотация. Вновь добавленный узел, соответствующий переменной `newNode` в листинге 3.5, обозначен на рисунках как `X`, а его родитель — как `P`.

В тот момент, когда завершается цикл `while`, временная переменная `parent` указывает на узел с ключом Bacon; следовательно, условия в строках 11 и 14 не выполняются и новый узел добавляется как правый потомок `parent`, это показано в нижней половине рис. 3.8.

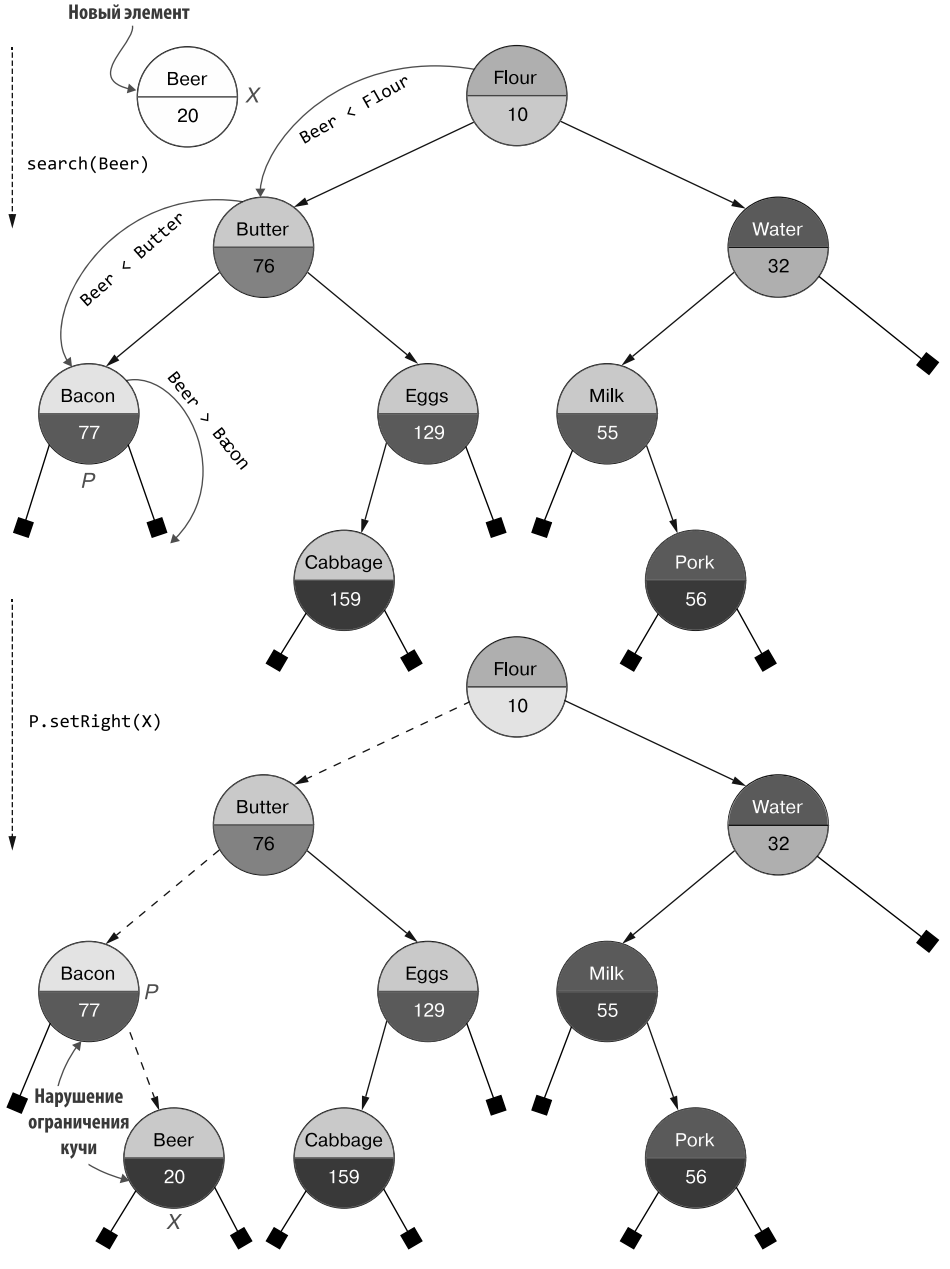
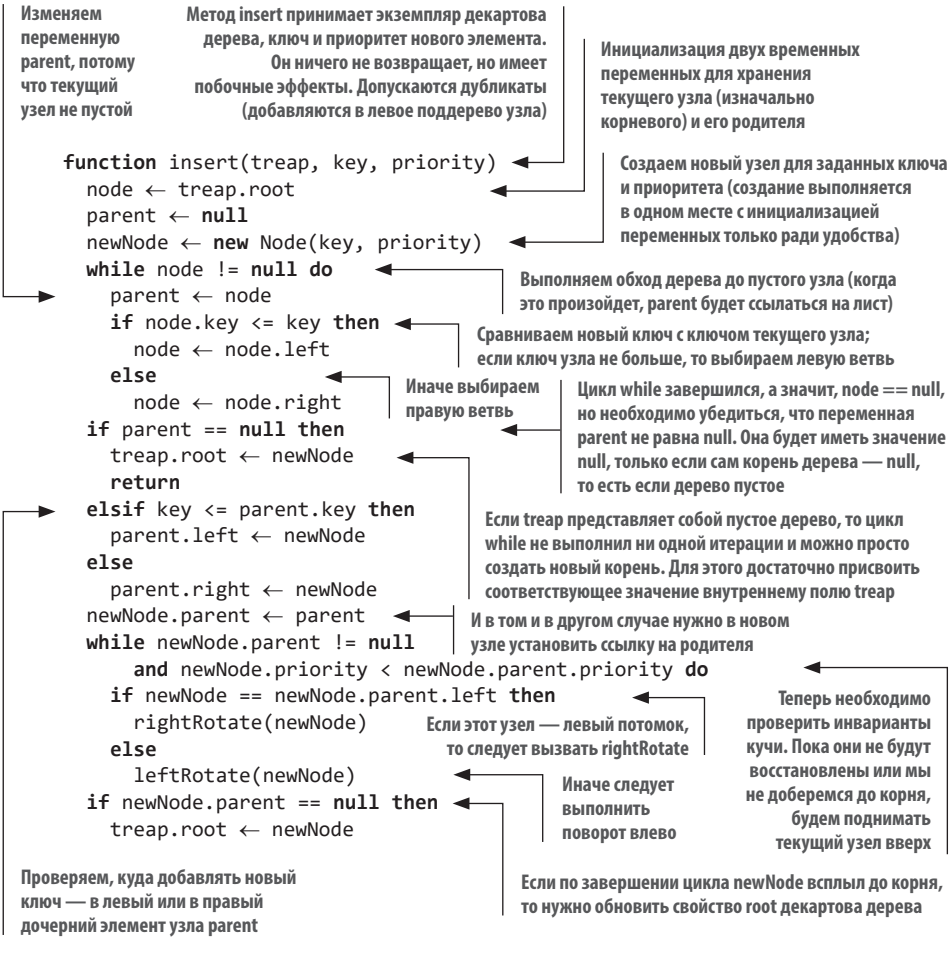


Рис. 3.8. Первые шаги при вставке нового элемента. Выполняется поиск ключа нового элемента, чтобы определить правильное место для вставки нового листа. Затем нужно проверить, не нарушает ли новый узел инварианты кучи. К сожалению, в этом примере так и происходит, поэтому необходимо исправить ситуацию, выполнив поворот влево (показан на рис. 3.9)

Листинг 3.5. Метод Node::insert



Глядя на приведенный пример, можно также заметить, что новый узел имеет более высокий приоритет (меньшее количество единиц на складе), чем его родитель; из-за чего в строке 19 происходит вход в цикл и выполняется поворот дерева влево. После первой итерации и поворота влево узел Beer по-прежнему имеет более высокий приоритет, чем его новый родитель Butter, как показано в верхней части рис. 3.9. Поэтому выполняется вторая итерация цикла, но на этот раз производится поворот вправо, потому что узел `X` теперь является левым потомком узла `P`.

Теперь (нижняя половина рис. 3.9) не нарушается ни один инвариант и можно выйти из цикла. Но так как новый узел не был поднят вверх, то проверка в строке 24 завершается неудачей и больше ничего делать не нужно.

Сколько времени потребуется для выполнения вставки? Новый лист добавится за время $O(h)$, потому что для этого нужно выполнить проход по дереву от корня к листу. Затем потребуется поднять новый узел не выше чем до корня и при каждом повороте

перемещать узел на один уровень вверх, то есть всего потребуется не больше h поворотов. Каждый поворот требует обновления постоянного числа указателей, поэтому для подъема нового узла и всего метода в конечном итоге потребуется $O(h)$ шагов.

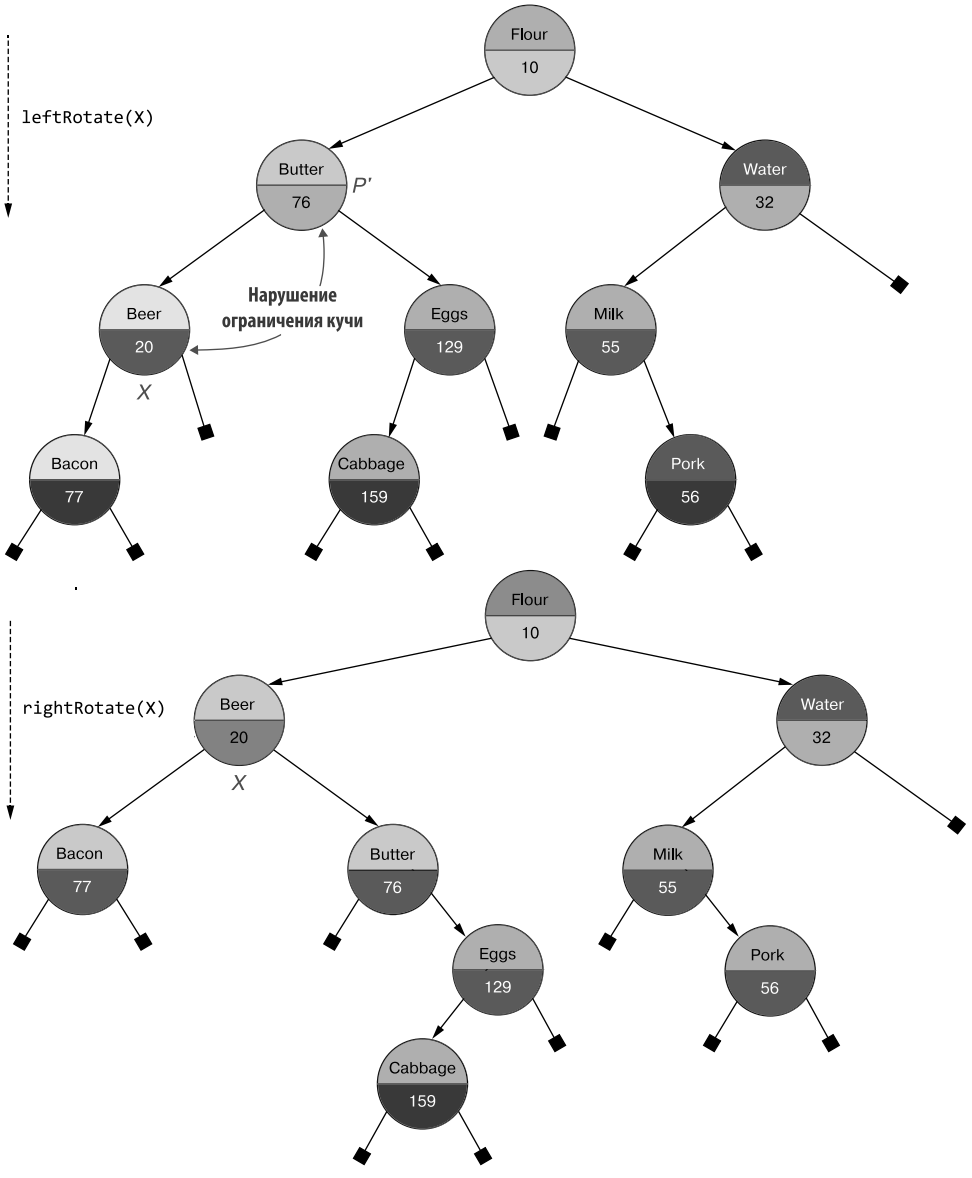


Рис. 3.9. Вставка (продолжение). Чтобы исправить инварианты кучи, нужно выполнить правый поворот декартова дерева, изображенного в конце рис. 3.8. Это действие поднимает новый узел вверх на один уровень, результат показан в верхней части этого рисунка. Инварианты кучи по-прежнему нарушены, поэтому нужно выполнить еще один поворот (на этот раз влево)

3.3.5. Удаление

Удаление ключа из декартова дерева — концептуально более простая операция, но она требует применения совершенно другого подхода по сравнению с двоичными деревьями поиска. В двоичных деревьях поиска удаленный узел заменяется последующим (или предшествующим) элементом, а для декартовых деревьев это решение не годится, потому что приоритет замещающего узла может оказаться меньше приоритетов его новых дочерних элементов, и, если такое случится, узел нужно сместить вниз. Более того, в общем случае последующий узел в дереве двоичного поиска не является листом, поэтому его необходимо рекурсивно удалить.

Более простое решение — последовательно перемещать удаляемый узел вниз, пока он не достигнет листа. Когда он окажется в позиции листа, его можно отсоединить от дерева безо всяких последствий.

Концептуально это похоже на присваивание удаляемому узлу самого низкого возможного приоритета и исправление инвариантов кучи путем опускания этого узла. Операция не остановится, пока узел с бесконечным (отрицательным) приоритетом не достигнет листа. Ее выполнение показано на рис. 3.10, а реализация — в листинге 3.6.

Листинг 3.6. Метод Treap::remove

```
function remove(treap, key)
  node ← search(treap.root, key)
  if node == null then
    return false
  if isRoot(node) and isLeaf(node) then
    treap.root ← null
    return true
  while not isLeaf(node) do
    if node.left != null
        and (node.right==null
          or node.left.priority > node.right.priority) then
      rotateRight(node.left)
    else
      rotateLeft(node.right)
    if isRoot(node.parent) then
      treap.root ← node.parent
  if node.parent.left == node then
    node.parent.left ← null
  else
    node.parent.right ← null
  return true
```

Метод remove принимает экземпляр декартова дерева и удаляемый ключ и возвращает true, если ключ был удален, или false, если ключ не удалось найти. Он также имеет побочные эффекты: изменяет дерево treap, переданное в аргументе

Поиск ключа в дереве

Если search вернул null, значит, искомый ключ не найден и его нельзя удалить

Если treap имеет только один узел, то после его удаления останется пустое дерево. В этом случае размер дерева будет равен 1 или, что то же самое, node будет одновременно и листом, и корнем

Проверяем, какой из двух потомков узла должен заменить его. Узел будет иметь по крайней мере одного потомка (потому что это не лист), и если у него есть оба потомка, то нужно выбрать из них имеющего самый высокий приоритет (то есть в нашем примере — самое низкое значение, потому что мы реализуем неубывающее декартово дерево)

Иначе нужно опустить этот узел до уровня листьев

Если выбран левый потомок, выполняем поворот вправо

Иначе — влево

Если узел был корнем, то нужно обновить свойство root дерева treap. Это условие может выполниться только в первой итерации цикла, поэтому, если производительность имеет решающее значение, будет разумно разбить цикл и обрабатывать первую итерацию отдельно

После выхода из цикла while узел node достиг позиции листа, и его можно исключить из дерева, сбросив указатель на него в его родителе

Вернуть true, потому что узел был удален

В листинге 3.6 особенно хорошо видно, почему было полезно вернуть узел с искомым ключом из метода `search`. Благодаря этому мы смогли повторно использовать этот узел в реализации метода `remove`, который на первом шаге ищет удаляемый ключ, а затем управляет найденным узлом, который нужно переместить вниз.

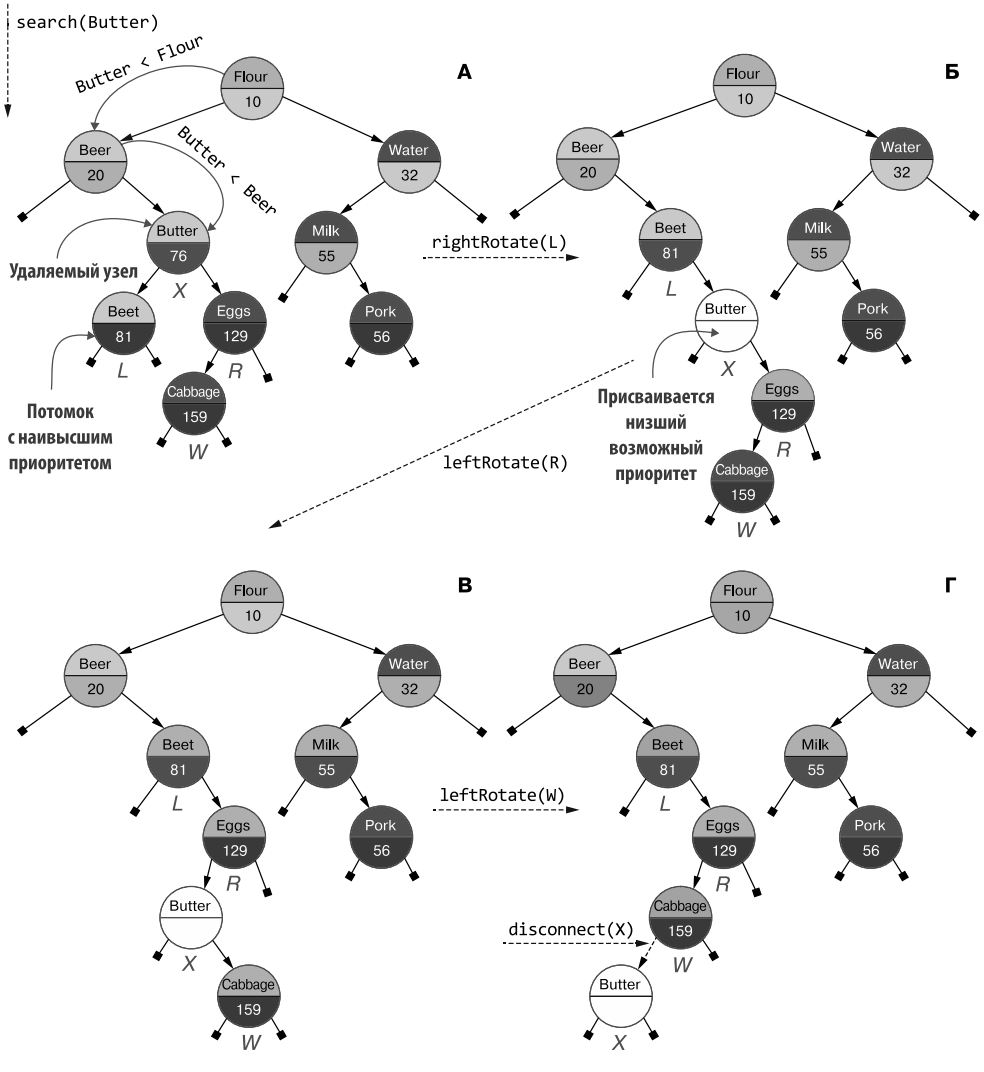


Рис. 3.10. Удаление ключа Butter из примера декартова дерева. Первый шаг — поиск узла, содержащего удаляемый ключ. Затем ему присваивается самый низкий возможный приоритет, после чего он перемещается до уровня листа. Для этого нужно выяснить, какой из дочерних элементов имеет наивысший приоритет, и выполнить поворот относительно него. Наконец, как только узел достигнет уровня листа, его можно просто исключить из дерева, не нарушив никаких инвариантов

Особого внимания, как всегда, заслуживает удаление корня.

Рассмотрим пример, в котором будет применяться алгоритм удаления. Предположим, что необходимо удалить товар Butter («масло») из инвентарного перечня (например, потому, что он больше не будет продаваться или продан весь).

Первый шаг, показанный на рис. 3.10, *А*, —поиск ключа `"Butter"` в дереве (строка 2 в листинге 3.6). После обнаружения узла (очевидно, что он не пустой, строка 3), как обычно отмеченного символом `X` на рисунках, алгоритм убеждается, что это не корень и не лист одновременно (следовательно, проверка в строке 5 вернет `false`) и можно войти в цикл `while` в строке 8.

В строке 9 выбирается левый потомок `X`, обозначенный на рисунке буквой `L`, как потомок с наивысшим приоритетом. Поэтому далее выполняется поворот вправо (строка 10), в результате чего получается дерево, изображенное на рис. 3.10, *Б*.

Здесь, на рисунке, мы изменили приоритет опускаемого узла на $+\infty$[1], но делать это в коде нет необходимости; можно просто продолжать опускать узел, не проверяя его приоритет, пока он не станет листом.

В данный момент `X` еще не лист, хотя у него есть только один потомок (тоже бывший) — правый потомок `R`; поэтому алгоритм переходит к следующей итерации цикла `while` и на этот раз выполняет поворот влево, создавая дерево, изображенное на рис. 3.10, *В*. Затем еще один поворот влево, и `X` наконец становится листом.

В этот момент алгоритм выходит из цикла `while`, а мы, находясь в строке 15, можем быть уверены, что `node` не является корнем (иначе это было бы определено еще в строке 5), а значит, у него есть родитель. Теперь нужно исключить узел из дерева, удалив указатель в его родителе, для чего следует проверить, каким потомком он является — левым или правым.

После того как правой ссылке присвоено значение `null`, алгоритм завершается, результат его работы — успешно удаленный ключ.

Если сравнить этот метод с простой версией двоичного дерева поиска, то можно отметить одно из его достоинств: отпадает необходимость рекурсивно вызывать `remove` для узла, следующего за удаляемым узлом (или предшествующего ему). Мы просто выполняем одно удаление, хотя с несколькими возможными поворотами. Но есть у метода удаления ключа и свой недостаток: при удалении узла, близкого к корню, удаляемый узел придется сдвинуть вниз на несколько слоев, пока он не достигнет уровня листа.

Другими словами, время работы алгоритма удаления в худшем случае равно $O(h)$, где h — высота дерева. Как следствие, высота дерева становится особенно важным фактором.

---

[1] В нашем примере низший приоритет соответствует максимальному количеству товара на складе, поэтому $+\infty$ — это максимально возможное значение единиц товара на складе.

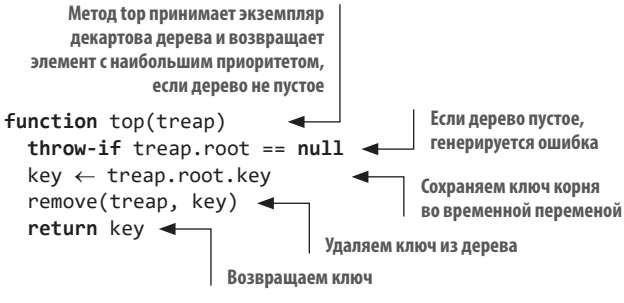
Из примеров понятно, что использование декартова дерева для хранения ключей и приоритетов может привести к несбалансированному дереву, а удаление узла может сделать дерево еще более несбалансированным из-за множества поворотов, начинающихся с уже плохой ситуации.

3.3.6. Методы top, peek и update

Остальные методы класса `Treap` реализовать проще. Метод `peek` такой же, как и в случае с обычными кучами, отличается только способом получения доступа к корню кучи.

Для реализации метода `top`, если вдруг понадобится убедиться, что декартово дерево способно беспрепятственно заменить кучу, можно использовать метод `remove` и написать практически только одну строку кода, как показано в листинге 3.7.

Листинг 3.7. Метод Treap::top



После проверки декартова дерева на наличие узлов остается только получить ключ, хранящийся в корне, а затем удалить его из дерева.

Точно так же, если понадобится изменить приоритет, связанный с ключом, можно последовать той же логике, что и для простых куч, поднимая (при повышении приоритета) или опуская (при понижении) обновленный узел. Единственное отличие в том, что вместо простой перемены узлов местами нужно выполнить последовательность поворотов, чтобы переместить обновленный узел. Реализацию этого метода я оставляю читателям в качестве самостоятельного упражнения (вы можете сравнить свое решение с имеющимся в репозитории книги).

3.3.7. Методы min и max

Итак, осталось реализовать два последних метода: `min` и `max`. Они возвращают минимальный и максимальный ключи, хранящиеся в декартовом дереве.

Эти ключи хранятся соответственно в самом левом и самом правом узлах дерева. Но будьте внимательны, указанные узлы не обязательно должны быть листьями, как показано на рис. 3.11.

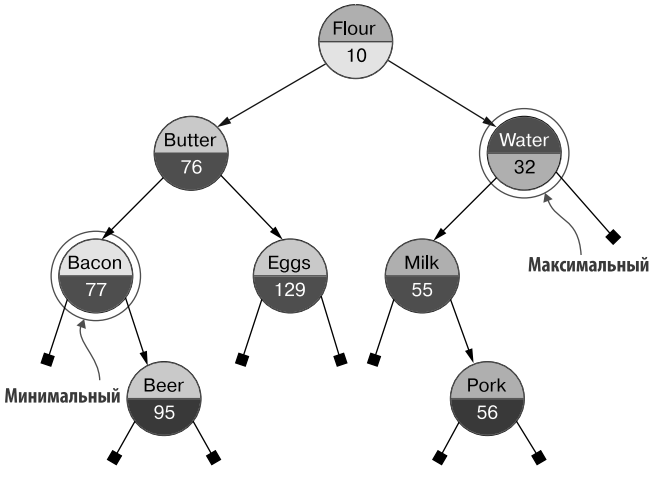


Рис. 3.11. Поиск минимального и максимального ключей в декартовом дереве (и в двоичном дереве поиска): минимальный ключ хранится в самом левом узле, а максимальный — в самом правом. Имейте в виду, что эти узлы не обязательно должны быть листьями

В листинге 3.8 показана возможная реализация метода `min`. Точно так же, как в двоичном дереве поиска, она просто выполняет обход дерева, всегда выбирая левую ветвь, пока не будет достигнут узел, ссылка на левый дочерний элемент в котором равна `null`. Метод `max` имеет симметричную реализацию, достаточно заменить `node.left` на `node.right`.

Листинг 3.8. Метод Treap::min

```
function min(treap)
  throw-if treap.root == null
  node ← treap.root
  while node.left != null do
    node ← node.left
  return node.key
```

Метод min принимает экземпляр декартова дерева и возвращает элемент с наивысшим приоритетом, если только дерево не пустое

Если дерево пустое, генерируется ошибка (так как в нем очевидно нет минимального элемента)

Сохраняем во временной переменной node ссылку на корень дерева (она точно будет непустой, благодаря проверке в строке 2)

Возвращаем ключ узла

Пока не достигнут самый левый узел, продолжаем обход левой ветви

3.3.8. Еще раз о производительности

На этом мы завершим обсуждение реализации декартовых деревьев. В следующих разделах рассмотрим и проанализируем в деталях примеры их практического применения.

А пока подведем итоги и еще раз оценим производительность методов декартова дерева (табл. 3.2). Обратите внимание, что:

- время выполнения всех операций зависит только от высоты дерева, но не от количества элементов. Конечно, в худшем случае $O(h) = O(n)$ для асимметричных деревьев;
- мы не касались здесь вопросов потребления памяти, потому что все эти методы требуют постоянного объема дополнительного пространства.

Таблица 3.2. Операции декартовых деревьев и затраты времени на их выполнение для деревьев с высотой h и количеством ключей n

Операция	Время выполнения	Худший случай
`Insert`	$O(h)$	$O(n)$
`Top`	$O(h)$	$O(n)$
`Remove`	$O(h)$	$O(n)$
`Peek`	$O(1)$	$O(1)$
`Contains`	$O(h)$	$O(n)$
`UpdatePriority`	$O(h)$	$O(n)$
`Min/Max`	$O(h)$	$O(n)$

3.4. ПРИМЕНЕНИЕ: РАНДОМИЗИРОВАННЫЕ ДЕКАРТОВЫ ДЕРЕВЬЯ

Итак, теперь вы можете проводить инвентаризацию товаров на складе и определять, какие из них вскоре закончатся. Это, безусловно, произведет впечатление на всех, кто соберется вечером за семейным столом!

Надеюсь, наш пример помог вам понять, как работают декартовы деревья, но... должен признаться, что на самом деле декартовы деревья не используются для индексации многомерных данных.

В следующих главах, и в частности в главе 7, когда будем говорить о кэше, я покажу, что существуют более удачные способы решения задач, эквивалентных обсуждавшейся в этой главе.

Позвольте пояснить: использование декартовых деревьев одновременно в роли деревьев и куч совершенно законно и даже может дать достойную производительность при определенных условиях, но в общем случае, как мы с вами видели, хранение данных, организованных по обоим критериям, скорее всего, приведет к несбалансированному дереву. То есть операции будут выполняться за линейное время.

Но не для этого были изобретены декартовы деревья, и не для этого они используются в настоящее время. Дело в том, что существуют более удачные способы индексации многомерных данных, а декартовы деревья имеют более удачное применение. Как увидим далее, их можно использовать в качестве строительного блока для реализации другой эффективной структуры данных.

3.4.1. Сбалансированные деревья

Один из аспектов, на который мы с вами обратили внимание: несбалансированные декартовы деревья обычно имеют ветви, длина которых в худшем случае может оказаться равной количеству узлов.

И наоборот, при обсуждении куч было показано, что сбалансированные деревья, как и кучи, имеют логарифмическую высоту, что делает операции с ними довольно производительными.

Однако при работе с кучами загвоздка в том, что преимущества сбалансированных деревьев сводятся на нет из-за ограничения набора операций. Операция поиска ключа в куче не имеет эффективной реализации, так же как и операция поиска максимального или минимального ключей[1] и удаления или обновления произвольного элемента, когда его положение в куче заранее не известно.

Тем не менее в литературе по алгоритмам можно найти много других примеров сбалансированных деревьев — структур данных, гарантирующих, что высота дерева будет логарифмической даже в худшем случае. Некоторые примеры уже упоминались в разделе 3.2, это деревья 2–3[2] (рис. 3.12) и красно-черные деревья[3] (рис. 3.13).

Однако алгоритмы поддержания ограничений этих деревьев, как правило, довольно сложны, причем настолько, что многие учебники по алгоритмам вообще опускают рассмотрение метода удаления.

Но, к счастью, существует возможность использовать декартовы деревья, кажущиеся совершенно несбалансированными, для получения двоичных деревьев поиска, стремящихся к сбалансированности[4]. Причем использовать для этого набор более простых и чистых алгоритмов (по сравнению с красно-черными деревьями и им подобными структурами).

Как рассказывалось во введении к этой главе, простые двоичные деревья поиска имеют ту же проблему, поскольку их структура зависит от порядка вставки элементов.

Если вернуться к последнему разделу, то можно заметить, что деревья могут быть перекошены, если конкретная комбинация ключей и приоритетов, а также порядок вставки элементов особенно неудачны, потому что повороты могут привести к ухудшению балансировки дерева (см. рис. 3.9).

---

1 Напомню, что элементы в куче частично упорядочены по приоритету, но не упорядочены по ключу. Получить элемент с наивысшим приоритетом легко, но, чтобы получить элемент с наименьшим (или наибольшим) ключом, придется проверить все элементы.

2 *Aho A. V., Hopcroft J. E., Ullman J. D.* The design and analysis of computer algorithms. Pearson Education India, 1974. (*Ахо А. В., Хопкрофт Дж. Э., Ульман Дж. Д.* Разработка и анализ компьютерных алгоритмов.)

3 *Guibas L. J., Sedgewick R.* A dichromatic framework for balanced trees // 19th Annual Symposium on Foundations of Computer Science (sfcs 1978). IEEE, 1978.

4 Имеется в виду сбалансированные с высокой степенью вероятности.

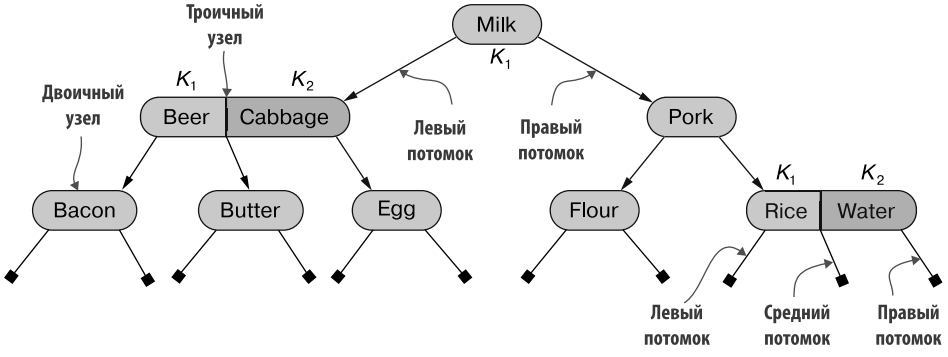


Рис. 3.12. Дерево 2–3 с ключами, которые использовались в примере с продуктовым магазином. Узлы в деревьях 2-3 могут содержать один или два ключа, отсортированных по возрастанию, и соответственно две или три ссылки. Помимо левого и правого потомков, троичные узлы имеют также средний дочерний элемент. Все ключи K в поддереве, на которое указывает средняя ссылка, должны удовлетворять условию: $K_1 > K \geq K_2$, где K_1 и K_2 — первый и второй ключи троичного узла соответственно. Деревья 2-3 гарантированно сбалансированы, благодаря особенностям операции вставки: к листьям добавляются ключи, а когда лист получает третий элемент, он разделяется, и средний элемент всплывает до родительского узла (также возможно рекурсивное разделение). Гарантируется, что высота дерева 2-3 с n ключами находится между $\log_2(n)$ и $\log_3(n)$

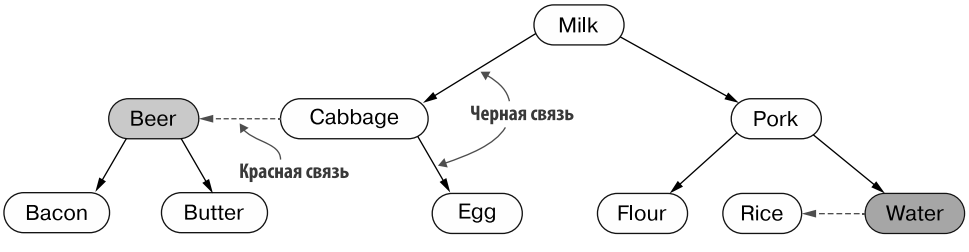


Рис. 3.13. Красно-черное дерево с теми же ключами из нашего примера. Красно-черные деревья — одна из самых простых реализаций деревьев 2–3. Красно-черное двоичное дерево поиска похоже на обычное двоичное дерево поиска, за исключением того, что связи между узлами могут быть двух видов: красными (пунктирные линии) и черными (сплошные линии). Красные связи соединяют ключи, которые в соответствующем дереве 2–3 принадлежали бы одному и тому же троичному узлу. Черные ссылки являются фактическими ссылками дерева 2–3. Есть два ограничения. 1. Ни один узел не имеет двух красных связей (входящих или исходящих); это отражает тот факт, что в дереве 2–3 есть только двоичные и троичные узлы. 2. Все пути от корня к листу имеют одинаковое количество черных связей. Соответственно, узлы могут быть отмечены как красные или как черные. Здесь для ясности использованы красный (затененный) и белый (незатененный) цвета. На любом пути не может быть двух красных узлов, следующих друг за другом. В совокупности эти ограничения гарантируют, что максимально длинный путь в красно-черном двоичном дереве поиска с чередующимися красными и черными связями будет не более чем в два раза длиннее кратчайшего возможного пути в дереве, содержащем только черные ссылки. В свою очередь, это гарантирует логарифмическую высоту дерева. Эти инварианты поддерживаются соответствующим использованием поворотов после вставки и удаления

Рассмотрим идею использования поворотов для балансировки дерева. Если лишить приоритеты смысла (в нашем примере мы забудем о количестве товаров на складе), то теоретически можно обновить значение приоритета каждого узла, так что исправление инвариантов кучи приведет к более сбалансированному дереву.

Рисунок 3.14 иллюстрирует этот процесс. Правая ветвь дерева не сбалансирована, но, обновив узел предпоследнего уровня, можно принудительно повернуть его вправо, что поднимет узел на один уровень и приведет к перебалансировке его поддерева, а в этом простом примере — всего дерева.

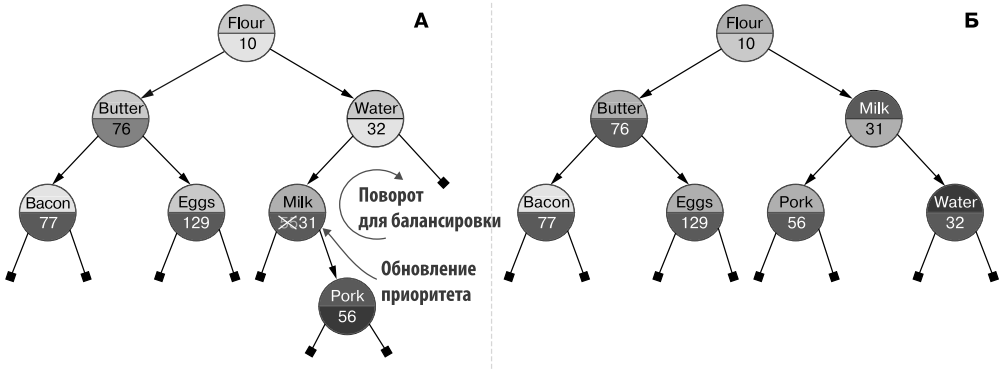


Рис. 3.14. Перебалансировка декартова дерева путем обновления его приоритетов. Если изменить приоритет узла с ключом Milk, присвоив значение меньше, чем у его родителя (но в данном случае больше, чем у корня), то можно исправить инварианты кучи, применив поворот вправо. Кстати, в этом конкретном примере мы получим идеально сбалансированное дерево

Теперь нужно внести ясность: отказываясь от значения поля приоритета, мы реализуем некую новую структуру, отличную от того, что было в разделе 3.3. В частности, она больше не привязана к общедоступному интерфейсу очереди с приоритетами и не предлагает методов `top` и `peek`. Вместо этого получается простое двоичное дерево поиска, внутри которого используются идеи, разработанные для декартовых деревьев с целью поддержания сбалансированной структуры. В табл. 3.3 показаны методы и контракт двоичного дерева поиска. Структура данных, которая будет рассматриваться далее, соответствует этому API.

Таблица 3.3. API и контракт двоичного дерева поиска

Абстрактная структура данных: двоичное дерево поиска (BST)	

API	`class BST {`
	`  insert(element)`
	`  remove(element)`
	`  contains(element)`
	`  min()`
	`  max()`
	`}`

Контракт с клиентом	Записи сортируются по элементам (ключам)

3.4.2. Введение в рандомизацию

Получение лучших результатов с использованием более простых алгоритмов — звучит слишком хорошо, чтобы быть правдой... и отчасти это действительно так — за простоту приходится платить.

Обновление приоритетов для поддержания баланса дерева казалось простым делом в небольшом примере, но делать это систематически на большом дереве слишком сложно и дорого.

Сложно, потому что это похоже на решение головоломки: каждый раз, выполняя поворот относительно внутреннего узла, мы рискуем сделать нижние уровни в поддеревьях, сдвинутых вниз, еще более несбалансированными, и совсем непросто придумать правильный порядок поворотов для получения наилучшей возможной структуры дерева.

Дорого, потому что нужно следить за высотой каждого поддерева и потому что требуется выполнить дополнительную работу для определения последовательности поворотов.

В предыдущем подразделе мы говорили о *стремлении к сбалансированности*, описывая возможный результат. Вероятно, наиболее внимательные читатели уже отметили ключевой момент: здесь имеется в виду рандомизация — введение элемента случайности в структуры данных, с которыми приходится иметь дело.

Случайность будет постоянным фактором в первой части книги. Будут описаны несколько структур данных, использующих ее, включая фильтры Блума. Чтобы помочь вам разобраться в теме, я подготовил краткое введение в рандомизированные алгоритмы в приложении E и предлагаю заглянуть туда, прежде чем углубляться в этот раздел.

В оригинальной работе Арагона и Раймунда декартовы деревья были представлены как средство получения «рандомизированных сбалансированных деревьев поиска». Авторы использовали ту же идею, что была описана в подразделе 3.4.1, — применение приоритетов для создания сбалансированной древовидной структуры, — но обошли сложность настройки вручную, задействовав универсальный генератор случайных чисел для выбора значений приоритетов узлов.

На рис. 3.15 показана более сбалансированная версия дерева, полученного в конце на рис. 3.9. Сбалансированность достигается с помощью замены приоритетов случайно сгенерированными действительными числами. Для приоритетов можно также использовать случайные целые числа, но действительные числа снижают вероятность совпадений и улучшают конечный результат.

В разделе 3.5 я покажу, что если приоритеты взяты из равномерного распределения, то ожидаемая высота дерева будет пропорциональна логарифму от количества узлов.

Самое приятное, что почти весь код, необходимый для реализации этой новой структуры данных, уже написан. Внутри можно использовать декартово дерево (листинг 3.9), а все методы этой новой структуры будут лишь обертками вокруг

методов декартова дерева, за исключением `insert`. Это единственный метод, для которого придется написать дополнительную строку кода.

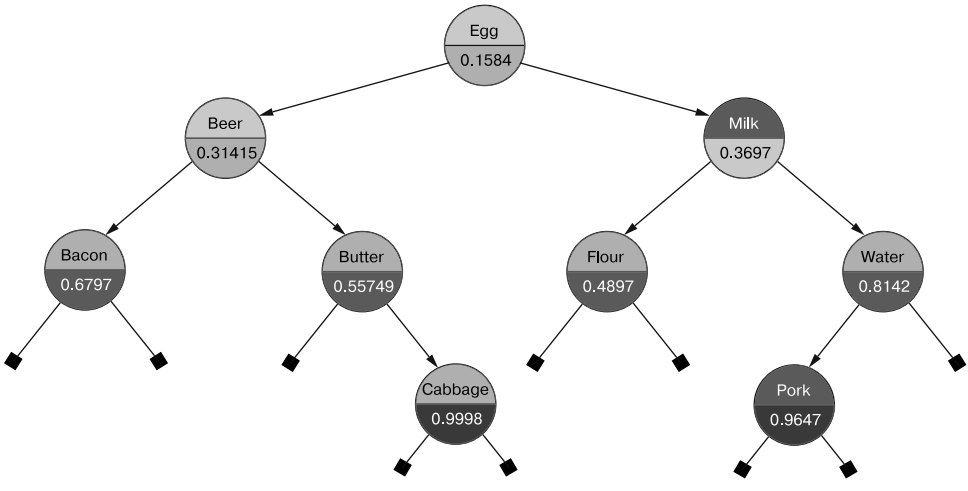


Рис. 3.15. Пример рандомизированного декартова дерева (Randomized Treap, RT) с теми же ключами, что и в дереве на рис. 3.9 (после вставки ключа Beer). Приоритеты — случайные числа в диапазоне от 0 до 1. Это лишь одна из возможных структур ключей, соответствующая одному случайному набору приоритетов

Листинг 3.9. Класс RandomizedTreap

```
class RandomizedTreap
  #type Treap
  treap

  #type RandomNumberGenerator
  randomGenerator

  function RandomizedTreap()
    this.treap ← new Treap
```

Как показано в листинге 3.10, нужно лишь сгенерировать случайный приоритет при вставке нового ключа в рандомизированное декартово дерево.

Реализацию RT на Java можно найти в репозитории книги на GitHub[1].

Листинг 3.10. Метод RandomizedTreap::insert

```
function insert(key)
  return this.treap.insert(key, this.randomGenerator.next())
```

Метод insert принимает ключ для вставки в дерево. Он ничего не возвращает, но имеет побочные эффекты

Вставка нового элемента в декартово дерево: роль ключа будет играть аргумент метода, а роль приоритета — случайное действительное число

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#treap.

3.4.3. Применение рандомизированных декартовых деревьев

Как было показано в предыдущем подразделе, рандомизированные декартовы деревья являются основной областью применения декартовых деревьев. А теперь попробуем ответить на вопрос о применении рандомизированных декартовых деревьев.

В общем случае рандомизированные декартовы деревья можно использовать везде, где находят применение двоичные деревья поиска. В частности, в ситуациях, где требуется сбалансированное дерево, но достаточно гарантий в среднем, а не в худшем случае.

Еще один аспект, который следует учитывать, — всегда, когда используются рандомизация и границы «среднего случая», вероятность сохранности гарантий тем выше, чем больше количество элементов, а для небольших деревьев легко могут получиться асимметричные структуры. Однако для небольших деревьев разница в производительности между немного перекошенными и сбалансированными деревьями, очевидно, также будет менее значимой.

Двоичные деревья поиска, в свою очередь, часто используются для реализации словарей и множеств. Подробнее об этих структурах будет говориться в главе 4.

Есть и другие примеры применения рандомизированных декартовых деревьев: хранение данных, прочитанных из отсортированного потока, подсчет количества элементов меньше (больше) любого заданного элемента динамического набора и вообще все случаи, когда нужно поддерживать динамически меняющийся набор элементов в отсортированном порядке, имея при этом возможность быстрого поиска, вставки и удаления.

Практическими примерами использования двоичных деревьев поиска могут служить управление набором областей виртуальной памяти (Virtual Memory Area, VMA) в ядре операционной системы и трассировка идентификаторов IP-пакетов. В последнем случае хеш-таблица была бы эффективнее, но она также и более уязвима для атак, когда злоумышленник может отправлять пакеты с IP-адресов, хеширующихся в одно и то же хеш-значение. Это превратит хеш-таблицу в несортированный список (если используется конкатенация[1]), она станет узким местом, так что злоумышленник, возможно, замедлит разрешение пакетов и работу ядра вообще.

3.5. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ И ПРОФИЛИРОВАНИЕ

Как было показано в разделе 3.3, все методы рандомизированных декартовых деревьев требуют времени, пропорционального высоте дерева. Как вы знаете (см. введение к этой главе и приложение В), в худшем случае высота двоичного дерева может линейно зависеть от количества элементов. И, по сути, одна из проблем двоичных деревьев поиска заключается в существовании определенных последовательностей операций вставки, которые обязательно вызовут перекос дерева. Эта проблема делает двоичные деревья поиска особенно уязвимыми для атак при использовании их в качестве словарей, потому что для снижения производительности структуры данных злоумышленнику достаточно отправить упорядоченную последовательность, которая

[1] Подробнее о хешировании рассказывается в приложении В.

приведет к вырождению дерева в связанный список, имеющий единственный путь от корня к листу, содержащему все элементы.

Рандомизированные декартовы деревья предлагают двойное улучшение. Прежде всего введение элемента случайности в назначение приоритетов не позволяет злоумышленникам[1] использовать известные последовательности. Но это также в среднем даст более сбалансированное дерево, как отмечалось в разделе 3.4.

Что значит «в среднем»? И какое улучшение можно получить? Есть два ответа на эти вопросы. С теоретической точки зрения можно проанализировать ожидаемую высоту рандомизированного декартова дерева и математически доказать, что в среднем высота будет логарифмической.

С практической точки зрения можно просто запустить симуляцию и убедиться, что наши ожидания верны, сравнив высоту двоичного дерева поиска и рандомизированного декартова дерева с одним и тем же набором элементов.

3.5.1. Теория: ожидаемая высота

Для оценки ожидаемой высоты рандомизированной структуры данных нужно ввести некоторые понятия из области статистики.

Прежде всего нам понадобится понятие ожидаемого значения для *случайной величины* V. Его можно неформально определить как среднее (не наиболее вероятное) значение, которое переменная примет в большом наборе событий.

Более формально, если V может принимать значения в конечном счетном множестве $\{v_1, v_2 \ldots v_M\}$ с вероятностями $\{p_1, p_2 \ldots p_M\}$, то ожидаемое значение V можно определить как:

$$E[V] = \sum_{i=1}^{M} v_i p_i.$$

Введем для нашего рандомизированного декартова дерева случайную величину D_k, определяющую глубину данного узла N_k, где индекс $k \in \{0 \ldots n-1\}$ обозначает индекс ключа узла в отсортированном наборе: N_k имеет k-й наименьший ключ в дереве.

Проще говоря, D_k определяет количество предков узла, содержащего k-й наименьший ключ, N_k. Это число можно также интерпретировать как количество узлов, находящихся в пути от корня дерева до N_k.

Формально:

$$D_k = \sum_{i=0}^{n-1} N_i \text{ — предок } N_k.$$

[1] Разумеется, это справедливо, если генераторы псевдослучайных чисел реализованы должным образом и в пределах, обусловленных неспособностью классических компьютеров обеспечить истинную случайность. В любом случае мы существенно усложняем задачу злоумышленникам.

Событие «N_i является предком N_k» можно обозначить индикатором (двоичной переменной) A_k^i: то есть для любых пар узлов N_i и N_k, $A_k^i == 1$ означает, что N_i находится в пути между N_k и корнем, а $A_k^i == 0$ означает, что они находятся в разных ветвях или, самое большее, N_i является потомком N_k.

Тогда ожидаемое значение для D_k превращается в:

$$E\left[D_k\right] = \sum_{i=0}^{n-1} 1P\left(N_i - \text{предок } N_k\right) = \sum_{i=0}^{n-1} P\left(A_k^i\right).$$

Для вычисления вероятности $P(A_k^i)$ нужно ввести новую переменную и лемму (промежуточный результат).

Определим $N(i, k) = N(k, i) = \{N_i, N_{i+1} \ldots N_{k-1}, N_k\}$ как подмножество узлов дерева с ключами между i-м и k-м наименьшими[1] ключами во всем дереве.

Очевидно, что $N(0, n-1) = N(n-1, 0)$ содержит все узлы декартова дерева. На рис. 3.16 показано несколько примеров этих подмножеств.

Обратите внимание, что последующий и предшествующий узлы для любого узла N всегда находятся на пути между N и корнем или между N и листом. Другими словами, чтобы найти предшествующий или последующий узел, достаточно просмотреть поддерево с корнем в N, где искомый последующий узел будет самым левым или самым правым, или путь между N и корнем.

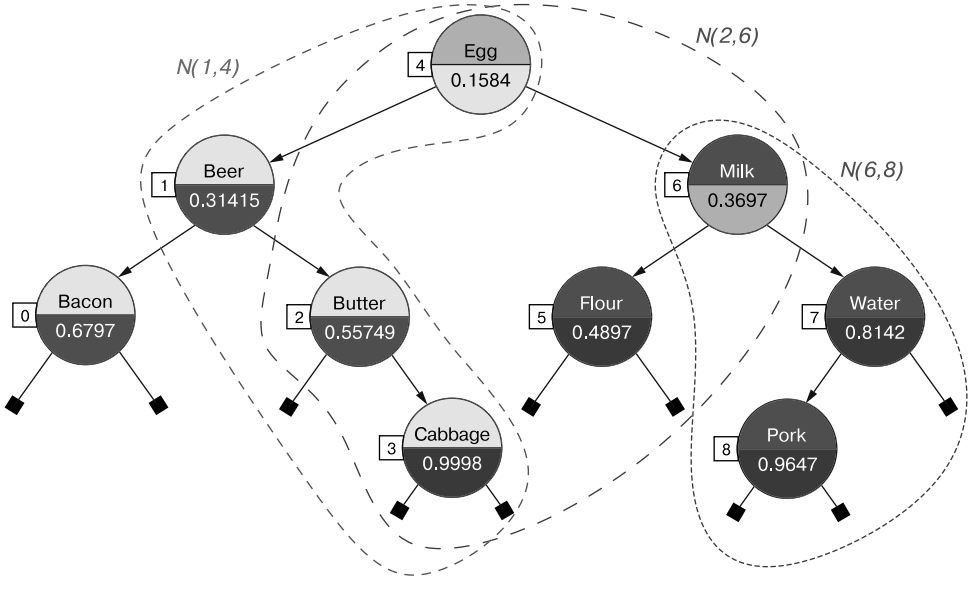


Рис. 3.16. Несколько примеров подмножеств узлов N(i, k), определенных в этом разделе

[1] Предполагается, что $i < k$; если $i > k$, то подмножество будет выглядеть так: $\{N_k, N_{k+1} \ldots N_{i-1}, N_i\}$.

Можно доказать, что справедлива следующая лемма.

> *Для всех $i = k$, $0 \le i, k \le n - 1$, N_i является предком N_k тогда и только тогда, когда N_i имеет наименьший приоритет среди всех узлов в $N(i, k)$.*

Я не буду приводить доказательство этой леммы. При желании вы легко сможете сделать это самостоятельно, применив метод индукции.

Итак, вооружившись леммой, можно вычислить вероятность, с которой узел с `i`-м наименьшим ключом станет предком узла с k-м наименьшим ключом. Поскольку предполагается, что приоритеты взяты из однородного непрерывного множества всех действительных чисел от 0 до 1, то каждый узел в подмножестве узлов с равной вероятностью будет иметь наименьший приоритет.

Следовательно, для каждого $i \neq k$ можно сформулировать вероятность того, что `i` является предком k, как:

$$P\left(A_k^i\right)_{i \neq k} = \frac{1}{\left|N\left(i, k\right)\right|} = \frac{1}{\left|k - i\right| + 1},$$

в то время как для $i = k$ вероятность просто равна 0 (узел не может быть собственным предком).

Подставляя эти значения в формулу ожидаемого значения D_k, получаем:

$$E\left[D_k\right] = \sum_{i=0}^{n-1} P\left(A_k^i\right) = \sum_{i=0}^{k-1} \frac{1}{k - i + 1} + \sum_{i=k}^{k} 0 + \sum_{i=k+1}^{n-1} \frac{1}{i - k + 1}.$$

Средний член суммы, очевидно, оценивается как 0, тогда как в первом члене, когда $i = 0$, знаменатель становится равным $k - 1$ и уменьшается на 1 по мере увеличения i, пока $i = k - 1$ не станет равным 2.

Аналогичные соображения можно привести для последнего члена формулы, что дает нам:

$$E\left[D_k\right] = \sum_{j=2}^{k-1} \frac{1}{j} + \sum_{j=2}^{n-k} \frac{1}{j} = \sum_{j=1}^{k-1} \frac{1}{j} - 1 + \sum_{j=1}^{n-k} \frac{1}{j} - 1 = H_{k-1} - 1 + H_{n-k} - 1.$$

Обе суммы в предыдущей формуле на самом деле являются частичными суммами гармонического ряда, обозначаемого как H_n; $H_n < \ln(n)$, где ln — натуральный логарифм. В конечном итоге получаем:

$$E\left[D_k\right] = H_{k-1} + H_{n-k} - 2 < \ln\left(k - 1\right) + \ln\left(n - k\right) - 2 < 2\ln\left(n\right) - 2.$$

Этот результат гарантирует, что при большом количестве попыток среднее значение высоты рандомизированного декартова дерева равно O(log(n)), логарифму от количества хранимых ключей (независимо от порядка добавления, удаления и распределения ключей).

3.5.2. Экспериментальное определение высоты

Вы можете возразить: это, мол, всего лишь гарантия в среднем по нескольким попыткам. А что, если нам не повезет в самый неудобный момент? Чтобы лучше понять фактическую производительность этой структуры данных, можно провести небольшой эксперимент, как это было сделано в разделе 2.10 в отношении *d*-ичной кучи. На этот раз используем инструмент профилирования для Java, JProfiler[1].

Мы могли бы вообще обойтись без инструмента профилирования, потому что наши тесты будут всего лишь оценивать реализации простого двоичного дерева поиска и рандомизированного декартова дерева и после выполнения одной и той же последовательности операций с экземплярами этих контейнеров сравнят их высоты.

Этот эксперимент даст нам представление об асимптотическом улучшении сбалансированных деревьев по сравнению с двоичным деревом поиска, потому что, как обсуждалось выше, операции с двоичными деревьями (сбалансированными или нет) выполняются за количество шагов, пропорциональное высоте дерева.

Однако также известно, что в асимптотическом анализе принято отбрасывать постоянные коэффициенты. А это скрывает сложность кода, которая обычно возникает при использовании более продвинутых алгоритмов. Таким образом, оценка фактического времени работы предоставит дополнительную информацию, которая может помочь в выборе наилучшей реализации для наших приложений.

В тестах, которые доступны в репозитории книги на GitHub[2], опробуются три сценария:

- создание больших деревьев возрастающего (случайного) размера со случайными целыми числами в качестве ключей и с использованием начальной последовательности операций вставки, за которой следуют случайные удаления и вставки (в отношении 1:1);
- то же, что и в предыдущем сценарии, но возможные значения ключей ограничены более узким диапазоном (например, 0... 100, что приведет к созданию дубликатов в дереве);
- вставка в деревья упорядоченной последовательности чисел.

Результаты первого теста показаны на рис. 3.17. Как можно видеть, высота обоих деревьев растет логарифмически. Поначалу это кажется довольно обескураживающим, потому что, по всей видимости, рандомизированное декартово дерево не дает никаких выгод по сравнению с простым двоичным деревом поиска.

---

[1] JProfiler — это коммерческий инструмент. Но для этой работы можно также использовать альтернативы с открытым исходным кодом.

[2] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction/blob/master/Java/tests/org/mlarocca/containers/treap/RandomizedTreapProfiling.java.

Но давайте рассмотрим пару важных моментов.

Во-первых, правила игры: задан целевой размер дерева *n*, затем (в оба контейнера) добавляются одни и те же целые числа, выбранные случайным образом без всяких ограничений. После вставки этих чисел выполняется еще *n* операций. Каждая из них может удалить существующий ключ (выбранный случайным образом) или добавить новое случайное целое число. Очевидно, что в процессе тестирования размеры деревьев растут.

Здесь использована эффективная реализация двоичного дерева поиска (ее можно найти в репозитории книги[1]), которая ограничивает эффект перекоса при удалении элементов[2]. Этот прием улучшает баланс двоичного дерева поиска и уменьшает разрыв со сбалансированными деревьями.

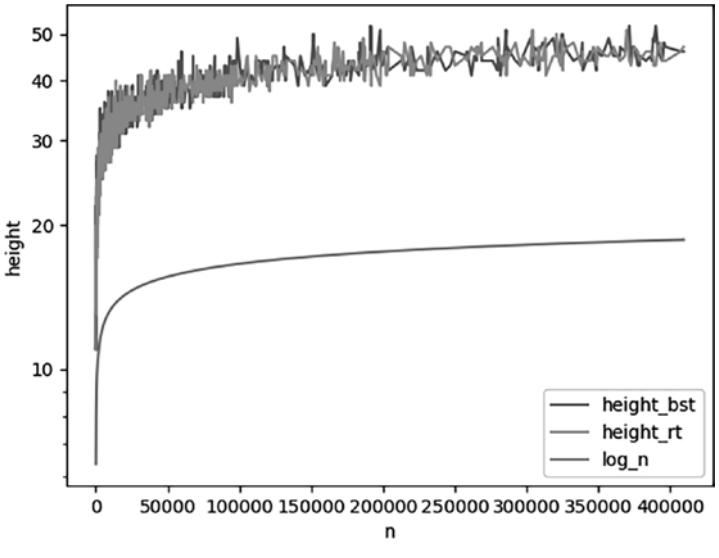


Рис. 3.17. Высота рандомизированного декартова дерева по сравнению с высотой двоичного дерева поиска. С обеими структурами данных были выполнены одни и те же операции, а ключи — случайные целые числа. Обратите внимание, что ось Y имеет логарифмическую шкалу

---

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction/blob/master/Java/src/org/mlarocca/containers/tree/BST.java.

[2] Стандартная реализация метода remove в двоичном дереве поиска, когда удаляемый узел N имеет двух потомков, использует последующий узел для замены удаляемого ключа (и затем рекурсивно удаляет этот предшествующий узел). При большом количестве удалений дерево становится перекошенным, наклоненным влево. Для смягчения этого эффекта применяется решение, с вероятностью 50 % использующее предшествующий узел вместо последующего.

Во-вторых, в этом эксперименте добавляются совершенно случайные ключи, а так как диапазон возможных значений намного больше количества элементов в контейнере, ожидаемое количество дубликатов незначительно, и можно предположить, что все последовательности вставляемых ключей будут иметь одинаковую вероятность. В этом сценарии маловероятен выбор последовательности, из-за которой высота дерева окажется суперлогарифмической (шансы очень малы уже при n ~= 100).

По сути, мы используем ту же концепцию рандомизированных декартовых деревьев, просто элемент случайности перемещается в генератор последовательностей ключей для вставки. Но, к сожалению, не всегда есть возможность выбирать данные для добавления в наши контейнеры!

Для проверки этой гипотезы проведем второй эксперимент, уменьшив влияние генератора случайных чисел за счет ограничения диапазона допустимых ключей. Результат ограничения множества допустимых значений ключей диапазоном 0... 1000 показан на рис. 3.18. Теперь сразу видна разница между двумя структурами данных: двоичное дерево поиска растет линейно с наклоном примерно 10^{-3}, тогда как рандомизированное декартово дерево по-прежнему имеет логарифмическую высоту.

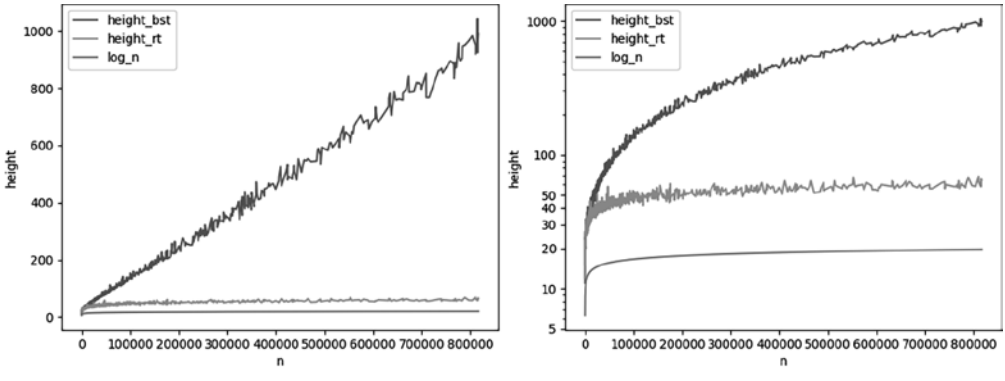


Рис. 3.18. Изменение высоты двоичного дерева поиска и рандомизированного декартова дерева в случае ограничения подмножества случайных целых диапазоном от 0 до 1000. Диаграммы показывают те же значения высоты: *слева* — в линейной шкале, а *справа* — в логарифмической

Эти различия обусловлены двумя причинами:

- при большом количестве дубликатов (их количество увеличивается с ростом n) создаются последовательности вставок с более длинными сериями отсортированных данных;
- с увеличением числа дубликатов двоичное дерево поиска естественным образом склоняется влево. То же наблюдалось в листинге 3.5: нужно было разорвать связь при вставке нового ключа и при обнаружении дубликата всегда выбирать

левое поддерево. К сожалению, в рассматриваемом сейчас случае необходимо детерминированное решение. Нельзя использовать тот же обходной маневр, что и для удаления, поэтому, когда входные последовательности содержат большое количество дубликатов, как в этом тесте, двоичное дерево поиска ощутимо искажается.

Это уже хороший результат для декартовых деревьев, потому что реальные входные данные вполне могут содержать несколько дубликатов.

А что можно сказать о случаях, когда дубликаты не допускаются? Защищены ли мы от неудачных последовательностей в таких ситуациях? По правде говоря, мы с вами до сих пор не уточнили, насколько важна роль упорядочения. Что может быть лучше, чем попробовать наихудший вариант — полностью упорядоченную последовательность? На рис. 3.19 показаны результаты этого тестирования. Здесь исключена всякая случайность, просто в контейнеры добавлены все целые числа от 0 до $n - 1$, никакие элементы при этом не удалялись.

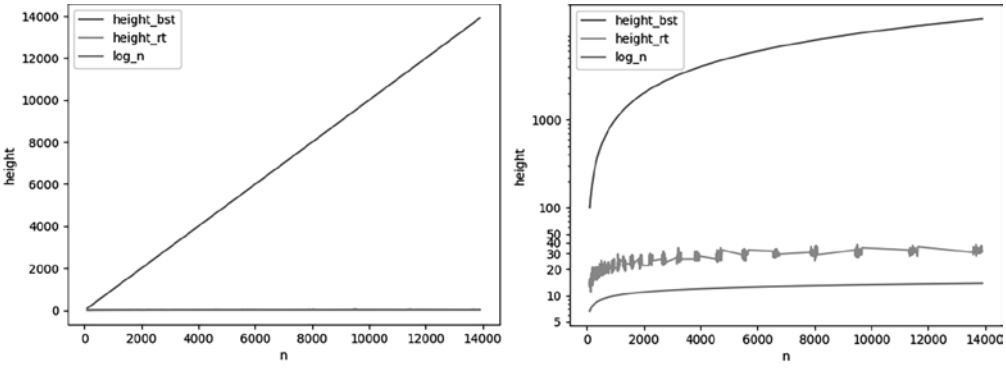


Рис. 3.19. Высота рандомизированного декартова дерева и двоичного дерева поиска после вставки последовательности чисел 0, 1... n – 1. На диаграммах видны те же значения высоты: *слева* в линейной шкале, а *справа* — в логарифмической

Как и ожидалось, высота двоичного дерева поиска не просто растет линейно, но всегда равна количеству узлов (на этот раз наклон равен 1), потому что дерево, как и ожидалось, вырождается в связанный список.

Рандомизированное декартово дерево, напротив, продолжает расти логарифмически, как и в других тестах, даже в самом неблагоприятном случае.

Исходя из результатов тестирования, можно сделать вывод, что если параметром, который нужно улучшить, является высота дерева, то рандомизированные декартовы деревья действительно имеют преимущество перед двоичными деревьями поиска и сохраняют логарифмическую высоту во всех ситуациях. Поэтому их производительность сравнима с более сложными структурами данных, такими как красно-черные деревья.

ОПАСНОСТИ РЕКУРСИИ

Вырождение двоичного дерева поиска в этом тесте привело к сбою в рекурсивной реализации метода add, представленного в нашем репозитории. Этот сбой был вызван переполнением стека[1] для n ~= 15 000 элементов.

Здесь уместно еще раз вспомнить, насколько важно быть осторожными при написании рекурсивных методов, а также насколько важно использовать правильную структуру данных. Теоретически декартовы деревья тоже могут вызывать переполнение стека, но поскольку их высота является логарифмической, потребуется добавить ~$2^{15\,000} = 10^{5000}$ элементов, чтобы вызвать подобное переполнение (что в любом случае намного больше, чем можно выделить в ОЗУ любого компьютера, так что такая ситуация маловероятна).

3.5.3. Профилирование времени выполнения

Высота дерева не единственный важный критерий. Хотелось бы еще знать, не кроется ли здесь подвох с точки зрения времени выполнения и использования памяти (рис. 3.20).

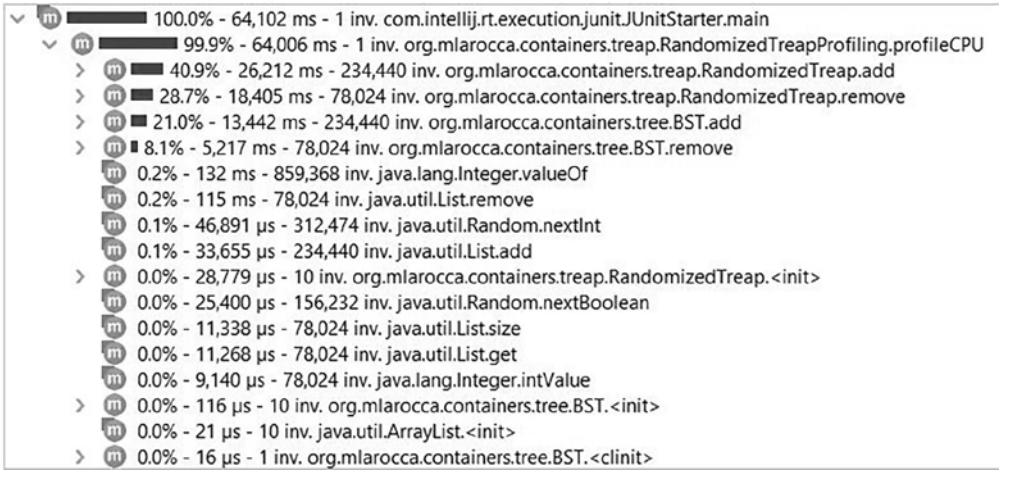


Рис. 3.20. Профилирование использования процессорного времени при вставке/удалении случайных неограниченных целых чисел из двоичного дерева поиска и рандомизированного декартова дерева

[1] Переполнение стека, его связь с рекурсией и способы предотвращения таких проблем обсуждаются в приложении Д.

Чтобы ответить на этот вопрос, выполним профилирование первого теста (с неограниченными целыми числами в качестве ключей и, соответственно, с небольшим возможным количеством дубликатов или даже без них) с помощью JProfiler. При профилировании используем реализации двоичного дерева поиска и рандомизированного декартова дерева, представленные в репозитории книги. Стоит отметить, что профилирование дает информацию только об исследуемых реализациях, и можно получить разные результаты для оптимизированной и неоптимизированной реализации.

Результаты профилирования показаны на рис. 3.20. Видно, что совокупное время работы вставки (метода `add`) для рандомизированного декартова дерева почти в два раза больше, чем для двоичного дерева поиска; для метода `remove` это соотношение увеличивается до 3,5 раза.

Результаты этого первого теста наводят на мысль, что в общем случае слишком велики оказываются накладные расходы из-за большей сложности рандомизированного декартова дерева. Однако такой результат вполне ожидаем, ведь при примерно одинаковой высоте деревьев реализация рандомизированного декартова дерева существенно сложнее.

И что теперь? Можно выбросить и забыть наш класс `RandomizedTreap`? Не торопитесь! Давайте посмотрим на результаты профилирования второго тестового примера, представленного в подразделе 3.5.2, где в контейнеры добавляются случайные целые числа из ограниченного диапазона [0, 1000].

В предыдущем подразделе было показано, что в этом случае высота двоичного дерева поиска растет линейно, тогда как высота рандомизированного декартова дерева продолжает расти логарифмически.

На рис. 3.21 размещен результат профилирования этого теста. Сразу видно, что ситуация радикально поменялась. Теперь двоичное дерево поиска работает намного хуже. Настолько хуже, что совокупное время выполнения метода `BST::add` в 8 раз превышает совокупное время выполнения `RandomizedTreap::add`. Для метода `remove` соотношение еще хуже; рандомизированное декартово дерево показывает производительность почти в 15 раз лучшую. Это как раз та ситуация, когда предпочтительнее использовать нашу новую, необычную, сбалансированную структуру данных!

Для полноты картины рассмотрим также наихудший сценарий для двоичных деревьев поиска. На рис. 3.22 показаны результаты профилирования последнего теста, представленного в подразделе 3.5.2, где в контейнеры вставляется упорядоченная последовательность. В этом случае, как мне кажется, результаты даже не нуждаются в комментариях, потому что речь идет о тысячах секунд против микросекунд (пришлось уменьшить тестовый набор, потому что производительность двоичного дерева поиска ухудшилась до неприемлемых показателей).

Рис. 3.21. Профилирование использования процессорного времени при вставке (удалении) случайных целых чисел из диапазона от 0 до 1000 в двоичное дерево поиска и рандомизированное декартово дерево

Учитывая все обстоятельства, из этих результатов можно сделать вывод, что если данные распределены равномерно по всему диапазону и количество дубликатов невелико, то стоит хорошенько подумать, прежде чем задействовать рандомизированные декартовы деревья. Напротив, если есть уверенность, что данные будут иметь много дубликатов или добавляться в порядке, близком к порядку сортировки, то определенно стоит избегать применения простых двоичных деревьев поиска и отдавать предпочтение сбалансированным деревьям.

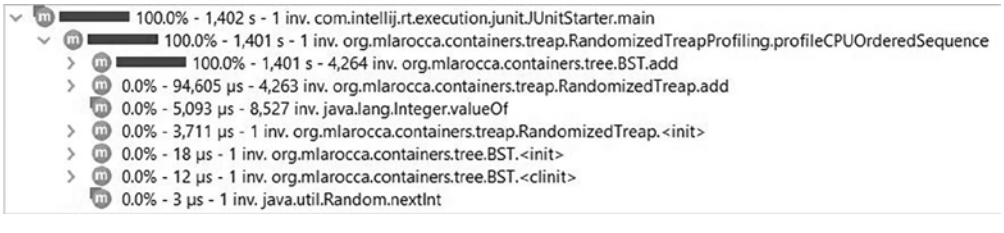


Рис. 3.22. Профилирование использования процессорного времени при вставке упорядоченной последовательности целых чисел в двоичное дерево поиска и рандомизированное декартово дерево

3.5.4. Профилирование потребления памяти

В отношении процессорного времени все понятно, а что можно сказать о потреблении памяти? Возможно, в некоторых ситуациях рандомизированное декартово дерево действительно работает быстрее, но требует так много пространства, что большие наборы данных могут не уместиться в памяти.

Для начала примем во внимание следующее: потребление памяти будет примерно одинаковым у всех тестов, представленных в предыдущих подразделах (конечно, для контейнеров одинакового размера). Это связано с тем, что количество узлов в обоих случаях не зависит от высоты деревьев. Такие деревья не поддерживают сжатие, а сбалансированным и асимметричным деревьям всегда потребуется `n` узлов для хранения `n` ключей.

Значит, можно вполне удовлетвориться результатами профилирования распределения памяти в наиболее общем случае, когда оба дерева сбалансированы. На рис. 3.23 показано совокупное потребление памяти, выделенной для экземпляров двух классов.

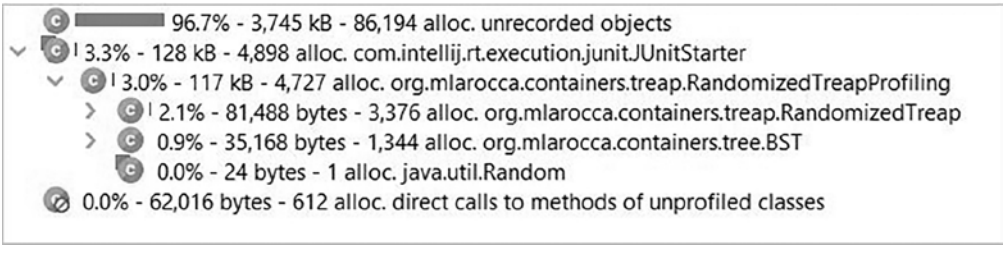


Рис. 3.23. Совокупный объем памяти, выделенной для двоичного дерева поиска и рандомизированного декартова дерева в наиболее общем случае, со случайными вставками и удалениями

Согласно результатам, рандомизированное декартово дерево `RandomizedTreap` потребляет памяти в два с небольшим раза больше, чем двоичное дерево поиска `BST`. Это неприятно, но вполне ожидаемо, если учесть, что каждый узел в декартовом дереве хранит не только ключ (типа `Integer` в этом тесте), но также приоритет типа `Double`.

Если попробовать использовать ключи другого типа, например `String`, то разница окажется намного меньше, как показано на рис. 3.24. При хранении ключей в виде строк с длиной от четырех до десяти символов соотношение в потреблении памяти уменьшается до 1,25.

Hot Spot	Total Allocated Memory	Allocations
com.intellij.rt.execution.junit.JUnitStarter	6,813 kB (100 %)	350,907
org.mlarocca.containers.treap.RandomizedTreapProfiling	6,803 kB (99 %)	350,691
org.mlarocca.containers.treap.RandomizedTreap	3,558 kB (52 %)	183,372
org.mlarocca.containers.treap.Treap	3,425 kB (50 %)	177,832
org.mlarocca.containers.treap.Treap$TreapNode	3,351 kB (49 %)	174,139
org.mlarocca.containers.tree.BST	2,836 kB (41 %)	149,718
org.mlarocca.containers.tree.BST$BSTNode	2,765 kB (40 %)	146,201
java.lang.Integer	200 kB (2 %)	8,937
java.lang.Double	88,152 bytes (1 %)	3,673
org.mlarocca.containers.treap.Treap$TreapEntry	44,688 bytes (0 %)	1,862
java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock	64 bytes (0 %)	2
java.util.concurrent.locks.ReentrantReadWriteLock	48 bytes (0 %)	3

Рис. 3.24. Совокупный объем памяти, выделенной для двоичного дерева поиска и рандомизированного декартова дерева при хранении ключей в виде строк длиной от четырех до десяти символов

3.5.5. Выводы

Сравнительный анализ производительности и высоты двоичных деревьев поиска и рандомизированных декартовых деревьев показывает, что последние немного проигрывают в потреблении памяти и в скорости в общем случае. Но, когда нет гарантий неупорядоченности ключей или операций, двоичные деревья поиска могут стать узким местом с еще большей вероятностью.

Знакомясь со структурами данных в главе 1, мы выяснили, что умение выбирать правильные структуры данных — это скорее умение избежать неправильного выбора. Здесь как раз такой случай. Разработчики должны знать о ситуациях, когда предпочтительнее использовать сбалансированное дерево, чтобы избежать атак или простого снижения производительности.

Стоит повторить, что первая часть анализа, предметом которой является высота деревьев, ценна сама по себе и не зависит от используемого языка программирования[1]. Но анализ времени выполнения и потребления памяти имеет ценность только для конкретной реализации, языка программирования, вариантов организации и т. д. Теоретически все эти аспекты можно оптимизировать под конкретные требования к приложению.

Я советую всегда тщательно анализировать требования, выяснять, что критично в вашем ПО и где нужны определенные гарантии, касающиеся времени выполнения и потребления памяти, а затем тестировать критические участки. Не тратьте время на некритические разделы; практически для любого программного обеспечения остается верным принцип Парето, согласно которому можно увеличить производительность на 80 %, оптимизировав 20 % кода. Точное соотношение может варьироваться в каждом конкретном случае, но главная направленность принципа, заключающаяся в том, что можно добиться значительного улучшения за счет оптимизации наиболее важных частей приложения, скорее всего, останется в силе.

Постарайтесь найти баланс между чистотой кода, временем, затраченным на его разработку, и эффективностью. «Преждевременная оптимизация — корень всех зол», как сказал Дональд Кнут (Donald Knuth)[2], потому что попытка оптимизировать весь код, скорее всего, отвлечет вас от поиска критических проблем и в результате вы получите менее чистый, хуже читаемый и менее качественный код, более сложный в поддержке.

---

[1] Если предположить, что алгоритмы реализованы дословно, то это также не зависит от конкретной реализации.

[2] «Настоящая проблема заключается в том, что программисты не там, где нужно, и не тогда, когда нужно, тратят слишком много времени на заботу об эффективности. Преждевременная оптимизация — корень всех зол (или, по крайней мере, большей их части) в программировании». *Кнут Дональд Эрвин* (Knuth, Donald Ervin). The art of computer programming. — Pearson Education, 1997. (В России эта книга издавалась в виде серии с названием «Искусство программирования». Последнее переиздание — в 2022 году. — *Примеч. пер.*)

Всегда старайтесь сначала написать чистый код, а затем оптимизируйте узкие места и критические разделы, особенно те, для которых действует соглашение об уровне обслуживания с требованиями к времени выполнения и объему потребляемой памяти.

Вот вам конкретный пример: реализация на Java, которую можно найти в репозитории книги, интенсивно использует класс `Optional` (чтобы избежать использования `null` и предоставить более удобный интерфейс и лучший способ обработки неудачных операций) и, как следствие, множество лямбда-функций.

Если более детально исследовать потребление памяти, отключив фильтр профилирования пакета (для ускорения, потому что обычно не требуется фиксировать показатели работы стандартной библиотеки), то конечный результат может кого-то из вас удивить (рис. 3.25).

Name	Instance Count	Size
java.util.Optional	2,301,196	36,819 kB
org.mlarocca.containers.tree.BST$BSTNode$$Lambda$76.1390835631	790,363	18,968 kB
org.mlarocca.containers.tree.BST$BSTNode$$Lambda$75.1397616978	790,353	12,645 kB
org.mlarocca.containers.treap.Treap$TreapNode$$Lambda$79.874217	408,841	6,541 kB
org.mlarocca.containers.treap.Treap$TreapNode$$Lambda$77.889729	331,995	7,967 kB
org.mlarocca.containers.treap.Treap$TreapNode$$Lambda$78.148912	331,995	7,967 kB
org.mlarocca.containers.treap.Treap$TreapNode$$Lambda$83.108887	315,976	5,055 kB
org.mlarocca.containers.treap.Treap$TreapNode$$Lambda$86.105296	239,130	5,739 kB
org.mlarocca.containers.treap.Treap$TreapNode$$Lambda$87.121009	239,130	5,739 kB
byte[ ]	207,667	5,805 kB
java.lang.String	207,221	4,973 kB
org.mlarocca.containers.tree.BST$BSTNode$$Lambda$85.274722023	109,747	2,633 kB
org.mlarocca.containers.tree.BST$BSTNode$$Lambda$84.453523494	109,744	1,755 kB
java.lang.Double	60,505	1,452 kB
org.mlarocca.containers.treap.Treap$TreapNode	60,504	2,420 kB
org.mlarocca.containers.tree.BST$BSTNode	60,504	1,936 kB

Рис. 3.25. Распределение памяти по классам в нашем тесте без фильтрации по пакету

Как видите, основной объем памяти используется экземплярами `Optional` и лямбда-выражениями (неявно созданными в `Optional::map` и т. д.).

Дополнительное ускорение может дать поддержка многопоточности. Если приложение не предполагает совместного использования этих контейнеров различными потоками, то можно не стараться сделать реализацию потокобезопасной и сэкономить на накладных расходах, связанных с созданием и синхронизацией блокировок.

ВЫБОР МЕЖДУ ЧИСТЫМ И ОПТИМИЗИРОВАННЫМ КОДОМ

Если производительность и потребление памяти критичны для приложения, то можно написать другую, оптимизированную версию кода, отказавшись от использования всех этих замысловатых новых возможностей языка. Также можно отказаться от рекурсии и использовать явные циклы.

Но если требования к эффективности стоят не на первом месте, то лучше предпочесть более чистый и удобный в сопровождении код, использующий более удачные интерфейсы и API, потому что в долгосрочной перспективе читабельный код намного упростит вашу жизнь и работу будущих членов команды.

РЕЗЮМЕ

- Двоичные деревья поиска отличаются хорошей производительностью всех типичных методов контейнера, но только будучи сбалансированными. Однако в зависимости от порядка вставки ключей двоичное дерево поиска может оказаться перекошенным.
- Пограничный случай — когда в двоичное дерево поиска добавляется упорядоченная последовательность, в результате чего создается дерево, содержащее единственный путь длиной n, де-факто вырожденное в связанный список.
- Декартово дерево (treap) — это гибрид двоичного дерева и кучи, соблюдающий инварианты двоичного дерева поиска для ключей и инварианты кучи для приоритетов.
- Если назначаемые приоритеты извлекаются случайным образом из равномерно распределенного и непрерывного множества значений (например, из множества действительных чисел в диапазоне от 0 до 1), то можно математически доказать, что при достаточно больших значениях n дерево будет хранить n элементов и поддерживать высоту не более $2 \log(n)$.
- Помимо теоретических доказательств, можно проверить на практике (что мы и сделали, использовав реализации на Java), что рандомизированные декартовы деревья сохраняют логарифмическую высоту даже в сценариях, наихудших для двоичных деревьев поиска.
- Кроме того, в общем случае обе структуры демонстрируют сопоставимую производительность с точки зрения затрат процессорного времени и потребления памяти, но в пограничных случаях, когда двоичные деревья сдают позиции, производительность рандомизированных декартовых деревьев оказывается намного выше.

4 Фильтры Блума: отслеживание содержимого с меньшими затратами памяти

В этой главе

- ✓ Описание и анализ фильтров Блума.
- ✓ Отслеживание больших документов с использованием небольшого объема памяти.
- ✓ Почему словари не лучшее решение.
- ✓ Уменьшение размера отпечатка памяти с помощью фильтров Блума.
- ✓ Когда фильтры Блума улучшают производительность.
- ✓ Использование метрик для настройки качества фильтров Блума.

Начиная с этой главы, мы будем рассматривать менее популярные структуры данных, решающие, как это ни странно, распространенные проблемы. *Фильтры Блума* — один из самых ярких примеров таких структур; они часто используются в разнообразных отраслях, но известны не так широко, как можно было бы ожидать.

В разделе 4.1 представлена задача, которая станет путеводной звездой в этой главе: отслеживание больших сущностей с хранением наименьшего возможного отпечатка памяти.

В разделе 4.2 мы обсудим несколько более сложных решений и исследуем их сильные и слабые стороны; последние, в частности, разработчики алгоритмов могут воспринимать как возможность и благодатную почву для улучшения.

В процессе обсуждения вы познакомитесь со *словарем* — абстрактным типом данных, который представлен в разделе 4.3, а в разделе 4.4 переключимся на конкретные структуры данных, реализующие словари: хеш-таблицы, двоичные деревья поиска и фильтры Блума.

Вы, наверное, догадались, что наибольший интерес для нас представляет последняя из этих структур, она и есть главная тема этой главы. В разделе 4.5 описаны принципы работы фильтра Блума, а еще ниже, в разделе 4.6, подробно исследуется каждый из его методов на примерах псевдокода важнейших частей.

Раздел 4.7 завершает первую часть главы обсуждением некоторых типичных вариантов использования фильтров Блума: от распределенных баз данных и файловых систем до маршрутизаторов.

Для достижения главной цели я постарался сохранить практическую направленность материала, чтобы оснастить вас необходимыми инструментами и научить определять, когда можно использовать фильтры Блума.

В разделе 4.8 акцент смещается в теоретическую плоскость. Здесь будет представлена справочная информация для читателей, желающих понять, не только как работают фильтры Блума, но и почему они функционируют именно так. Чтобы лучше понять эти разделы, читатели могут предварительно заглянуть в приложение Е (а также в приложения Б и В), там представлен материал по рандомизированным алгоритмам, нотации «О большое» и основным структурам данных.

В следующих нескольких разделах исследуется производительность рассматриваемой структуры данных, включая время выполнения и размер отпечатка памяти (раздел 4.9), а также точность алгоритма (раздел 4.10).

Наконец, в разделе 4.11 описываются некоторые наиболее совершенные варианты фильтров Блума, используемые для предоставления новых возможностей или снижения вероятности ложноположительных результатов.

4.1. СЛОВАРИ: ОТСЛЕЖИВАНИЕ СОДЕРЖИМОГО

Рассмотрим гипотетический сценарий: вы работаете в компании, достаточно большой, чтобы иметь свою службу электронной почты. Служба устарела и поддерживает только базовые возможности. После последней реорганизации новый технический директор решает заново создать почтовую службу, чтобы получить возможность действовать более агрессивно на рынке, и поручает вашему руководителю заняться переделкой продукта.

Согласно заданию новая клиентская программа электронной почты должна поддерживать операции со списками контактов и иметь другие крутые «фишки». Например, при добавлении нового получателя электронного письма приложение должно проверить, есть ли он в списке контактов, и, если его там нет, вывести диалог

с предложением добавить нового получателя в список контактов (например, как показано на рис. 4.1).

И, само собой разумеется, реализовать эту функцию поручили именно вам.

Ресурсов на проект выделено мало, поэтому вы займетесь только рефакторингом клиентской программы, а на стороне сервера придется оставить унаследованный код и унаследованные службы, работающие на проприетарных машинах.

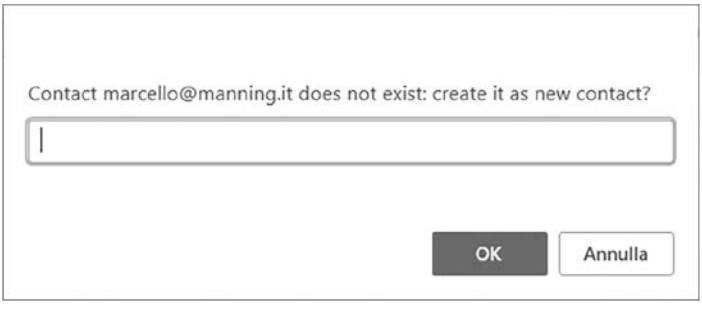


Рис. 4.1. Клиент электронной почты предлагает добавить нового получателя в список контактов после его неудачного поиска

Об обращении к базе данных для сверки адреса электронной почты со списком контактов не может быть и речи: вы ограничены ресурсами, доступными на этой устаревшей машине, и не имеете средств на реорганизацию и горизонтальное масштабирование[1]. Ваша БД не может обрабатывать больше нескольких запросов в секунду, а по прогнозу вашего руководства, каждую секунду будут составляться сотни электронных писем (они, конечно, чересчур оптимистичны и не слишком точны в своих прогнозах, но давайте пока не будем об этом беспокоиться!). Первая мысль, что приходит вам в голову, — настроить удаленный распределенный кэш, такой как Memcached, Cassandra или Redis. Однако даже простое обращение к серверу кэша заняло бы в лучшем случае около сотни миллисекунд. К тому же ограниченность бюджета не позволяет ни развернуть новый сервер для кэша, ни купить его как облачный сервис.

В конце концов вы приходите к выводу, что у вас есть только один способ решить задачу: асинхронно получить список контактов при входе в систему (или при первом щелчке на кнопке Создать в текущем сеансе браузера), сохранить список контактов

[1] Под вертикальным масштабированием (scale up) понимается перемещение приложения на более мощный и дорогой компьютер, а под горизонтальным масштабированием (scale out) обычно понимают реорганизацию приложения для работы в распределенной архитектуре на нескольких недорогих машинах. В отличие от вертикального масштабирования, имеющего верхний предел (самая мощная машина, которую может купить компания с учетом того, что цены в сегменте высокопроизводительной вычислительной техники растут экспоненциально), при правильном проектировании горизонтальное масштабирование теоретически не имеет пределов.

в сеансовом хранилище веб-страницы и сверяться с этой локальной копией каждый раз, когда требуется проверить существование контакта.

На рис. 4.2 показана возможная архитектура минимально жизнеспособного продукта. Теперь нужно просто найти эффективный способ просмотра списка контактов и сверки с ним адреса электронной почты.

Поиск определенной записи в списке — распространенная задача в информатике, известная как *задача словаря*.

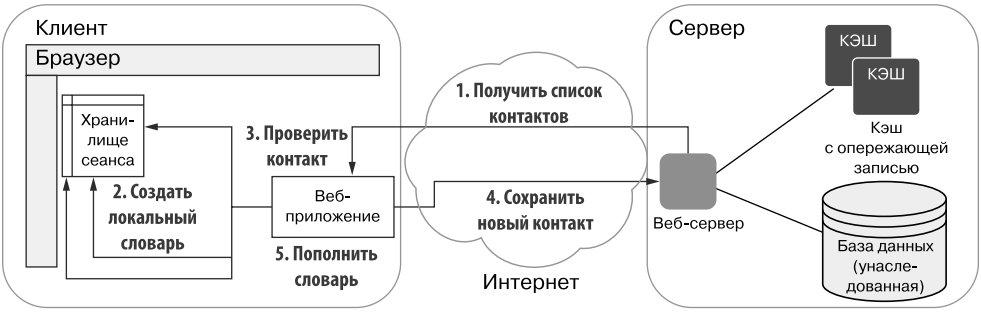


Рис. 4.2. Возможная архитектура функции «сохранить новый контакт (с предложением)». На стороне клиента веб-приложение загружает список контактов с веб-сервера и на его основе создает словарь в хранилище сеанса. Всякий раз, когда пользователь добавляет получателя электронного письма, веб-приложение проверяет словарь. Если получателя нет в списке контактов, то открывается диалог, предлагающий пользователю сохранить новый контакт. Если пользователь ответил утвердительно, то для сохранения нового контакта веб-серверу отправляется еще один HTTP-запрос и одновременно в локальный словарь добавляется новая запись (без прохождения через сервер)

4.2. АЛЬТЕРНАТИВЫ РЕАЛИЗАЦИИ СЛОВАРЯ

Название раздела не должно удивлять; ведь то, что должно делать приложение, можно сравнить с поиском слова в словаре (здесь под словарем я понимаю одну из тех старых огромных бумажных книг, которые были почти полностью вытеснены онлайн-словарями и поисковыми системами) или даже номера в телефонной книге (не пережившей третьей компьютерной революции).

Напомню, что наше веб-приложение для контактов должно:

- загружать список контактов с сервера;
- создавать локальную копию для быстрого поиска;
- позволять искать контакты;
- поддерживать возможность добавления нового контакта, если поиск адреса не увенчается успехом;
- синхронизировать список с сервером при добавлении нового контакта (или изменении существующего).

Соответственно, нам нужна структура данных, специализирующаяся на такого рода операциях; она должна поддерживать и быструю вставку, и в то же время поиск записи по значению.

Простой массив не позволяет эффективно (лучше, чем *сублинейно*) определять индекс, или присутствие в нем элемента `X`. Единственный способ узнать, имеется ли элемент в массиве, — обойти все его элементы, хотя, если он отсортирован, для ускорения можно использовать *поиск методом дихотомии*.

Например, представьте массив строк `["the", "lazy", "fox"]`. Чтобы отыскать строку `"lazy"`, мы будем вынуждены просмотреть все элементы в массиве.

Ассоциативный массив, напротив, по определению имеет метод, обеспечивающий эффективный способ доступа к хранимым записям с поиском по значению. Обычно эта структура позволяет хранить пары «ключ — значение». Например, список мог бы быть таким: `<("the", article), ("lazy", adjective), ("fox", noun)>`, и мы могли бы выполнить поиск слова `"lazy"`, а ассоциативный массив вернул бы `adjective`.

Еще одно его отличие от обычных массивов — порядок вставки в ассоциативный массив не имеет значения; более того, порядок хранения элементов в ассоциативном массиве вообще никак не регламентируется. Это цена, которую приходится платить за ускорение поиска по значению.

Чтобы понять, насколько эффективно можно решить эту задачу, нужно вникнуть в детали реализации. Использование абстракции «словарь» позволяет обсудить, как решить задачу (например, определить присутствие получателя электронного письма в списке контактов), не вдаваясь в детали представления структуры данных. Так мы можем сосредоточиться на самой задаче.

4.3. ОПИСАНИЕ API СТРУКТУРЫ ДАННЫХ: АССОЦИАТИВНЫЙ МАССИВ

Ассоциативный массив (также называемый *словарем*[1], *таблицей символов* или просто *отображением*) состоит из набора таких пар «ключ — значение», что:

- каждый возможный ключ появляется в коллекции не более одного раза;
- каждое значение можно получить с помощью соответствующего ключа.

Самый простой способ понять суть ассоциативного массива — представить обычный массив как частный случай ассоциативного массива, в котором роль индексов играют целые числа между 0 и размером массива минус 1 и всегда есть возможность получить значение элемента, указав его индекс. Так, (обычный) массив `["the", "lazy",`

[1] Я буду использовать термин «ассоциативный массив», чтобы избежать путаницы между проблемой словаря и абстрактным типом данных «словарь». Эти два термина взаимосвязаны, но не идентичны полностью.

"fox"] можно интерпретировать как словарь, хранящий ассоциации (0, "the"), (1, "lazy") и (2, "fox").

Ассоциативные массивы обобщают эту идею, позволяя использовать ключи практически любого типа.

Абстрактная структура данных: ассоциативный массив (он же словарь)	

API	`class Dictionary {`
	`  insert(key, value)`
	`  remove(key) → value`
	`  contains(key) → value`
	`}`

Контракт с клиентом	Словарь хранит все пары, добавленные клиентом. Если пара `(K,V)` была добавлена в словарь (и не удалена впоследствии), то `contains(K)` вернет `V`

Определив API, можно набросать решение первоначальной задачи.

Когда пользователи запускают клиента электронной почты, тот получает список контактов с сервера и сохраняет их в словаре, который можно хранить в памяти (список контактов, не помещающийся в хранилище сеансов браузера, — это что-то невероятное даже для звезды Instagram!). Если пользователь добавляет новый контакт в адресную книгу, выполняется вызов метода `insert` словаря. Аналогично, если пользователь удаляет существующий контакт, производится синхронизация словаря вызовом `remove`. Всякий раз, когда пользователь пишет электронное письмо и добавляет получателя, сначала проверяется словарь, и только если контакта нет в адресной книге, выводится окно диалога с предложением сохранить новый контакт.

Таким образом, клиент никогда не посылает HTTP-запрос серверу (и по цепочке — базе данных), чтобы проверить присутствие контакта в адресной книге, а чтение из базы данных происходит только один раз при запуске (или при составлении первого письма во время сеанса).

4.4. КОНКРЕТНЫЕ СТРУКТУРЫ ДАННЫХ

С теоретической точки зрения все выглядит просто, но практическая реализация ассоциативных массивов для использования в реальных системах — совсем другое дело.

Теоретически, если область возможных ключей достаточно узкая, все равно можно использовать массивы, упорядочив ключи и используя их позиции в упорядоченной последовательности как индексы в реальном массиве. Например, если область ключей состоит из слов {"a", "terrible", "choice"}, то можно отсортировать ключи лексикографически, а затем сохранить соответствующие им строковые значения в обычном массиве, например: {"article", "noun", "adjective"}. Чтобы представить словарь, содержащий только значение для ключа "choice", можно установить значения, соответствующие отсутствующим ключам, равными null: {null, "noun", null}.

Однако гораздо чаще набор возможных значений для ключей достаточно широк, что делает непрактичным использование массива, содержащего элементы для всех возможных ключей: это потребовало бы слишком много памяти, большая часть которой оставалась бы неиспользованной.

Далее рассмотрим две простейшие реализации и три наиболее широко используемые альтернативы, решающие эту проблему памяти.

4.4.1. Несортированный массив: быстрая вставка, медленный поиск

Даже если вы никогда не видели ни одного из этих бумажных динозавров (словарей), то должны быть знакомы с печатными книгами, такими как эта (если, конечно, вы не купили ее электронную версию!).

Предположим, вам нужно найти в этой книге определенное слово — может быть, имя, например Блум, — и отметить все места, где оно встречается. Один из возможных вариантов — это прочитать всю книгу, начиная с первой страницы, слово за словом. Если вы решите найти все вхождения слова «Блум», то придется прочитать книгу от корки до корки.

Книга, представленная в виде набора слов, расположенных в том порядке, в каком они напечатаны, подобна несортированному массиву. В табл. 4.1 перечислены основные операции и их производительность для случая использования несортированного массива.

Преимущество несортированных массивов в том, что при их создании не требуется дополнительной работы, а новые записи добавляются довольно просто, если, конечно, в массиве достаточно места.

Таблица 4.1. Использование несортированного массива в роли словаря

Операция	Время выполнения	Объем дополнительной памяти
Создание структуры	O(1)	Нет
Поиск элемента	O(*n*)	Нет
Вставка нового элемента	O(1)*	Нет
Удаление элемента	O(1)	Нет

* Амортизированное время.

4.4.2. Отсортированные массивы и поиск методом дихотомии: медленная вставка, быстрый поиск

Очевидно, что реализация на основе несортированного массива весьма непрактична. Если после прочтения всей книги понадобится найти второе слово, например «фильтр», то придется начать все сначала и еще раз прочитать сотни тысяч слов в книге. Вот почему у большинства книг есть так называемый индекс (алфавитный

указатель), обычно расположенный ближе к концу книги. Там можно найти список в алфавитном порядке самых необычных слов (и имен), используемых в книге. Обычные слова в него не включаются, потому что они используются слишком часто и не имеет смысла их перечислять: затраты на поиск мест, где они используются, минимальны. И наоборот, чем реже встречается слово (имена — прекрасный пример), тем большее значение оно имеет в тексте[1].

При наличии индекса можно заглянуть в него и отыскать имя Блум. Благодаря тому что индекс отсортирован в лексикографическом порядке, поиск в нем не займет много времени. Но, возможно, вам немного не повезет и понадобится отыскать такой термин, как «хеширование», находящийся ближе к концу индекса.

Поэтому, имея отсортированный список, мы часто бессознательно выполняем поиск методом дихотомии[2]: открываем телефонную книгу на случайной странице где-то посередине (или ближе к началу или концу, если примерно представляем, где может находиться искомое имя), затем сравниваем первую букву первого слова на этой странице с первой буквой в искомом имени и переходим либо к предыдущей, либо к следующей странице. Например, если мы все еще ищем имя Блум и открываем телефонную книгу на странице, где первая фамилия — Курц, то мы знаем, что можно не просматривать все последующие страницы и просмотреть только предыдущие. Затем случайно открываем другую страницу (перед страницей с фамилией Курц) и видим первую фамилию на этой странице — Барроу. Здесь мы понимаем, что фамилия Блум находится где-то после страницы с Барроу, но перед страницей с именем Курц.

Но вернемся к задаче со списком контактов. Итак, одним из решений могут быть сортировка контактов и поиск методом дихотомии.

Как показано в табл. 4.2, с точки зрения времени выполнения первоначальные затраты (на сортировку списка) и затраты на добавление нового элемента довольно высоки. Кроме того, может потребоваться дополнительная память, если понадобится создать копию исходного списка, чтобы сохранить исходный порядок.

---

[1] Это рассуждение также положено в основу метрики TF-IDF (сокращение от term frequency — inverse document frequency — «частота термина — обратная частота документа»), используемой в задачах поиска и анализа текстов. Метрика TF-IDF вычисляется как отношение количества вхождений термина в документ (TF) к логарифму другой дроби — общему количеству документов в корпусе, деленному на количество документов, в которых встречается этот термин (IDF). То есть чем чаще термин встречается в документе и реже в корпусе, тем выше будет его оценка TF-IDF. И наоборот, оценка TF-IDF будет тем ниже, чем больше число документов, где он используется.

[2] Метод дихотомии (деления пополам) получил свое название вследствие того, что поиск всегда начинается с точки деления списка пополам. Искомый элемент сравнивается с элементом в середине списка и в зависимости от результата сравнения поиск рекурсивно продолжается в первой или второй половине списка (или останавливается, если найдено искомое).

Таблица 4.2. Использование несортированного массива в роли словаря

Операция	Время выполнения	Объем дополнительной памяти
Создание структуры	O(*n* log(*n*))	O(*n*)
Поиск элемента	O(log(*n*))	Нет
Вставка нового элемента	O(*n*)	Нет
Удаление элемента	O(*n*)	Нет

4.4.3. Хеш-таблица: в среднем время постоянно, если порядок неважен

Хеш-таблицы и хеширование описываются в приложении B. Хеш-таблицы часто используются для реализации ассоциативных массивов, предназначенных для хранения значений из очень широких множеств (например, из множества всех возможных строк или множества всех целых чисел), но только когда должно храниться ограниченное их количество. В таких случаях используется хеш-функция, отображающая множество значений (*домен*, или исходное множество) в меньшее множество с M элементами (*содомен*, или целевое множество), с индексами простого массива, где хранятся значения, связанные с каждым ключом. (Как объясняется в приложении B, мы можем выбирать размер M, исходя из некоторых соображений ожидаемой производительности.) Обычно значения из домена называют *ключами*, а значения в содомене являются индексами в диапазоне от 0 до $M - 1$.

Поскольку целевое множество хеш-функции обычно меньше исходного, неизбежны конфликты: по крайней мере, два значения будут отображены в один и тот же индекс. Хеш-таблицы, как было показано в главе 2, используют несколько стратегий для разрешения конфликтов, таких как *цепочки* или *открытая адресация*.

Еще одна важная деталь, о которой следует помнить: различаются между собой хеш-отображения (hash maps) и хеш-множества (hash sets). Первые позволяют связать значение[1] с ключом, вторые позволяют только определить наличие или отсутствие ключа в множестве. Хеш-множества реализуют особый случай словаря — тип множества `Set`. Согласно нашему определению словаря как абстрактной структуры данных, приведенному в начале этого раздела, *множество* — это особая разновидность словаря, в котором значения имеют логический тип и, как следствие, второй параметр метода `insert` становится избыточным, так как предполагается, что с ключом в хеш-множестве неявно связано значение `true`.

Абстрактная структура данных: множество (Set)	

API	`class Set {`
	`  insert(key)`
	`  remove(key)`
	`  contains(key) → true/ false`
	`}`

Контракт с клиентом	Множество хранит набор ключей. Если ключ `K` был добавлен в множество (и не удален впоследствии), то `contains(K)` вернет `true`, иначе он вернет `false`

[1] Не путайте с индексами, сгенерированными с помощью хеш-функции.

Как объясняется в приложении B, все операции с хеш-таблицей (и хеш-множеством) могут выполняться за амортизированное время `O(1)`.

4.4.4. Двоичное дерево поиска: логарифмическое время выполнения всех операций

Двоичные деревья поиска (Binary Search Tree, BST) нам уже знакомы — мы встречались с ними в главах 2 и 3, и они также обсуждаются в приложении B.

BST — это особый вид двоичного дерева, способный хранить ключи, которые поддерживают полное упорядочение, то есть любые два ключа можно сравнить и определить, равны ли они, и если нет, узнать, какой из них меньше. Полная упорядоченность получает дополнительный выигрыш от свойств *рефлексивности*, *симметричности* и *транзитивности*.

ОТНОШЕНИЯ ПОРЯДКА

Для множества S, на котором определено отношение упорядочения $\leq$, это отношение является полным упорядочением, если для любых трех ключей x, y, z выполняются следующие свойства.

Рефлексивность: $x \leq x$.

Антисимметричность: если $x \leq y$, то $y \geq x$.

Полнота: либо $x \leq y$, либо $y \leq x$.

Транзитивность: если $x \leq y$ и $y \leq z$, то $x \leq z$.

Эти свойства позволяют двоичным деревьям поиска гарантировать, что положение ключа в дереве можно определить, просто просмотрев единственный путь от корня к листу.

Фактически, вставляя новый ключ, мы сравниваем его с корнем дерева. Если он меньше, поворачиваем налево и идем в левое поддерево; иначе поворачиваем направо и идем в правое поддерево. Затем вновь сравниваем ключ с корнем поддерева и продолжаем в том же духе, пока не дойдем до листа — того места, куда нужно вставить ключ.

Напоминаю, что в главе 2 говорилось о кучах (или в приложении B, если вы читали его): все операции в двоичном дереве поиска выполняются за время, пропорциональное высоте дерева (самый длинный путь от корня до листа). В частности, в сбалансированных двоичных деревьях поиска все операции выполняются за время $O(\ln(n))$, где n — количество ключей в дереве.

Конечно, по сравнению с амортизированным временем $O(1)$ работы хеш-таблиц даже сбалансированные двоичные деревья поиска кажутся не лучшим выбором для реализации ассоциативного массива. Но дело в том, что, несмотря на *небольшую* потерю в производительности основных методов, двоичные деревья поиска позволяют существенно улучшить производительность таких операций, как поиск

предшествующего и последующего ключей, поиск минимума и максимума: все они выполняются за асимптотическое время O(ln(*n*)), тогда как те же операции с хеш-таблицами требуют времени O(*n*).

Более того, двоичные деревья поиска могут вернуть все хранимые ключи (или значения), отсортированные по ключу, за линейное время, тогда как при использовании хеш-таблиц придется отсортировать набор ключей после его извлечения, соответственно, потребуется O(*M* + *n* * ln(*n*)) сравнений.

Теперь, познакомившись с основными структурами данных, наиболее часто используемыми для реализации словарей, обобщим некоторые аспекты. В табл. 4.3 показано время выполнения основных операций разными реализациями словарей, упоминавшимися выше.

Таблица 4.3. Время выполнения операций разными реализациями словарей

Операция	Несортированные массивы	Сортированные массивы	BST	Хеш-таблицы
Создание	O(1)	O(*n* log(*n*))	O(*n* log(*n*))	O(*n*)
Поиск элемента	O(*n*)	O(log(*n*))	O(log(*n*))	O(*n* / *M*)*
Добавление элемента	O(1)*	O(*n*)	O(log(*n*))	O(*n* / *M*)*
Удаление элемента	O(1)	O(*n*)	O(log(*n*))	O(*n* / *M*)*
Сортировка списка	O(*n* log(*n*))	O(*n*)	O(*n*)	O(*M* + *n* log(*n*))
Поиск минимального/максимального	O(*n*)	O(1)	O(1)**	O(*M* + *n*)
Поиск предыдущего/последующего	O(*n*)	O(1)	O(log(*n*))	O(*M* + *n*)

* Амортизированное время.

** При сохранении максимального/минимального значений (и амортизированное время для их замены при вставке/удалении).

Глядя на табл. 4.3, нетрудно догадаться, что, если не требуется заботиться о выполнении операций с элементами в определенном порядке, то амортизированное время хеш-таблиц является наилучшим. Если *n* ~= *M* (и существует приблизительно столько же сегментов, сколько и элементов), то хеш-таблица может выполнять вставку, удаление и поиск за амортизированное постоянное время.

4.4.5. Фильтр Блума: так же быстр, как хеш-таблицы, при этом экономит память (но есть подвох)

Эта структура данных в книге еще не обсуждалась, но вполне возможно, что вы уже слышали ней: имеется в виду *фильтр Блума*. Такое название выбрано в честь Бартона Ховарда Блума (Burton Howard Bloom), который изобрел ее в 1970-х годах.

Между хеш-таблицами и фильтрами Блума есть четыре заметных различия:

- простые фильтры Блума не хранят данных; они просто определяют присутствие *данного значения* в наборе. Другими словами, они реализуют API *хеш-множества*, а не хеш-таблицы;
- фильтры Блума требуют меньше памяти, чем хеш-таблицы, и это основная причина их использования;
- отрицательные ответы всегда точные, но положительные ответы иногда могут оказаться ошибочными. Мы подробно обсудим эту черту в нескольких последующих разделах, а пока просто имейте в виду, что иногда фильтр Блума может ответить, что в него было добавлено некоторое значение, хотя на самом деле это не так (так называемый ложноположительный ответ);
- значение нельзя удалить из фильтра Блума[1].

Существует компромисс между точностью фильтра Блума и объемом используемой памяти. Чем меньше памяти, тем выше вероятность получить ложноположительный ответ. К счастью, существует точная формула, позволяющая вывести объем памяти, необходимый для удержания частоты ложноположительных ответов в определенных пределах, исходя из количества значений, которые нужно хранить. Подробно она будет рассмотрена в дополнительных разделах ближе к концу главы.

4.5. ЗА КУЛИСАМИ: КАК РАБОТАЮТ ФИЛЬТРЫ БЛУМА

Теперь углубимся в детали реализации. Фильтр Блума состоит из двух элементов:

- массива из m элементов;
- набора из k хеш-функций.

Массив (концептуально) представлен набором битов, каждый из которых изначально имеет значение 0, а каждая хеш-функция возвращает индекс от 0 до $m - 1$.

Крайне важно сразу уточнить, что между элементами массива и ключами, которые добавляются в фильтр Блума, нет однозначного соответствия. Вместо этого будем использовать k бит (и, следовательно, k элементов массива) для хранения каждого элемента; k здесь обычно намного меньше, чем m.

Обратите внимание, что k — это константа, которая выбирается при создании структуры данных, поэтому каждый добавляемый элемент хранится в памяти одного и того же объема, равного k бит. В отношении строковых значений такое положение вещей кажется удивительным, потому что означает, что в фильтр можно добавлять строки произвольной длины, выделяя для каждой один и тот же объем памяти k бит.

---

[1] По крайней мере, не в простой версии. Как будет показано далее, некоторые реализации поддерживают удаление элементов.

При добавлении нового ключа фильтр вычисляет k индексов для массива, заданного значениями от $h_0(key)$ до $h_{(k-1)}(key)$, и устанавливает эти биты в 1.

При поиске элемента для него вычисляется k хешей, как изложено в описании метода `insert`, но на этот раз проверяются k бит по индексам, возвращаемым хеш-функциями, и результат `true` возвращается тогда и только тогда, когда все биты в этих позициях имеют значение 1.

На рис. 4.3 показано, как действуют эти операции.

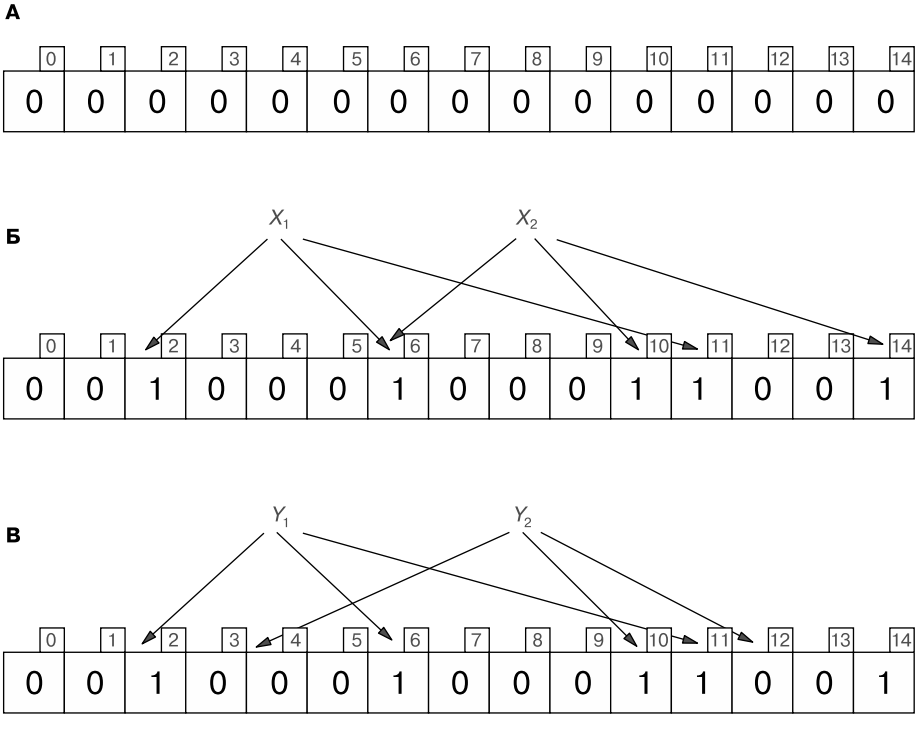


Рис. 4.3. Пример фильтра Блума. *А*. Изначально фильтр имеет вид массива нулей. *Б*. Чтобы сохранить элемент X_i, он хешируется k раз (в примере k = 3), при этом каждый хеш дает индекс бита; после чего все эти k бит устанавливаются в 1. Обратите внимание, что две тройки индексов, сгенерированных для элементов X_1 и X_2, частично перекрываются (для обоих одна из хеш-функций вернула индекс шестого элемента). Более подробный пример работы операции вставки приводится на рис. 4.5. *В*. Аналогично, чтобы проверить присутствие элемента Y_i в наборе, его хешируют k раз и для каждого полученного индекса проверяют соответствующие им биты, и только если все они установлены равными 1, возвращается true. Судя по нижнему рисунку, элемент Y_1 находится в наборе (хотя нельзя исключать вероятность ложноположительного ответа), а элемент Y_2 отсутствует в наборе, потому что один из элементов с индексом, сгенерированным одной из хеш-функций, равен 0. Более подробный пример работы операции приводится на рис. 4.4

В идеале нужны k различных независимых хеш-функций, чтобы для одного и того же значения не повторялись никакие два индекса. Нелегко спроектировать большое количество независимых хеш-функций, но можно получить довольно хорошие приближения, используя несколько решений (см. перечень ниже).

- Применение параметрической хеш-функции `H(i)`. Эта метафункция генерирует хеш-функции. Она принимает начальное значение `i` и возвращает хеш-функцию H_i=`H(i)`. На этапе инициализации фильтра Блума можно создать `k` таких функций, от H_0 до H_{k-1}, вызывая генератор `H` для `k` различных (и обычно случайных) значений.
- Применение одной хеш-функции `H`, но с инициализацией списка `L` из k случайных (и уникальных) значений. Для каждого вставляемого/искомого ключа создается k значений путем добавления `L[i]` к ключу, после чего результаты хешируются с помощью `H`. (Как мы знаем, хорошо спроектированные хеш-функции дают очень разные результаты даже для мало различающихся входных данных.)
- Применение двойного или тройного хеширования[1].

Последнее не гарантирует независимости между сгенерированными хеш-функциями, но было доказано[2], что это ограничение можно ослабить за счет минимального увеличения вероятности получить ложноположительный ответ. Для простоты в нашей реализации используется двойное хеширование с двумя независимыми хеш-функциями: *Murmur-хеширование*[3] и *хеширование Фаулера-Нола-Во*[4] (fnv1).

В общем случае наша i-я хеш-функция для i между 0 и k – 1 будет иметь следующий вид:

```
h_i(key) = murmurhash(key) + i * fnv1(key) + i * i
```

4.6. РЕАЛИЗАЦИЯ

Достаточно теории; пришло время заняться практикой! Как и ранее, в последующих разделах рассмотрим фрагменты псевдокода и исследуем его ключевые моменты. Тривиальные методы будут опущены. В репозитории книги на GitHub[5] можно найти полный код реализаций вместе с модульными тестами.

---

1 Метод двойного хеширования часто используется для разрешения конфликтов хешей (как описывается в приложении B, в подразделах B2.2–B2.3). Когда возникает конфликт, к начальной позиции ключа добавляется смещение, вычисленное с помощью вторичного хеширования. Тройное хеширование вычисляет смещение, используя линейную комбинацию двух вспомогательных хеш-функций.

2 *Dillinger P. C., Manolios P.* Fast and accurate bitstate verification for SPIN. International SPIN Workshop on Model Checking of Software. — Berlin; Heidelberg: Springer, 2004.

3 https://sites.google.com/site/murmurhash/.

4 http://www.isthe.com/chongo/tech/comp/fnv/index.html.

5 https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#bloom-filter.

4.6.1. Использование фильтра Блума

Вернемся к примеру обработки контактов: можно ли применять фильтр Блума, чтобы ускорить работу с контактами? Как уже упоминалось, необходимо использовать список подобно словарю, поэтому создадим новый фильтр Блума, получим список контактов с сервера и добавим их в фильтр Блума. В листинге 4.1 показана обобщенная реализация этого процесса инициализации.

Листинг 4.1. Процесс начальной инициализации клиента электронной почты

```
function initBloomFilter(server, minSize)
  contactsList ← server.loadContacts()
  size ← max(2 * |contactsList|, minSize)
  bloomFilter ← new BloomFilter(size)
  for contact in contactsList do
    bloomFilter.insert(contact)
  return bloomFilter
```

Метод initBloomFilter принимает объект server, обслуживающий операции с сервером, и минимальный размер фильтра Блума; возвращает вновь созданный фильтр Блума

При запуске загружается список контактов с сервера

Размер фильтра Блума должен быть по меньшей мере в два раза больше текущего списка контактов и не меньше minSize — минимального значения, переданного в аргументе

Создаем пустой фильтр Блума выбранного размера

Обходим список контактов

Добавляем каждый контакт в фильтр Блума

Кроме настройки приложения, немалый интерес представляют две операции: проверка наличия контакта в списке и добавление нового контакта.

При выполнении первой операции (листинг 4.2) проверяется фильтр Блума, и если выяснится, что контакт никогда не добавлялся, то можно с уверенностью утверждать, что контакт отсутствует в системе. Однако если фильтр Блума возвращает `true`, то это может быть ложным срабатыванием, тогда нужно связаться с сервером для перепроверки.

Листинг 4.2. Проверка адреса электронной почты

```
function checkContact(bloomFilter, server, contact)
  if bloomFilter.contains(contact) then
    return server.contains(contact)
  else
    return false
```

Метод checkContact проверяет присутствие адреса электронной почты в списке контактов. В проверке участвуют фильтр Блума, объект связи с сервером и адрес электронной почты. Возвращается true, если адрес уже присутствует в списке контактов

Проверяем присутствие адреса в списке контактов с помощью фильтра Блума

Если фильтр Блума вернул true, нужно проверить присутствие адреса в списке контактов, обратившись к серверу, потому что ответ фильтра может быть ложноположительным

Иначе, поскольку фильтр Блума всегда точен в отрицательных ответах (он может ошибаться только в положительных ответах), можно уверенно возвращать false

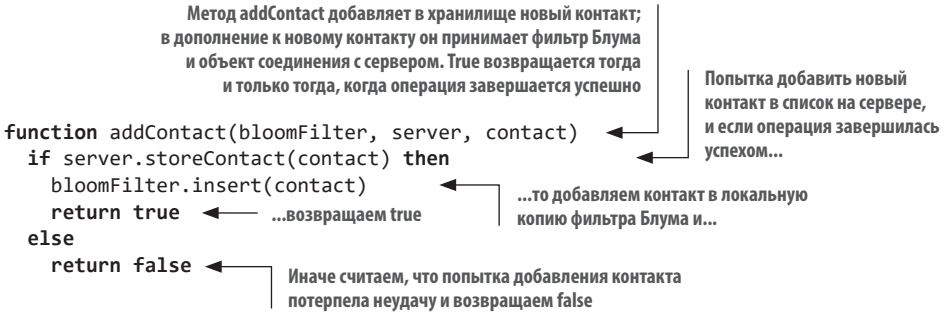
Для добавления новых контактов всегда необходимо выполнять синхронизацию с хранилищем[1], как показано в листинге 4.3. Поскольку это, возможно, подразуме-

[1] Можно, конечно, проверить локальный фильтр Блума на наличие контакта, но, даже если он возвращает истину, без проверки на стороне сервера у нас не будет уверенности в том, что результат не является ложноположительным!

вает удаленное соединение через сеть, существует немалая вероятность того, что обращение к серверу завершится неудачей; поэтому прежде, чем обновлять фильтр Блума, нужно обработать возможные сбои и убедиться, что удаленный вызов прошел успешно.

Для большей точности в реальных реализациях также следует синхронизировать доступ к серверу и фильтру Блума, используя механизм блокировки (см. главу 7) и заключать всю операцию в блок try-catch, откатывая (или повторяя попытку) если вызов был неудачным.

Листинг 4.3. Добавление нового контакта



4.6.2. Чтение и запись битов

Теперь перейдем к реализации фильтра Блума и, как обычно, начнем со вспомогательных методов — основных строительных блоков, необходимых для реализации API.

В частности, нам понадобятся:

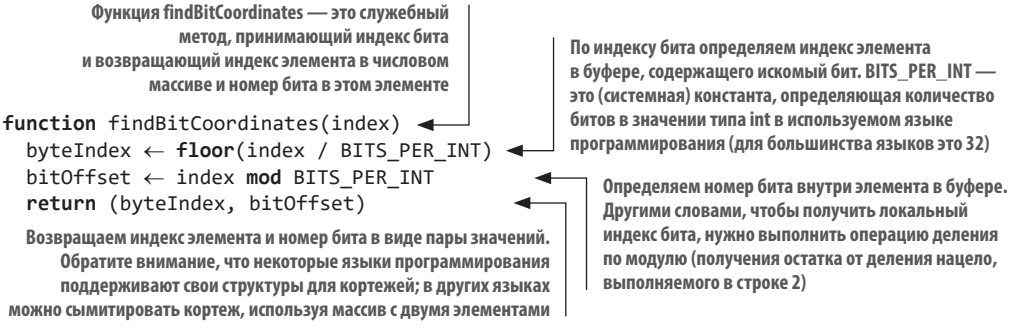
- возможность читать и писать биты в любом месте буфера нашего фильтра;
- возможность отображать ключи в индексы битов в буфере;
- набор детерминированно сгенерированных хеш-функций для преобразования ключей в список индексов.

Напомню, что фильтры Блума используются для экономии памяти, поэтому хранить биты неэффективно, это противоречит логике. Необходимо упаковать биты в наименьший целочисленный тип, доступный в выбранном языке программирования, и соответственно для чтения и записи битов реализовать отображение индекса требуемого бита в пару целых чисел.

На самом деле в современных языках программирования эти операции можно ускорить, используя числовые массивы фиксированного размера и векторную алгебру. При этом, получив запрос на доступ к i-му биту в фильтре, следует по индексу i определить две координаты: номер элемента в массиве, где хранится i-й бит, и номер бита в этом элементе.

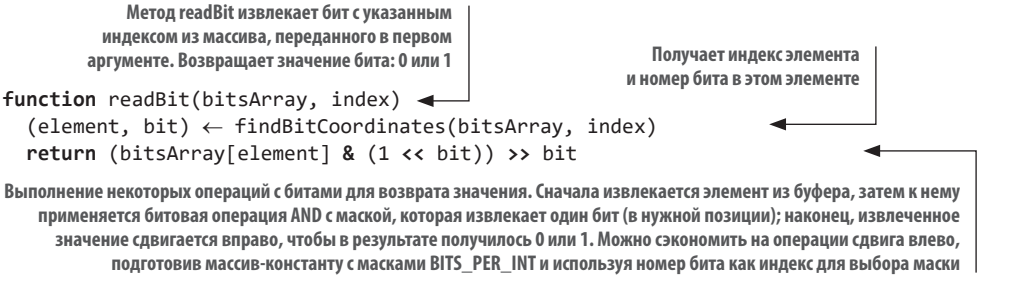
В листинге 4.4 показано, как можно вычислить эти координаты.

Листинг 4.4. findBitCoordinates



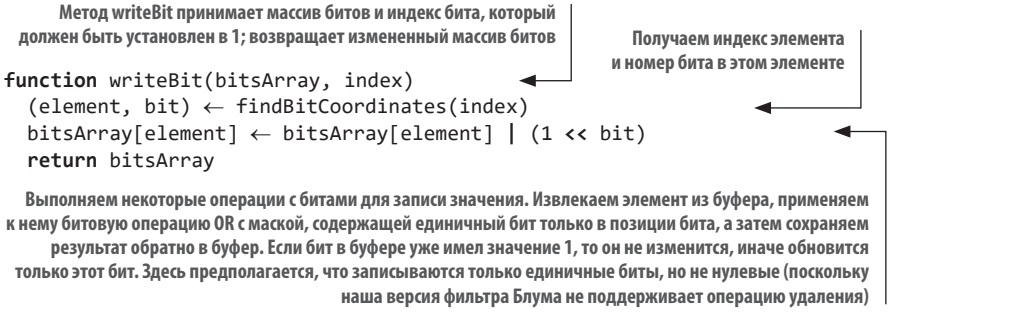
Получив эти два индекса, можно прочитать или записать любой бит; для этого достаточно прибегнуть к операциям битовой арифметики. Например, в листинге 4.5 показан метод `readBit`, отвечающий за чтение.

Листинг 4.5. Метод readBit



В листинге 4.6 показан аналогичный метод записи `writeBit`. Возможно, вас удивит, что методу не передается значение записываемого бита, но, учитывая, что фильтр Блума (по крайней мере, эта его версия) не поддерживает удаление элементов, записываться может только значение 1.

Листинг 4.6. Метод writeBit



Посмотрим, как работают методы `readBit` и `writeBit`. Предположим, у нас есть такой буфер: `B=[157, 25, 44, 204]` и `BITS_PER_INT=8`.

Вызываем `readBit(B, 19)` и получаем, что искомый бит находится в: `element==2`, `bit==3`.

Следовательно:

- `bitsArray[element]` (даст число `44`);
- `(1 << bit)` (`8`);
- `bitsArray[element] & (1 << bit)` (`8`).

И метод `readBit` вернет `1`.

Выполняя вызов `writeBit(B, 15)`, получаем: `element==1`, `bit==7`.

Следовательно:

- `bitsArray[element]` (даст число `25`);
- `(1 << bit)` (`128`);
- `bitsArray[element] | (1 << bit)` (`153`).

И буфер изменится так: `B=[157, 153, 44, 204]`.

4.6.3. Поиск места, где хранится ключ

Чтобы получить индексы всех битов, используемых для хранения ключа, нужно выполнить двухэтапный процесс, описанный в листинге 4.7.

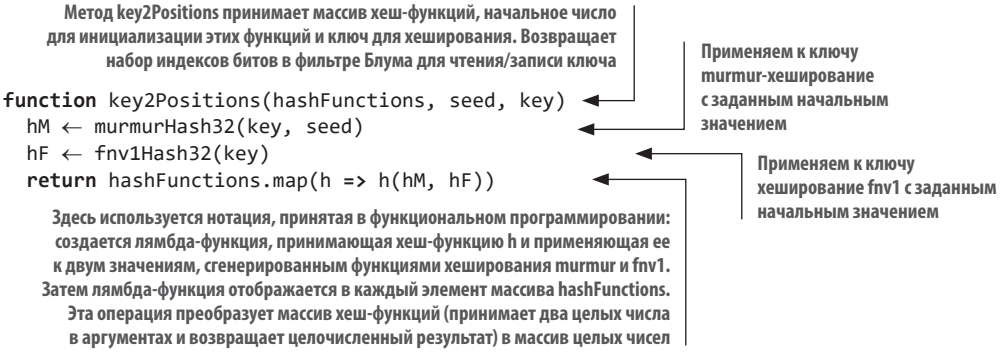
Имейте в виду, что наша конечная цель — преобразовать строку в k позиций, между 0 и $m - 1$.

Прежде всего будем использовать две хеш-функции для строк, сильно отличающиеся друг от друга: функцию *murmur-хеширования* и функцию хеширования *fnv1*. Вероятность, что для данной строки обе дадут один и тот же результат, чрезвычайно мала.

Для каждой из k позиций получаем соответствующую хеш-функцию из нашего пула. Для каждой позиции i между 0 и $k - 1$ генерируем (при инициализации) функцию двойного хеширования h_i. Таким образом, для i-го бита будет получена функция $h_i(h_M, h_F)$, где h_M — результат murmur-хеширования входного ключа, а h_F — результат хеширования fnv1.

Наивысший уровень рандомизации можно было бы получить, выбирая случайное начальное значение в каждом прогоне, но для нас важнее детерминированное поведение, чтобы иметь возможность тестировать или воссоздавать фильтр Блума из сериализованного представления для поддержки быстрого перезапуска в случае сбоя. Следовательно, необходимо предусмотреть возможность передачи начального значения конструктору фильтра Блума.

Листинг 4.7. Метод key2Positions

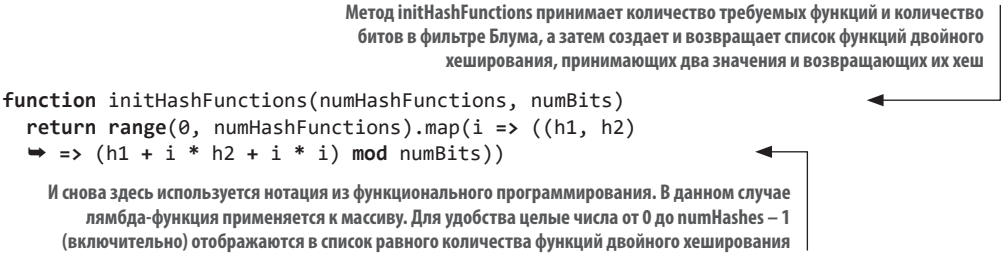


4.6.4. Генерирование хеш-функций

В листинге 4.7 отмечено, что при вызове метода `key2Positions` ему передается массив хеш-функций, и затем он используется для преобразования ключа в список индексов: позиций в битовом массиве фильтра, где хранится ключ. Теперь пришло время посмотреть, как инициализируются эти k хеш-функций (листинг 4.8), необходимых для отображения каждого ключа (уже преобразованного в строку) в набор k индексов, определяющих биты с информацией о ключе (хранимом или нет).

Набор функций создается с использованием двойного хеширования для объединения двух аргументов k различными способами. По сравнению с линейным или квадратичным хешированием, двойное хеширование увеличит количество возможных хеш-функций с $O(k)$ до $O(k^2)$. Конечно, это количество далеко от идеала $O(k!)$, гарантированного равномерным хешированием, но на практике оно достаточно близко к желаемому (что означает низкий уровень конфликтов).

Листинг 4.8. Метод initHashFunctions



4.6.5. Конструктор

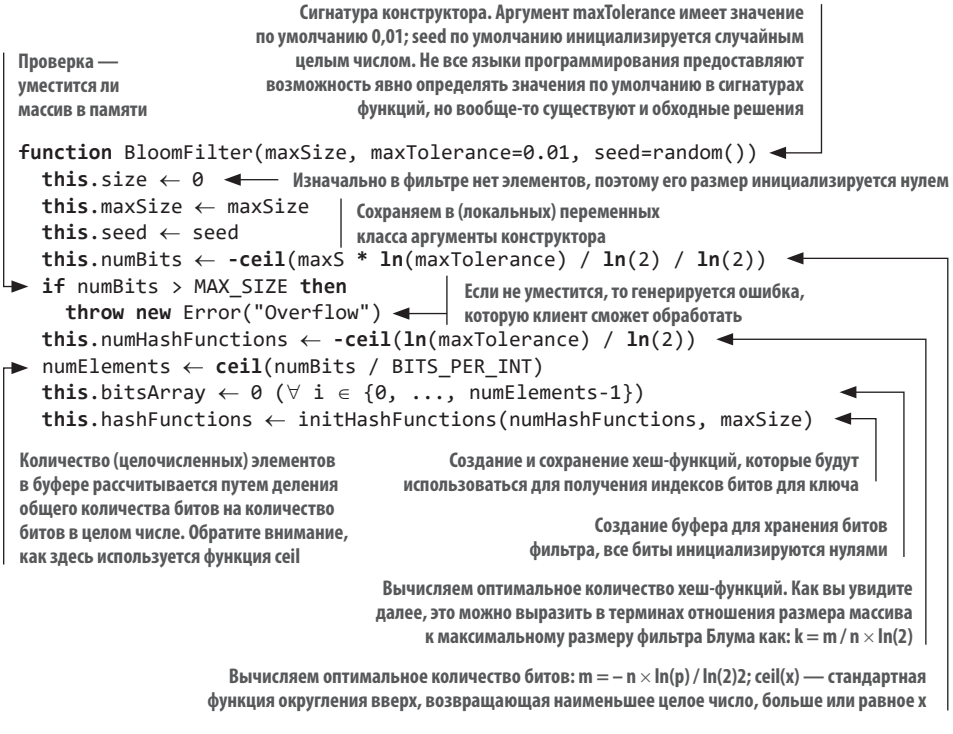
Теперь перейдем к общедоступному API, отражающему API множества `set`, который был определен в подразделе 4.3.3. Начнем с конструктора.

Как это часто бывает, конструктор содержит в основном шаблонный код для настройки внутреннего состояния фильтра Блума. В этом случае, однако, выполняются

некоторые нетривиальные действия, связанные с вычислением объемов ресурсов. Эти операции нужно выделить, чтобы контейнер соответствовал точности, требуемой клиентом.

В листинге 4.9 показана возможная реализация конструктора. Обратите внимание, в частности, на строки 5 и 8, где вычисляются количество битов и количество хеш-функций соответственно. Они необходимы, чтобы оценить желаемую долю ложноположительных результатов в пределах допуска, заданного параметром `maxTolerance`, и, соответственно, определить количество элементов массива (строка 9), необходимых для хранения фильтра. Здесь предполагается, что массив состоит из целых чисел, а `BITS_PER_INT` — это системная переменная, определяющая размер целых чисел в битах. Очевидно, что для языков, поддерживающих несколько целочисленных типов, можно использовать массивы байтов, если они доступны.

Листинг 4.9. Конструктор фильтра Блума



На этапе создания достаточно знать только максимальное количество элементов, которое, как ожидается, будет содержать фильтр. Если в какой-то момент мы поймем, что сохранили в фильтре больше элементов, чем `maxSize`, то будем знать, что места хватит, и это хорошо, но при этом мы не сможем гарантировать ожидаемую точность, и это плохо.

Учитывая сказанное выше, можно передать необязательный второй параметр, чтобы гарантировать ожидаемую точность. По умолчанию порог вероятности

ложноположительных результатов (`maxTolerance`) установлен равным 1 %, но можно добиться большей точности, передав меньшее значение, или согласиться на худшую точность в обмен на меньшее потребление памяти.

Последний необязательный параметр необходим, как объяснялось в предыдущем разделе, чтобы обеспечить детерминированное поведение фильтра. Если вызывающая сторона опустит его, то будет выбрано случайное начальное значение.

После проверки полученных аргументов (в листинге 4.9 проверки опущены) выполняется настройка базовых полей. Затем наступает самая сложная часть: исходя из заданного количества элементов и ожидаемой точности вычисляется размер буфера. Здесь используется формула, описанная в разделе 4.10, но нужно убедиться, что размер буфера сможет разместиться в отведенной памяти.

Получив размер буфера, вычисляем оптимальное количество хешей, необходимых для поддержания уровня ложноположительных результатов на минимально возможном уровне.

4.6.6. Проверка ключа

Теперь можно приступать к вспомогательным операциям, которые используются методами API фильтра Блума. Одно важное замечание: ключи, как предполагается, будут строками, но они также могут быть сериализуемыми объектами. В таком случае понадобится функция сериализации, превращающая эквивалентные объекты (например, два множества, содержащие одни и те же элементы) в одну и ту же строку; иначе, независимо от того, насколько хорошей получится реализация фильтра Блума (или любого другого словаря), приложение будет работать с ошибками.

ПРИМЕЧАНИЕ

Предварительная обработка данных часто не менее важна, чем основные алгоритмы.

Благодаря уже написанным вспомогательным функциям проверить наличие ключа в фильтре Блума становится проще простого. Достаточно получить позиции битов, соответствующих ключу, и проверить их на равенство 1. Чтобы оценить весь процесс в целом, взгляните на рис. 4.4 и на псевдокод в листинге 4.10.

Возможно, вы обратили внимание, что `contains` проверяет неравенство нулю значения, возвращаемого функцией `readBit`. Технически надо было бы проверить равенство бита единице, но это вынудило бы использовать дополнительную битовую операцию сдвига вправо. Если бит хранится в *i*-м элементе массива как его *j*-й бит (считая справа), то, по сути, достаточно сдвинуть *i*-е целое число, извлеченное из буфера, вправо на *j* бит или сравнить его с маской, состоящей из одной единицы, сдвинутой на *j* позиций влево. В данной реализации этого не требуется и можно сэкономить несколько миллисекунд (на каждой операции).

Также обратите внимание, что метод принимает необязательный второй параметр. Причина станет понятна в следующем разделе.

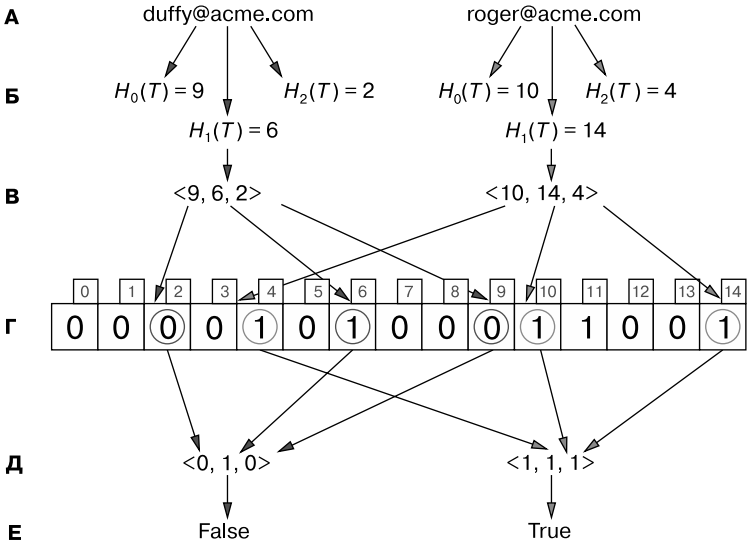


Рис. 4.4. Пошаговая проверка наличия элемента в фильтре Блума. *А*. У нас есть адрес электронной почты, который нужно проверить, — duffy@acme.com. *Б*. Ключ (адрес электронной почты) обрабатывается с помощью набора хеш-функций. В этом примере предполагается, что k = 3, поэтому ключ будет обрабатываться тремя разными хеш-функциями: H_0, H_1 и H_2. *В*. Каждая хеш-функция возвращает индекс в массиве битов, например в данном случае: <9, 6, 2>. *Г*. Мы извлекаем элементы битового массива с этими индексами. *Д*. Первый элемент с индексом 2 равен 0. Остальные биты равны 0 и 1 соответственно. *Е*. Не все проверенные биты равны 1, это означает, что адрес duffy@acme.com отсутствует в фильтре Блума и можно смело вернуть false. Те же действия выполняются для адреса roger@acme.com (показан справа). Поскольку все три бита равны 1, возвращаем true. Это означает, что roger@acme.com мог быть добавлен в фильтр Блума с определенной степенью достоверности

Листинг 4.10. Метод contains

Функция contains принимает ключ и возвращает true тогда и только тогда, когда все биты, соответствующие ключу, равны 1. При необходимости можно явно передать массив позиций битов для проверки. Некоторые операции с фильтром требуют многократного доступа к битам, и передача массива позволяет сэкономить на повторяющихся вычислениях. В языках, поддерживающих приватные методы и перегрузку, передачу второго параметра следует разрешать только во внутренних методах

Проверяем, был ли передан массив позиций. Если да, то позиции повторно не вычисляются

Получить позиции (индексы) битов, соответствующие текущему ключу

```
function contains(key, positions=null)
  if positions == null then
    positions ← key2Positions(this.hashFunctions, this.seed, key)
  return positions.all((i) => readBit(this.bitsArray, i) != 0)
```

Возвращает true тогда и только тогда, когда все проверенные биты не равны 0. Здесь снова используется нотация из функционального программирования применительно к методу all, который действует подобно map, за исключением того, что принимает предикат (конкретную лямбда-функцию, которая возвращает логическое значение), применяет его к каждому элементу в списке и возвращает true, только если предикат вернул true для всех элементов

4.6.7. Сохранение ключа

Сохранение ключа очень похоже на его проверку, нужно лишь приложить чуть больше усилий, чтобы подготовить элементы для добавления в фильтр, и использовать запись вместо чтения. На рис. 4.5 показан пошаговый пример, объединяющий все фрагменты кода, представленные до сих пор.

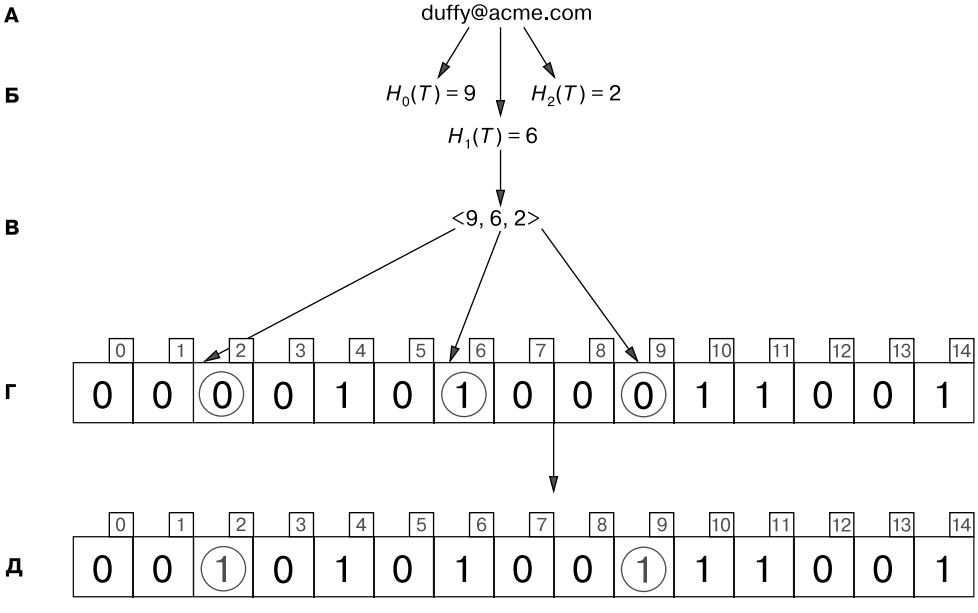
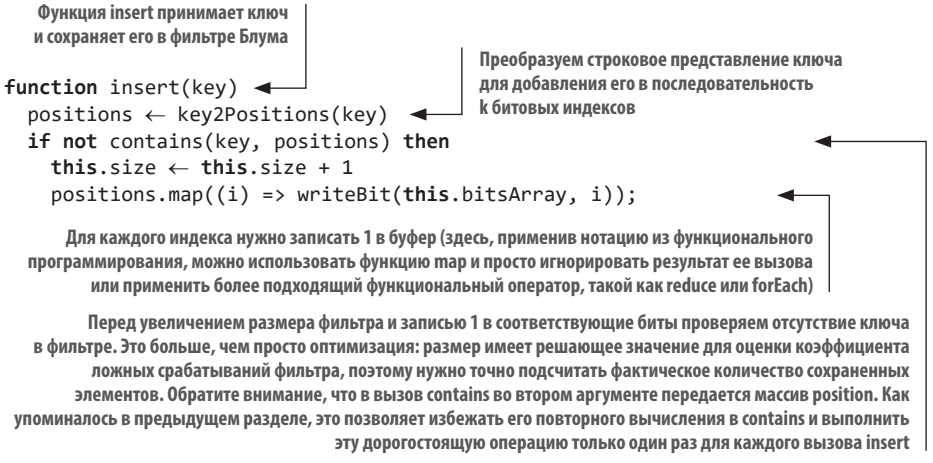


Рис. 4.5. Пошаговое добавление нового элемента в фильтр Блума. *А*. У нас есть адрес электронной почты, который нужно сохранить: duffy@acme.com. *Б*. Ключ (адрес электронной почты) обрабатывается с помощью набора хеш-функций. В этом примере предполагается, что k = 3, поэтому ключ будет обрабатываться тремя разными хеш-функциями: H_0, H_1 и H_2. *В*. Каждая хеш-функция возвращает индекс в массиве битов, например в данном случае: <9, 6, 2>. Обратите внимание, что эти индексы совпадают с показанными на рис. 4.4: хеш-функции действуют детерминированно, даже притом что их результаты могут показаться случайными. *Г*. Извлекаем элементы битового массива с этими индексами. Первый элемент с индексом 2 равен 0. Остальные биты равны 0 и 1 соответственно. *Д*. Биты, прежде равные 0, инвертируются (в этом примере биты с индексами 2 и 9). Теперь все биты, соответствующие хешу адреса duffy@acme.com, установлены равными 1, и любые будущие попытки поиска этого адреса будут давать значение true

Обратите внимание, что в этой реализации метода `insert` (листинг 4.11) при вычислении размера фильтра определяется количество уникальных элементов, добавленных в фильтр, а не общее количество вызовов метода `add`.

Это связано с тем, что точность фильтра не меняется при добавлении одного и того же ключа дважды, трижды или любое другое число раз. Во всех случаях будет считаться, что ключ добавлен один раз.

Листинг 4.11. Метод insert




```
function insert(key)
  positions ← key2Positions(key)
  if not contains(key, positions) then
    this.size ← this.size + 1
    positions.map((i) => writeBit(this.bitsArray, i));
```

Однако есть одна неувязочка. Если добавить новый уникальный ключ `x`, для которого индексы всех битов конфликтуют с местоположениями, уже установленными в 1, он будет рассматриваться как повторно добавленный ключ и размер структуры данных не увеличится. Впрочем, это не особенно важно, если учесть, что в такой ситуации и до фактического добавления нового конфликтующего ключа вызов `contains(x)` вернул бы ложноположительный результат.

В листинге 4.11 также видно, почему, создавая метод `contains`, мы добавили возможность передачи ему массива с уже вычисленными индексами битов для текущего ключа. Внутри `insert` выполняется чтение, за которым следует запись. Операция вычисления индексов битов ключа может быть довольно дорогостоящей, поэтому, чтобы избежать повторного ее выполнения, был предусмотрен способ передачи ее результата в `contains`. При этом добавлять эту возможность в API не стоит, потому что это внутренняя магия, о которой клиентам знать не обязательно. Если ваш язык программирования поддерживает полиморфизм и приватные методы, то было бы разумно ограничить доступность необязательного параметра приватной версией `contains` (которая вызывается общедоступной).

Другой возможный подход к экономии на повторяющихся вычислениях — проверка в `writeBit` равенства единице каждого устанавливаемого бита и возврат логического значения, указывающего, изменилось ли значение бита. В таком случае `insert` может проверить, был ли изменен хотя бы один бит. Эта альтернативная реализация, как и описанная выше, представлена в репозитории[1]. Выбирайте ту из них, которая кажется вам наиболее ясной.

В любом случае попытка подсчета уникальных ключей, добавленных в фильтр, обходится дорого. Оправдаются ли накладные расходы? Если предполагается, что

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction/blob/master/JavaScript/src/bloom_filter/bloom_filter.js.

в фильтр будет добавлено немного повторяющихся ключей, то, вероятно, этих накладных расходов лучше избежать. Но имейте в виду, что вам обязательно понадобится точная оценка текущей вероятности получения ложноположительного результата.

4.6.8. Оценка точности

Наша последняя задача — реализовать метод оценки вероятности получить ложноположительный результат на основе текущего состояния фильтра. То есть искомая вероятность должна зависеть от количества элементов, хранящихся в данный момент в фильтре, и его максимальной емкости.

Как будет показано в разделе 4.10, эта вероятность примерно равна[1]:

$$p = \left(1 - e^{-\frac{\text{числоХешей} \cdot \text{размер}}{\text{числоБитов}}}\right)^{\text{числоХешей}}.$$

В листинге 4.12 показан код метода, вычисляющего эту оценку.

Листинг 4.12. Метод falsePositiveProbability

```
function falsePositiveProbability()
  return pow((1 - pow(E, this.numHashes * this.size / this.numBits)),
  ➥ this.numHashes)
```

На этом завершим раздел, посвященный реализации фильтра Блума. Но прежде, чем углубимся в теорию, объясняющую математическую основу этой структуры данных, рассмотрим некоторые из ее основных применений.

4.7. ПРИМЕНЕНИЕ

Предлагаю вам подумать вот о чем: вполне вероятно, что какое-то программное обеспечение, которое вы применяете прямо сейчас, использует фильтры Блума. На самом деле, если вы читаете цифровую версию этой книги, то вы можете быть на 100 % уверены, что для ее загрузки использовались фильтры Блума, потому что интернет-узлы часто используют их для реализации своих таблиц маршрутизации[2].

4.7.1. Кэш

Для веб-приложений серьезной проблемой и, вероятно, одним из самых обсуждаемых аспектов в обзорах любого продукта, претендующего на популярность, является масштабируемость.

---

[1] Здесь e — это число Эйлера; мы еще встретимся с ним в разделе 4.10.

[2] Фильтры Блума также уже довольно давно используются в браузерах для «безопасного серфинга», они поддерживают черный список вредоносных сайтов. Например, движок Chromium только несколько лет назад заменил его сжатыми множествами префиксов PrefixSet.

В одной из моих любимых книг, посвященных проектированию и масштабированию, *Scalability Rules* (Abbot M. L., Fisher M. T., Addison-Wesley Professional, 2016) есть глава под названием Use Caching Aggressively («Максимально используйте кэширование»), где показано, что для масштабирования веб-приложения фундаментальное значение имеет кэширование на всех уровнях.

Под *кэшированием* подразумевается процесс хранения некоторой полученной информации в быстрой системе хранения А, чтобы при необходимости ее можно было прочитать в ближайшем будущем. Данные могли быть ранее извлечены из более медленной системы хранения Б или получены в результате сложных и продолжительных вычислений.

Кэш часто является единственным механизмом, который буквально спасает базы данных от «возгорания» или по крайней мере от сбоя. Даже ваш ноутбук имеет несколько уровней кэширования: от быстрого кэша L1 внутри процессора до кэша в памяти, используемого для обработки больших файлов. Сама операционная система кэширует страницы памяти, выгружая и загружая их с диска, чтобы получить виртуальное адресное пространство большого объема и создать у пользователя впечатление, что она обладает бóльшим объемом доступной памяти, чем на самом деле имеется в компьютере.

Другими словами, кэш — одна из основ современных информационных систем. Конечно, быстрое хранилище стоит дороже, поэтому его объем ограничен и бóльшую часть данных приходится из кэша удалять.

Алгоритм, выбирающий, какие данные оставить в кэше, определяет поведение кэша — частоту попаданий в кэш (когда искомые данные уже находятся в кэше) и частоту промахов. В числе наиболее известных подобных алгоритмов можно назвать *LRU* (Least Recently Used — «наиболее давно использовавшиеся» (данные)), *MRU* (Most Recently Used — «наиболее недавно использовавшиеся») и *LFU* (Least Frequently Used — наименее часто использовавшиеся). Эти алгоритмы подробно рассматриваются в главе 7, а пока достаточно отметить, что все они, как и многие другие политики вытеснения данных из кэша, страдают проблемой «однодневок». Другими словами, они борются с объектами, фрагментами памяти или веб-страницами, запрашиваемыми только один раз и никогда больше (за время существования кэша). Это особенно характерно для маршрутизаторов и *сетей доставки контента* (Content Delivery Networks, CDN), где в среднем 75 % запросов возвращают однократно используемые данные.

Применение словаря для отслеживания запросов дает возможность сохранять объект в кэше, только когда он запрашивается вторично, и отбрасывать однократно используемые данные, чтобы тем самым улучшить *коэффициент попаданий в кэш*. Фильтр Блума позволяет выполнять поиск таких данных за постоянное амортизированное время и использовать ограниченный объем памяти, при этом он считает допустимыми не особенно болезненные ложные срабатывания. Единственным отрицательным эффектом ложных срабатываний будет лишь незначительное снижение той эффективности кэша, которую мы ожидаем от использования фильтра Блума (так что никакого вреда от этого не будет).

4.7.2. Маршрутизаторы

Современные маршрутизаторы имеют ограниченный объем памяти, и, учитывая количество пакетов, которое они обрабатывают каждую секунду, им нужны чрезвычайно быстрые алгоритмы. Соответственно, они идеально подходят для применения фильтров Блума во всех операциях, где допустима небольшая доля ошибок.

Помимо кэширования, маршрутизаторы часто используют фильтры Блума для отслеживания запрещенных IP-адресов и ведения статистики, помогающей выявлять DoS-атаки (Denial of Service — «отказ в обслуживании»)[1].

4.7.3. Интернет-роботы

Интернет-роботы (crawler) — это автоматизированные программные агенты, сканирующие сеть (и даже всю Всемирную паутину), отыскивающие контент, анализирующие и индексирующие все, что им попадется.

Когда интернет-робот находит ссылки на странице или в документе, он обычно следует им и рекурсивно сканирует документы, на которые они указывают. Есть некоторые исключения: например, поисковые роботы игнорируют большинство типов файлов, равно как и ссылки, созданные с помощью тегов `<a>` с атрибутом `rel="nofollow"`.

СОВЕТ

На самом деле рекомендуется помечать таким образом любые якорные теги со ссылками на действия, имеющие побочные эффекты. В противном случае поисковые роботы, даже соблюдающие эту политику, будут вести себя непредсказуемо.

Если вы напишете собственный поисковый робот, не проявив должной осторожности, он может оказаться в бесконечном цикле между двумя или более страницами со взаимными ссылками (или цепочками ссылок) друг на друга.

Чтобы избежать зацикливания, поисковые роботы должны запоминать, какие страницы они уже посетили. Фильтры Блума лучше всего подходят для этого, потому что способны компактно хранить ссылки URL и выполнять проверку и сохранение этих URL за постоянное время.

Цена за ложные срабатывания здесь немного выше, чем в предыдущих примерах, потому что в случае ошибки поисковый робот никогда не посетит URL, вызвавший ложное срабатывание.

Для предотвращения этой ситуации можно сохранять полный список посещенных URL-адресов в словаре (или в другой коллекции) на диске, и только если фильтр Блума возвращает значение `true`, перепроверять ответ в словаре. Этот подход не дает экономии памяти, но обеспечивает некоторую экономию времени выполнения, если среди URL высок процент адресов, встречающихся один раз.

---

[1] *Geneiatakis D., Vrakas N., Lambrinoudakis C.* Utilizing Bloom filters for detecting flooding attacks against SIP based services // Computers & Security. — 2009. — Vol. 28.7 — P. 578–591.

4.7.4. Сборщик операций ввода/вывода

Еще одна область, где можно использовать кэширование на основе фильтра Блума, — уменьшение количества избыточных и дорогостоящих операций ввода/вывода. Механика работы та же, что и при сканировании сети: операция выполняется только тогда, когда имеет место «промах», а «попадания» обычно вызывают более глубокое сравнение (например, извлечение с диска только первых нескольких строк или блоков из документа и их сравнение при попадании).

4.7.5. Проверка орфографии

Существуют простые версии средств проверки орфографии, основанные на применении фильтров Блума в качестве словарей. По результатам поиска в фильтре Блума для каждого слова в тексте можно подтвердить его правильность или установить признак орфографической ошибки. Конечно, ложноположительные события могут привести к тому, что некоторые орфографические ошибки останутся незамеченными, но вероятность такого события можно контролировать. Однако современные средства проверки орфографии в основном используют *префиксные деревья* (trie): эти структуры данных обеспечивают хорошую производительность при поиске в тексте без ложных срабатываний.

4.7.6. Распределенные базы данных и файловые системы

Cassandra использует фильтры Блума для сканирования индекса, чтобы определить, есть ли в SSTable данные, соответствующие конкретной записи.

Аналогично Apache HBase использует фильтры Блума как эффективный механизм проверки наличия в StoreFile определенной записи или ячейки. Это помогает увеличить общую скорость чтения за счет фильтрации избыточных операций чтения с диска блоков HFile, которые не содержат искомой записи или ячейки.

Мы подошли к концу нашего экскурса в практику применения фильтров Блума. Разве что стоит еще упомянуть такие области их использования, как ограничители скорости, черные списки, ускорение синхронизации и оценка размера объединений в базах данных.

4.8. КАК РАБОТАЮТ ФИЛЬТРЫ БЛУМА[1]

До сих пор мы исходили из того, что фильтры Блума работают так, как было описано. Теперь пришло время углубиться в теорию и объяснить, как же на самом деле они работают. Этот раздел не является обязательным и не требуется для реализации или использования фильтров Блума, но знакомство с теоретическими выкладками поможет вам глубже понять рассматриваемую структуру данных.

[1] В этом и последующих разделах приводится большой объем теории и представлена расширенная концептуальная база.

Как уже упоминалось, фильтры Блума — это компромисс между памятью и точностью. Если вы собираетесь создать экземпляр фильтра Блума, выделив для него 8 бит, а затем попытаться сохранить в нем 1 миллион объектов, то велика вероятность, что никакого прироста производительности не получится. Напротив, все биты в фильтре Блума с 8-битным буфером будут установлены равными 1 примерно после 10–20 операций хеширования. После этого все вызовы `contains` будут возвращать `true`, и вы не сможете понять, был ли объект в действительности сохранен в контейнере. На рис. 4.6 показан пример такой ситуации.

В то же время, если выделить достаточно места и подобрать хорошие хеш-функции, то индексы, сгенерированные для разных ключей, не станут конфликтовать между собой и для любых двух разных ключей перекрытие между списками сгенерированных индексов будет минимальным, если вообще будет.

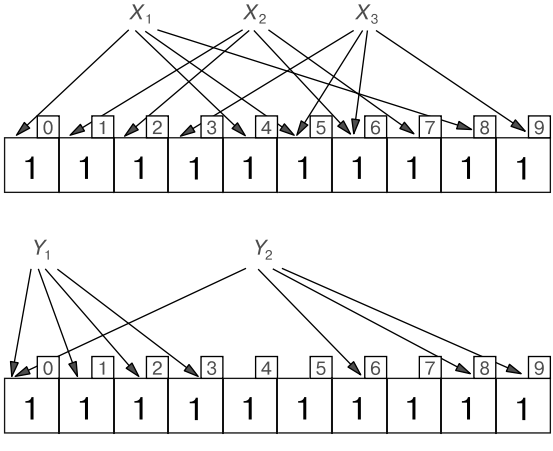


Рис. 4.6. Насыщение фильтра Блума. В примере m = 10 соответственно фильтр Блума имеет буфер с размером 10 бит, а k = 4, поэтому каждый элемент хранится в 4 битах. Этот фильтр можно «насытить», добавив три элемента и установив все биты равными 1. В таком случае, даже если и будут найдены Y_1 и Y_2, которые прежде не были добавлены в фильтр, результат все равно будет ложноположительным. Конечно, это пример крайнего случая (с нереально большим значением k для значения m), но он наглядно иллюстрирует, как происходит насыщение, приводящее к ухудшению точности

Но какой же объем буфера можно считать достаточным? Внутренние операции фильтра Блума преобразуют каждый ключ в последовательность k индексов, выбранных из m возможных альтернатив[1]: фокус в том, что мы эффективно сохраняем ключи, устанавливая в 1 эти k бит в буфере фильтра.

[1] Здесь m — это размер буфера фильтра.

Те из вас, кто еще помнит школьный курс алгебры, вероятно, уже догадались, что с помощью различных k элементов, взятых из множества m, можно представить только m^k различных последовательностей. Однако сделаю два уточнения.

1. На самом деле нет возможности использовать все эти последовательности. Для нас желательно, чтобы все индексы, связанные с ключом, были разными (иначе для хранения ключа будет отводиться меньше чем k бит), поэтому мы стремимся к тому, чтобы все списки из k индексов не содержали дубликатов.
2. Нас не интересует порядок этих k индексов. Совершенно неважно, запишем ли мы сначала бит с индексом 0, а затем бит с индексом 3 или наоборот. Соответственно, мы можем считать последовательности индексов множествами.

Учитывая сказанное, мы считаем допустимыми (по крайней мере теоретически) только часть всех возможных множеств k (различных) индексов, взятых из диапазона 0... m – 1, и количество всех этих возможных (допустимых) множеств задается формулой:

$$\binom{m}{k} = \frac{m!}{k!(m-k)!}.$$

Биномиальный коэффициент[1] в этой формуле выражает количество способов, которыми можно извлечь k (уникальных) элементов из множества размером m. В нем отражено количество множеств с k различными индексами, которые могут вернуть k хеш-функций.

Если требуется, чтобы каждый ключ отображался в другое множество k индексов, то, зная k и n (количество хранимых ключей), можно по формуле, приведенной выше, вычислить нижнюю границу для m (размера буфера).

С другой стороны, используя последовательность из m бит (буфер), можно представить только 2^m различных значений, то есть в каждый конкретный момент времени фильтр Блума может находиться только в одном из 2^m возможных состояний. Однако это не слишком точная (хотя и простая для вычисления) оценка границы n; она не учитывает, что на каждый ключ приходится k бит (2^m становится точной оценкой, только для k == 1).

В разделе 4.10 будет показано, как выбрать количество хеш-функций и размер массива, чтобы сделать оптимальным количество ложноположительных результатов проверок для фильтра Блума, способного хранить заданное количество ключей.

---

[1] Биномиальный коэффициент здесь — это m по k; https://ru.wikipedia.org/wiki/Биномиальный_коэффициент.

4.8.1. Почему невозможны ложноотрицательные результаты проверок...

Простая версия фильтра Блума не допускает возможности удаления. Это означает, что после установки в 1 при сохранении ключа бит никогда больше не будет сброшен в 0.

В то же время результат, возвращаемый хеш-функциями, детерминирован и не меняется с течением времени.

СОВЕТ

Обратите внимание на эти свойства и соблюдайте их в своей реализации, если у вас есть необходимость сериализовать фильтры Блума и десериализовать их позже.

Итак, если в процессе поиска обнаруживается, что один из битов, связанных с ключом *X*, равен 0, то можно с уверенностью утверждать, что *X* никогда не добавлялся в фильтр; иначе все биты, в которые хешируется *X*, были бы равны 1.

4.8.2. ...Но ложноположительные возможны

Обратное, однако, невозможно, к сожалению! Рассмотрим пример, чтобы понять почему. Предположим, имеется простой фильтр Блума с буфером на 4 бита и двумя хеш-функциями. Изначально буфер пуст:

```
B = [0, 0, 0, 0]
```

Затем в фильтр Блума вставляется значение `1`. Допустим, были выбраны такие хеш-функции, что h_0`(1) = 0` и h_1`(1) = 2`, поэтому ключ `1` отображается в индексы `{0,2}`. После обновления буфер теперь выглядит так:

```
B = [1, 0, 1, 0]
```

Затем вставляется значение `2`, для которого h_0`(2) = 1` и h_1`(2) = 2`. В результате получаем:

```
B = [1, 1, 1, 0]
```

Наконец, пусть h_0`(3) = 1` и h_1`(3) = 0`. Если выполнить проверку для значения `3` после этих двух вставок, оба бита с индексами 1 и 0 будут установлены равными 1, даже притом что значение `3` никогда не сохранялось в фильтре! Следовательно, проверка ключа `3` даст ложноположительный результат.

В то же время, если бы хеш-функции действовали иначе, например h_0`(3) = 3` и h_1`(3) = 0`, то, учитывая, что четвертый бит еще не был установлен (`B[3]== 0`), поиск вернул бы `false`.

Конечно, это упрощенный пример, специально созданный, чтобы доказать состоятельность нашей точки зрения: ложноположительные результаты возможны и будут тем более вероятны, чем менее тщательно будут выбираться параметры фильтра Блума. В разделе 4.10 я покажу, как настраивать эти параметры в зависимости от ожидаемого количества элементов, которые должны храниться в фильтре, и требуемой точности.

4.8.3. Фильтры Блума как рандомизированные алгоритмы

В приложении Е приводится классификация рандомизированных алгоритмов, и в частности дихотомия между алгоритмами *Лас-Вегас* и *Монте-Карло*.

Если у вас нет четкого представления о различиях между этими двумя классами или о рандомизированных алгоритмах в целом, то сейчас самое время отвлечься на знакомство с приложением Е.

Когда эти определения станут ясны, вы без труда определите, к какой категории относятся фильтры Блума.

А возможно, вы уже поняли, что фильтры Блума — это структуры данных Монте-Карло. Метод `contains`, алгоритм проверки присутствия ключа в фильтре Блума, является, в частности, алгоритмом *со смещением в ложную сторону* (false-biased). На самом деле он может возвращать `true` для некоторых ключей, которые никогда не добавлялись в фильтр, но всегда верно возвращает `true`, если ключ был добавлен ранее, поэтому ложноотрицательных результатов проверок не бывает (то есть каждый раз, когда алгоритм отвечает значением `false`, можно быть уверенными, что ответ правильный).

ПРИМЕЧАНИЕ

Фильтры Блума также являются компромиссом между памятью и точностью. Детерминированная версия фильтра Блума — это *хеш-множество*.

4.9. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ

Прежде чем приступить к анализу производительности фильтра Блума, я предлагаю изучить метрики для оценки алгоритмов классификации в приложении Е.

Вы уже знаете, как и почему работают фильтры Блума; теперь посмотрим, насколько они эффективны. Сначала оценим время выполнения наиболее важных операций с фильтром Блума, а затем, в разделе 4.10, решим задачу оценки точности метода `contains` для фильтра Блума с определенной структурой (а именно, будут учитываться размер буфера и количество используемых хеш-функций).

4.9.1. Время выполнения

Уже отмечалось, что фильтры Блума могут сохранять и отыскивать ключи за постоянное время. Технически это верно только для входных данных постоянной длины; здесь мы рассмотрим наиболее общий случай, когда ключи — это строки произвольной длины[1].

[1] Любой объект или значение можно преобразовать в строку (например, в строку нулей и единиц).

Но начнем с самого начала: с конструирования фильтра Блума, а затем перейдем к методам `insert` и `contains`.

4.9.2. Конструктор

Сконструировать фильтр Блума просто; нужно лишь инициализировать массив битов, сброшенных в 0, и сгенерировать набор из k хеш-функций. Известно, что реализация конструктора включает также некоторые вычисления, но их можно смело рассматривать как выполняемые за постоянное время.

Создание и инициализация массива занимают время $O(m)$, а создание каждой из хеш-функций — постоянное время; следовательно, для создания всего набора необходимо время $O(k)$.

Весь этап конструирования в целом потребует выполнить $O(m + k)$ шагов.

4.9.3. Сохранение элемента

Для любого сохраняемого ключа нужно создать k хеш-значений, определяющих индексы элементов массива битов, и установить соответствующие биты в 1.

Сделаем следующие предположения:

- для установки одного бита требуется постоянное время (включая, возможно, время выполнения битовых операций, если для экономии места используются сжатые буферы);
- хеширование ключа X требует времени $T(X)$;
- количество битов, используемых для хранения ключа, не зависит ни от размера ключа, ни от количества элементов, уже добавленных в контейнер.

С учетом этих предположений получим, что время выполнения `insert(X)` равно $O(k \times T(|X|))$. Фактически нужно выполнить каждую из k хеш-функций, сгенерировать хеш-значение из ключа и сохранить один бит. Для числовых ключей (целых или вещественных) $|X| = 1$ и $T(|X|) = O(1)$, то есть хеш-значение генерируется за постоянное время.

Однако если ключи имеют переменную длину (например, роль ключей играют строки), то для вычисления каждого хеш-значения требуется время, линейно пропорциональное длине строки. В этом случае $T(|X|) = O(|X|)$ и время выполнения будет зависеть от длины добавляемого ключа.

Теперь предположим, что самый длинный ключ будет состоять не более чем из z символов, где z — константа. Исходя из сделанного предположения о том, что длина ключа не зависит ни от чего другого, можно утверждать, что время выполнения операции `insert(X)` равно $O(k(1 + z)) = O(k)$, то есть это операция с постоянным временем выполнения, независимо от количества элементов, уже добавленных в контейнер.

4.9.4. Поиск элемента

Все изложенное верно и для поиска ключа: нужно преобразовать ключи в набор индексов (за время $O(z \times k)$), а затем проверить каждый бит по этим индексам (суммарное время $O(k)$). Таким образом, при ограничении длины ключей константой поиск тоже становится операцией с постоянным временем выполнения.

4.10. ОЦЕНКА ТОЧНОСТИ ФИЛЬТРА БЛУМА[1]

Прежде чем начать, нужно уточнить некоторые обозначения и сделать еще несколько предположений:

- m — количество битов в массиве;
- k — количество хеш-функций, используемых для отображения ключа в k различных позиций в массиве;
- все k хеш-функций независимы друг от друга;
- пул хеш-функций, откуда извлекаются k функций, является универсальным хеш-множеством.

Можно доказать, что при выполнении этих предположений хорошим приближением вероятности ложноположительного результата после n вставок может служить следующее выражение:

$$p(n, m, k) = \left(1 - e^{-\frac{kn}{m}}\right)^k,$$

где e — число Эйлера, основание натурального логарифма.

Теперь у нас есть формула оценки вероятности получения ложноположительного результата! Это хорошо само по себе, но еще лучше, что, используя эту формулу, можно подобрать оптимальные параметры фильтра Блума m и k — размер буфера и количество хеш-функций, необходимых для достижения оптимальной точности.

В формуле для $p(n, m, k)$ имеется три переменные:

- m — количество битов в буфере;
- n — количество элементов, которые будут храниться в контейнере;
- k — количество хеш-функций.

Из этих трех переменных k кажется наименее значимой. Или, говоря иными словами, она меньше остальных связана с нашей задачей. Значение переменной n,

[1] Этот раздел включает математические выкладки повышенной сложности.

похоже, легко оценить, но нельзя полностью контролировать. Вполне вероятно возникновение ситуации, когда понадобится сохранить в фильтре столько элементов, сколько будет получено запросов. Однако в большинстве случаев объем запросов предсказуем, а из него для пущей безопасности возможно вывести пессимистичную оценку.

Выбор значения m тоже может быть ограничен из-за ограниченного объема доступной памяти; скажем, нам нельзя использовать больше чем m битов.

А вот в отношении k нет никаких ограничений, и эту переменную можно настроить для получения оптимальной точности, что означает, как объясняется в приложении Е, минимальную вероятность ложноположительных результатов.

К счастью, рассчитать оптимальное значение k при заданных m и n не очень сложно — нужно просто найти минимум функции:

$$f(k) = \left(1 - e^{-\frac{kn}{m}}\right)^k.$$

Обратите внимание, что n и m в этой формуле являются константами.

Мы подробно рассмотрим поиск минимума для $f(k)$ в следующем разделе, а пока (если вас больше интересует результат) просто имейте в виду, что оптимальное значение равно:

$$k^* = \frac{m}{n}\ln(2).$$

Теперь, имея формулу для k^*, подставим ее в предыдущую формулу для $p(n, m, k)$ и после некоторых алгебраических манипуляций получим выражение оптимального значения m (назовем его m^*):

$$m^* = -n\frac{\ln(p)}{\ln(2)^2}.$$

Это означает, что, если заранее известно общее количество n уникальных элементов, которые будут добавлены в контейнер, и задана предельная вероятность p ложноположительного результата, то можно вычислить оптимальный размер буфера (и количество битов на ключ), чтобы гарантировать желаемую точность.

Рассматривая полученную формулу, обратите внимание на два аспекта:

- размер буфера фильтра Блума пропорционален количеству вставляемых элементов;
- в то же время необходимое количество хеш-функций зависит только от целевой вероятности ложноположительного результата p (в этом можно убедиться, подставив m^* в формулу для k^*, но не будем торопиться, математическая основа будет рассматриваться в следующем разделе).

4.10.1. Объяснение формулы отношения ложноположительных результатов

В этом подразделе подробнее обсудим формулы оценки точности фильтра Блума и для начала проанализируем, как рассчитывается оценка вероятности получения ложноположительных результатов.

После сохранения одного бита в фильтре Блума емкостью m битов вероятность того, что конкретный бит установлен равным 1, составляет $1/m$; тогда вероятность, что тот же самый бит установлен равным 0 после установки всех k бит, используемых для хранения элемента (при условии, что хеш-функции всегда дают k различных значений для одного и того же входного значения[1]), составит:

$$\left(1-\frac{1}{m}\right)^{k}.$$

Если рассматривать событие установки любого конкретного бита равным 1 как независимое, то после вставки n элементов вероятность, что каждый отдельный бит в буфере останется равным 0, определяется как:

$$p_{\text{bit}}=\left(1-\frac{1}{m}\right)^{kn}\approx e^{-\frac{kn}{m}}.$$

Чтобы возник ложноположительный результат, все k бит, соответствующих элементу V, должны быть независимо установлены в 1, и вероятность, что все эти k бит равны 1, определяется выражением:

$$p(n,m,k)=\left(1-p_{\text{bit}}\right)^{k}=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}\approx\left(1-e^{-\frac{kn}{m}}\right)^{k}.$$

А это, что самое интересное, и есть формула вероятности, которая была дана в начале текущего раздела.

Здесь, как упоминалось в предыдущем разделе, n и m можно рассматривать как константы. В этом есть определенный смысл, потому что во многих случаях заранее известно, сколько элементов придется добавить в фильтр Блума *(n)* и сколько битов памяти можно выделить для буфера *(m)*. Для нас желательно было бы отдать предпочтение точности, пожертвовав производительностью, настроив k, количество (универсальных) хеш-функций.

Это эквивалентно поиску глобального минимума функции f, определяемой как:

$$f(k)=\left(1-e^{-\frac{kn}{m}}\right)^{k}.$$

[1] Другими словами, как было показано в предыдущих разделах, мы предполагаем, что хеш-функции взяты из множества универсальных хеш-функций.

Те, кто знаком с предметом, вероятно, уже догадались, что нужно вычислить производные от f по k. (Если же вы не из числа этих счастливчиков, не волнуйтесь: вы можете пропустить следующие несколько абзацев и продолжить чтение с того места, где описывается использование результата этого вычисления.)

Чтобы не усложнять, перепишем f, применив натуральный логарифм и возведение в степень[1]:

$$f(k) = \left(1 - e^{-\frac{kn}{m}}\right)^k = e^{k\ln\left(1-e^{-\frac{kn}{m}}\right)}.$$

Эта функция дает минимальное значение, когда экспонента минимальна, отсюда можно определить функцию g:

$$g(k) = k\ln\left(1 - e^{-\frac{kn}{m}}\right)$$

и вычислить производные g, что намного проще.

Первая производная $g(k)$:

$$g'(k) = \frac{\partial g}{\partial k} = \ln\left(1 - e^{-\frac{kn}{m}}\right) + \frac{kn}{m}\frac{-e^{-\frac{kn}{m}}}{1 - e^{-\frac{kn}{m}}},$$

она равна 0 при $k = \ln(2) \times m/n$.

Чтобы убедиться, что это действительно минимум функции, нужно вычислить вторую производную и проверить, возвращает ли она отрицательное значение в точке, где g' равна *нулю*:

$$g''\left(\ln(2)\frac{m}{n}\right) < 0.$$

Я пропущу этот шаг для экономии места, но вы можете перепроверить и убедиться, что это действительно так.

Стоит отметить, что:

- формула для k дает единственное точное значение оптимального количества хеш-функций;
- k, очевидно, должно быть целым числом, поэтому результат следует округлить;
- чем больше значение k, тем хуже производительность операций вставки и поиска (поскольку для каждого элемента необходимо вычислить больше хеш-функций), поэтому выбор для k немного меньшего значения может оказаться неплохим компромиссом.

---

[1] e^x и $\ln(x)$ — обратные друг к другу функции, соответственно, при $x > 0$ выполняется условие $\ln(e^x) = e^{\ln(x)} = x$, даже если x является функцией (если, конечно, она всегда положительна).

При использовании оптимального значения k, вычисленного по формулам выше, частота ложноположительных результатов f становится равной:

$$f = \left(\frac{1}{2}\right)^k = (0{,}6185)^{\frac{m}{n}}.$$

Заменив значение k в формуле для $p(n, m, k)$, получим новую формулу, связывающую количество битов в буфере с (максимальным) количеством элементов, которые можно сохранить, независимо от k (значение k позволительно вычислить позже), чтобы гарантировать вероятность ложноположительного результата меньше определенного значения p:

$$p = p(n, m) = \left(1 - e^{-\left(\frac{m}{n}\ln(2)\frac{n}{m}\right)}\right)^{\frac{m}{n}\ln(2)} = \left(1 - e^{-\ln(2)}\right)^{\frac{m}{n}\ln(2)} = \left(\frac{1}{2}\right)^{\frac{m}{n}\ln(2)}.$$

Если взять логарифм по основанию 2 от обеих сторон:

$$\log_2(p) = \log_2\left[\left(\frac{1}{2}\right)^{\frac{m}{n}\ln(2)}\right] = -\frac{m}{n}\ln(2),$$

тогда решение для m дает:

$$m^* = -n\frac{\log_2(p)}{\ln(2)} = -n\frac{\ln(p)}{\ln(2)^2}.$$

Это означает, что, если заранее известно общее количество n уникальных элементов, которые будут добавлены в фильтр Блума, и p определяет максимально допустимую вероятность ложноположительного результата, то становится реальным вычислить оптимальный размер буфера, гарантирующий желаемую точность. Нужно также установить соответствующее значение k, используя выведенную формулу:

$$k^* = \ln(2)\frac{m^*}{n} = -\ln(2)n\frac{\ln(p)}{\ln(2)^2}\frac{1}{n} = -\frac{\ln(p)}{\ln(2)}.$$

Например, чтобы получить точность 90 % и, следовательно, не более 10 % ложноположительных результатов, подставим эти числа в формулу и получим:

$$k = -\frac{\ln(0{,}1)}{\ln(2)} = -\frac{-2{,}3025}{0{,}6931} = 3{,}32;$$

$$m = -n\frac{\ln(p)}{\ln(2)^2} = 4{,}792n.$$

4.11. УЛУЧШЕННЫЕ ВАРИАНТЫ

Фильтры Блума существуют уже почти 50 лет, и, естественно, за это время было предложено множество их вариантов и улучшений. Рассмотрим некоторые из них, сосредоточившись на тех, которые повышают точность.

4.11.1. Фильтр Блумьера

Как уже упоминалось, фильтры Блума являются более быстрыми и легковесными аналогами хеш-множества HashSet, они сохраняют только информацию о наличии/отсутствии ключа.

Совсем недавно был представлен компактный аналог хеш-таблиц HashTable: *фильтр Блумьера,* позволяющий связывать значения с ключами. После сохранения пары «ключ — значение» в фильтре Блумьера[1] для нее всегда возвращается правильный результат. Ложноположительные результаты все еще возможны; то есть для некоторых ключей, несмотря на то что они фактически не были сохранены, значения могут возвращаться.

4.11.2. Комбинирование фильтров Блума

Сохраняя один и тот же ключ в двух или более разных фильтрах Блума, например с разными размерами буферов и, что особенно важно, с разными наборами хеш-функций, можно уменьшить вероятность ложноположительных результатов.

Конечно, за это приходится платить, потому что увеличиваются объем потребляемой памяти и время, необходимое для сохранения или проверки ключа.

Однако не все так плохо: фильтры могут обрабатываться параллельно на многоядерном оборудовании! Таким образом, помимо поддержания постоянной временной границы O(k), реальные реализации могут действовать так же быстро, как обычные фильтры Блума (другими словами, постоянный коэффициент практически не меняется).

Эта структура работает следующим образом: `insert` вставляет ключ в разные фильтры независимо и параллельно, а `contains` возвращает комбинацию всех ответов — `true` возвращается только в том случае, если все фильтры вернули `true`.

Какова точность такого массива фильтров? Можно доказать, что один фильтр Блума с буфером m бит имеет такую же точность, как j фильтров Блума, использующих буферы по m / j бит каждый.

Однако при использовании параллельной версии алгоритма время работы может составить лишь долю исходного, $1/j$.

---

[1] *Chazelle B., Kilian J., Rubinfeld R., Tal A.* The Bloomier filter: An efficient data structure for static support lookup tables // Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (PDF). — 2004. — P. 30–39.

4.11.3. Многослойный фильтр Блума

Многослойный фильтр Блума[1] (Layered Bloom Filter, LBF) тоже основан на использовании нескольких фильтров Блума, организованных в многослойную структуру. Каждый слой обновляется при добавлении ключа, только если предыдущий слой уже сохранил этот ключ. Многослойные фильтры Блума обычно используются для реализации счетного фильтра: LBF с R слоями может подсчитывать до R вставок одного и того же ключа. Часто также поддерживается и удаление.

Каждый вызов `contains` проверяет слои, начиная с самого глубокого, и возвращает индекс последнего слоя, где был найден ключ, или –1 (что эквивалентно значению `false`), если ключ не найден ни в одном слое.

При сохранении ключа метод `insert` сохраняет его в первом слое, для которого `contains` вернет `false`.

Если предположить, что каждый слой имеет коэффициент ложноположительных результатов, равный P_F, и все слои используют разные наборы хеш-функций, тогда, если элемент был сохранен в фильтре c раз:

- вероятность, что `contains` вернет $c + 1$ — количество попыток сохранить ключ на единицу больше реального количества, — приблизительно равна P_F;
- вероятность возврата $c + 2$ равна P_F^2;
- аналогично вероятность возврата $c + d$ равна P_F^d.

Однако это приблизительные (и оптимистичные) оценки, потому что допущение об универсальности и независимости хеш-функций трудно гарантировать. Чтобы вычислить точные вероятности, нужно принять во внимание глубину и количество слоев.

В LBF с L слоями, каждый из которых использует k бит на ключ, время выполнения `insert` и `contains` становится равным $O(L \times k)$, но, поскольку L и k являются предопределенными константами, оно по-прежнему эквивалентно $O(1)$ и не зависит от количества ключей, добавленных в контейнер.

4.11.4. Сжатый фильтр Блума

Основная проблема фильтра Блума при использовании в веб-кэшах связана не с объемом используемой оперативной памяти, а с размером данных, передаваемых по сети, потому что эти фильтры приходится передавать между прокси-серверами.

В первый момент кажется, что этот вопрос размера является спорным. Но если предполагается сжимать фильтры Блума перед их передачей по сети, то он действительно приобретает практическую ценность. Как оказывается, можно оптимизировать

[1] *Zhiwang Cen, Jungang Xu, Jian Sun* (2010). A multi-layer Bloom filter for duplicated URL detection, Proc. 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE 2010). № 1. — P. V1–586–V1–591.

параметры фильтра, которые, в свою очередь, регулируют размер массива битов, и получить несжатый фильтр Блума большего размера, а затем его сжимать более эффективно.

В этом заключается идея сжатых фильтров Блума[1], в которых количество хеш-функций k выбирается так, чтобы число битов со значением 1 оставалось меньше $m/3$, где m — размер массива. Как следствие, в массиве 2/3 бит всегда будут установлены равными 0, что и нужно использовать для более эффективного сжатия.

Тогда каждый прокси-сервер должен будет распаковать фильтр Блума перед поиском элементов. Очевидно, здесь появляется еще одна цель для оптимизации. Надо найти компромисс между несжатым размером битового массива m и его сжатым размером, который желательно минимизировать.

Размер несжатого фильтра фактически определяет время поиска (согласно формулам, представленным в разделе 4.10), а размер сжатого фильтра — объем данных, передаваемых по сети.

Поскольку таблицы маршрутизатора периодически обновляются, распаковка всего фильтра может потребовать значительных накладных расходов на узлах. Хорошим компромиссом могут оказаться деление фильтра на части и их сжатие независимо друг от друга. Общая степень сжатия немного ухудшится, но при частых обновлениях (по сравнению с поиском) это будет способствовать снижению накладных расходов на распаковку, потому что каждый прокси-сервер, получив обновление, будет распаковывать не весь битовый массив, а только его часть.

4.11.5. Масштабируемый фильтр Блума

Масштабируемый фильтр Блума — еще одна комбинация нескольких фильтров Блума, работающая аналогично многослойному фильтру. Только здесь емкость каждого следующего слоя больше емкости предыдущего, благодаря чему уменьшается количество ложноположительных результатов. Это позволяет контейнеру динамически адаптироваться к количеству хранимых элементов и снижает вероятность ложных срабатываний.

РЕЗЮМЕ

- Для выполнения определенных операций можно выбрать наилучшую абстрактную структуру данных (Abstract Data Structure, ADT) или осуществлять выбор между разными конкретными реализациями ADT, и он тоже имеет большое значение.

---

[1] *Mitzenmacher M.* Compressed Bloom filters // IEEE/ACM Transactions on Networking (TON), 2002. — Vol. 10.5 — P. 604–612.

- Многие задачи в информатике связаны с отслеживанием значений. Это могут быть URL-адреса, просматриваемые поисковым роботом, документы, проверенные индексатором, значения для хранения в кэше и т. д.
- В зависимости от контекста иногда необходимо использовать различные дополнительные ограничения для реализации *множества*.
- Рандомизированные алгоритмы — это подмножество алгоритмов, выполнение которых может зависеть от рандомизации, их многократное применение к одним и тем же данным не всегда дает один и тот же результат.
- Рандомизированные алгоритмы делятся на алгоритмы Монте-Карло и Лас-Вегас в зависимости от местоположения неопределенности.
- Для оценки вероятности ложноположительных результатов в фильтрах Блума используется *метрика точности*.
- Если заранее известно максимальное количество элементов, которые будут храниться в фильтре Блума, можно применить точную формулу, чтобы вычислить общий объем памяти, необходимой для достижения сколь угодно низкой вероятности ложноположительных результатов.

5 Непересекающиеся множества: обработка за сублинейное время

В этой главе

- ✓ Решение задачи деления набора данных на непересекающиеся множества и динамического слияния разделов.
- ✓ Описание API структуры данных, представляющей непересекающиеся множества.
- ✓ Простые реализации всех методов с почти линейным временем выполнения.
- ✓ Уменьшение времени выполнения за счет выбора правильной структуры данных.
- ✓ Добавление простых в реализации эвристик для получения квазипостоянного времени выполнения.
- ✓ Распознавание вариантов использования, когда для повышения производительности требуется наилучшее решение.

В этой главе рассмотрим задачу, кажущуюся настолько тривиальной, что многие разработчики просто реализуют очевидное решение, сочтя анализ его производительности не стоящим внимания. Тем не менее если бы к структурам данных можно было применить выражение «волк в овечьей шкуре», то оно было бы лучшим заголовком для этой главы.

Мы используем непересекающиеся множества всякий раз, когда нужно разделить исходный набор объектов на непересекающиеся группы (то есть подмножества, не имеющие общих элементов). Например, можно взять список вин (исходный набор) и разделить вина по вкусовым качествам, создав непересекающиеся множества, объединяющие вина с похожим вкусом и не включающие одни и те же вина. Аналогично можно сгруппировать продукты по их природе и свойствам, например овощи, фрукты, полуфабрикаты и т. д. Это тривиальный пример непересекающихся множеств, показанный на рис. 5.1.

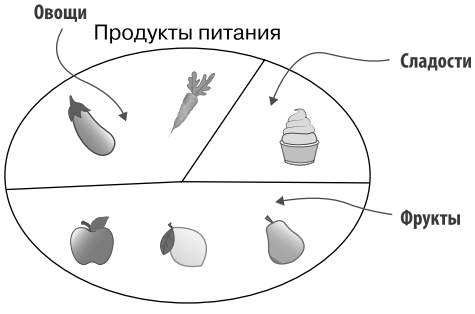


Рис. 5.1. Пример непересекающихся множеств. Весь набор, так называемая вселенная, — это множество «Продукты питания». Он делится на три подмножества: «Фрукты», «Овощи» и «Сладости». Основная особенность заключается в том, что эти подмножества не пересекаются друг с другом

Рассмотрим эту задачу, начав с ее определения, а затем исследуем самые простые (и наивные) алгоритмы, чтобы получить представление о том, как выглядит фактическое решение. После этого углубимся в повышение эффективности решений и посмотрим, как можно использовать их в составе более сложных алгоритмов. К концу главы вы сможете написать наилучшее возможное решение задачи получения непересекающихся множеств и использовать его для увеличения производительности приложений.

5.1. ЗАДАЧА РАЗДЕЛЕНИЯ НА ПОДМНОЖЕСТВА

Например, представьте такую ситуацию: вы запускаете в работу новый веб-сайт электронной коммерции, предоставляющий пользователям неперсонализированные рекомендации. Если это поможет вам убедиться в реальности ситуации, то представьте, что у вас есть машина времени и вы вернулись в 1999 год. Или, более правдоподобно, подумайте об открытии локализованного веб-сайта, имеющего тесные связи с розничными торговцами в вашей стране, или даже специализированного розничного веб-сайта, ориентированного на нишевые товары. В любом случае это может быть очень интересным упражнением.

Итак, неперсонализированные рекомендации. Вы спросите: что это такое? Позвольте мне сделать небольшое отступление, чтобы объяснить. Персонализированные

рекомендации предназначены для отдельных клиентов и подбираются на основе имеющихся данных об этих клиентах (прошлые покупки или метаданные, показывающие сходство с другими пользователями). Но иногда может не быть вообще никаких данных: например, когда запускается новый сайт или даже когда появляется новый клиент, о котором мы вообще ничего не знаем. Вот почему многие веб-сайты, такие как Twitter, Pinterest, Netflix и MovieLens, задают вам вопросы при регистрации; делается это с целью выяснить ваши вкусы и получить возможность дать некоторые, пусть и примерные, персональные рекомендации, основываясь на предпочтениях пользователей с похожими профилями.

С другой стороны, неперсонализированные рекомендации предназначены не для конкретного клиента, а для всех клиентов. Ими могут быть жестко заданные ассоциации, не зависящие от наличия данных, или, например, основанные на покупках, сделанных всеми другими клиентами.

Это именно то, что мы собираемся сделать: всякий раз, когда клиент добавляет что-то в свою корзину, давать ему рекомендацию о том, что он мог бы пожелать приобрести вместе с этим товаром. Наша цель — найти товары, которые часто покупаются вместе; иногда мы будем находить вполне разумные ассоциации, такие как молоко и хлеб, купленные вместе. В других случаях результат может быть более неожиданным: вы, вероятно, уже слышали об исследовании, проведенном в Walmart, согласно которому обнаружилась устойчивая связь между покупкой подгузников и пива — это один из самых цитируемых анекдотов в области науки о данных.

На рис. 5.2 показано, чего мы хотим достичь. Изначально из-за отсутствия каких-либо данных рассматриваем каждый товар как отдельную группу или, если хотите, принимаем, что никакие два товара не имеют связей между собой.

Когда два товара часто приобретаются вместе, между ними устанавливается связь, они начинают считаться частью одной группы. Для простоты представьте, что команда специалистов по обработке и анализу данных установила правило, согласно которому категории товаров X и Y объединяются, если в течение последнего часа товары из этих категорий покупаются вместе и сумма покупки превышает некоторый фиксированный порог.

Например, если за последний час телефоны и планшеты были куплены вместе более 500 раз (это более 1 % от общего количества покупок), то они должны быть объединены в общую группу.

В таком случае, если клиент покупает товар X, мы можем предложить ему приобрести случайный товар из той же группы.

Описанный процесс довольно распространен в науке о данных. Некоторые из вас, возможно, поняли, что это не что иное, как иерархическая кластеризация. Если вы не догадались, то не волнуйтесь, мы обсудим кластеризацию в подразделе 5.7.3.

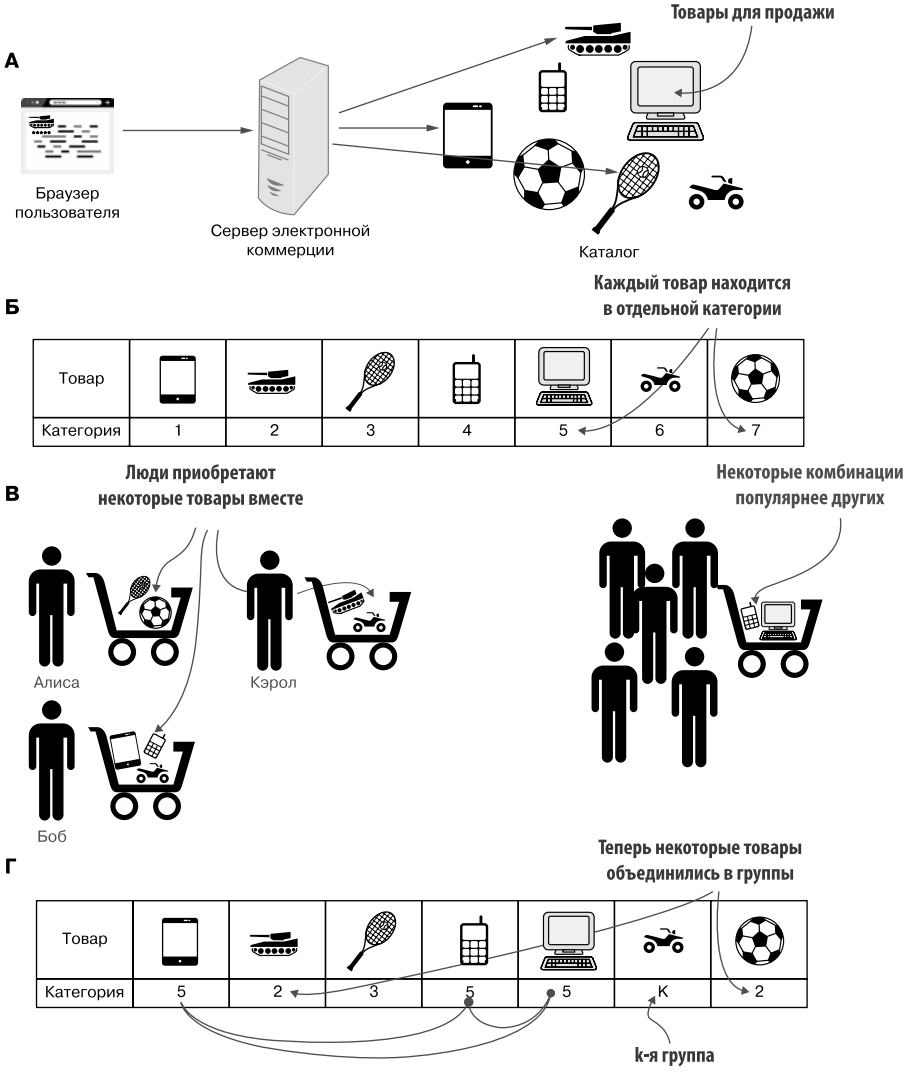


Рис. 5.2. Пример применения непересекающихся множеств. *А*. В этом сценарии веб-сайт электронной коммерции пытается понять, какие товары чаще всего продаются вместе, чтобы дать более точные рекомендации своим клиентам. *Б*. Изначально каждый товар находится в собственной категории. (Впрочем, описываемый подход к объединению в группы товаров, продающихся вместе, можно использовать, даже если товары уже объединены в предопределенные категории. Но ради простоты предположим все же, что в продаже есть только один товар из каждой категории.) *В*. Товары, которые часто покупаются вместе, такие как ноутбуки и внешние диски или теннисные ракетки и теннисные мячи, объединяются в группы. *Г*. Через некоторое время ситуация стабилизируется и сформируются устойчивые группы. Теперь, когда покупатель в следующий раз добавит в корзину футбольный мяч, мы сможем предложить ему приобрести еще и пару лыж

Конечно, это весьма упрощенный пример. В настоящей неперсонализированной системе рекомендаций мы бы выявляли ассоциации между товарами, измеряя силу связи как уверенность в том, что, когда покупается *X*, то покупается и *Y*. Для этого можно вычислить отношение количества покупок обоих товаров вместе к общему количеству покупок товара *Y*. Эта характеристика дает лучшее представление о том, что с чем связано, помогает точнее определить порог достоверности для объединения групп, и вместо выбора случайного товара в той же группе с ее помощью оказывается возможным показать пять самых сильных ассоциаций.

Как бы то ни было, кластеризацию (объединение) элементов в группы можно считать разумным шагом, потому что она улучшит производительность, позволив применять некоторые алгоритмы к каждой группе товаров по отдельности, а не для всего каталога.

Тем, кто заинтересовался системами неперсонализированных (и персонализированных) рекомендаций, можно посоветовать прекрасное и подробное руководство по этой теме: книгу *Practical Recommender Systems* (Кима Фалька (Kim Falk), Manning Publications, 2019)[1].

Но вернемся к нашему примеру: идея состоит в том, чтобы, начав работать с огромным множеством элементов, разделить его на непересекающиеся группы. Не забывайте при этом, что каталог постоянно пополняется новыми элементами, а отношения имеют динамический характер, поэтому необходимо иметь возможность обновлять не только список элементов, но и группы.

5.2. РАЗМЫШЛЕНИЯ О ВОЗМОЖНЫХ РЕШЕНИЯХ

Для обозначения непересекающихся групп в этом и следующих разделах я буду использовать термин *«раздел»* (partition). Однако иногда будут употребляться термины *«группа»*, *«набор»* и *«множество»* как его синонимы.

Мы ограничимся агрегатным случаем, когда два раздела могут объединиться в один большой набор, но не наоборот. То есть раздел нельзя разделить на два подмножества.

А теперь представьте, что во время обсуждения проекта вашей командой специалистов по обработке и анализу данных и командой инженеров службы поддержки один из инженеров вскочил и воскликнул: «Это просто! Нужно лишь создать отдельный массив (динамический массив или вектор) для каждого подмножества!»

Определенно не стоит уподобляться такому инженеру, поверьте мне на слово. И одна из целей этой книги — помочь вам не растеряться в такой ситуации. Потому что следующее, что произойдет: кто-то укажет, что для определения, находятся ли два элемента в одном и том же наборе, необходимо просмотреть все элементы всех подмножеств. Аналогично для выяснения, какому подмножеству принадлежит элемент,

[1] *Фальк К.* Рекомендательные системы на практике.

необходимо такое же количество проверок; то есть продолжительность поиска может линейно зависеть от общего количества элементов.

Такое решение страдало бы реальной проблемой производительности, и кажется очевидным, что надо найти более удачное решение.

Следующая идея в этом мозговом штурме: добавить к спискам подмножеств, упомянутым выше, ассоциативные массивы, отображающие элементы в подмножества. Это даст некоторое увеличение производительности отдельных операций; хотя, как мы увидим далее, операции слияния двух множеств по-прежнему требуют $O(n)$ присваиваний.

Производительность, однако, не главная задача при такой организации. Использование двух независимых структур данных — плохая идея, потому что их придется синхронизировать вручную, а это весьма чревато ошибками.

Более удачным решением выглядит создание класса-обертки, который внутренне использует эти две структуры: класс обеспечит инкапсуляцию и изоляцию, и написать код синхронизации структур придется только один раз (попутно появится возможность повторного применения этого кода). Что еще более важно, вы сможете организовать модульное тестирование своего класса изолированно и, следовательно, получить разумную гарантию, что он будет безошибочно выполнять свою работу при каждом использовании в приложении (при условии, что модульные тесты, предусматривающие пограничные случаи и проверяющие поведение класса во всех возможных контекстах, будут хорошо написаны).

Итак, предположим, что мы договорились о необходимости написать класс, который позаботится обо всем, отслеживая, какому (непересекающемуся) подмножеству принадлежит элемент, и инкапсулируем в нем всю логику. Отложим пока обсуждение реализации: прежде чем сосредоточиться на деталях, нужно еще обсудить общедоступный API и его поведение.

В зависимости от размера каталога можно было бы даже поместить такую структуру данных в память, но вместо этого лучше предположить, что у нас есть служба REST (рис. 5.3) с хранилищем, похожим на Memcached[1], что-то вроде Redis. В этом случае важна сохранность данных, потому что деятельность по мониторингу элементов будет длиться годами и не хотелось бы пересчитывать всю структуру непересекающихся множеств каждый раз, когда происходит какое-то изменение или появляется новый товар.

В качестве альтернативы, если размер вселенной[2] достаточно мал, чтобы поместиться в памяти, можно было бы представить себе механизм синхронизации, который периодически сериализует нашу структуру данных в памяти в постоянную базу данных.

---

[1] Простое хранилище ключей и значений (без поддержки SQL), используемое в качестве распределенной системы кэширования объектов.

[2] Множество всех возможных элементов в теории множеств традиционно называется вселенной (Universe, U).

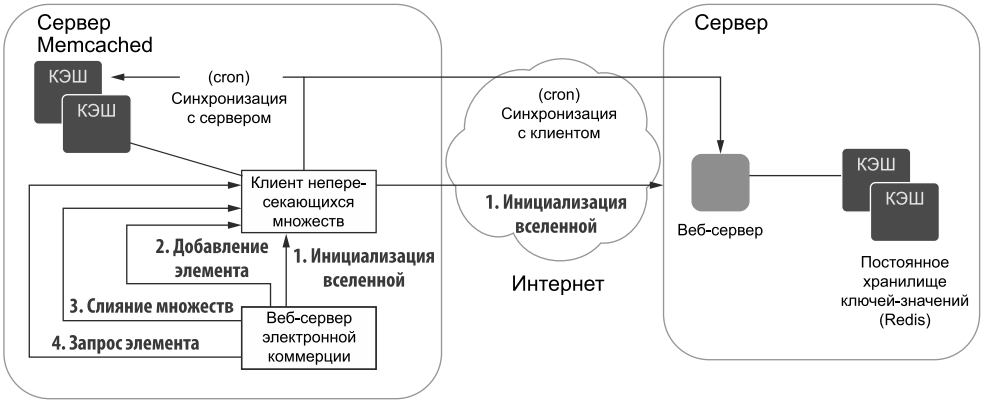


Рис. 5.3. Возможная организация приложения, использующего непересекающиеся множества. Клиентом непересекающихся множеств может быть что угодно, от библиотеки до клиента REST. Цель (тонкого) клиента в этом случае — служить интерфейсом между хранилищем в памяти (узел Memcached на рисунке) и сервером с постоянным хранилищем. Сервером может быть веб-сервер или другое специализированное приложение, хранящее данные на диске. Клиент непересекающихся множеств работает в той же интрасети, возможно, даже на том же компьютере, что и сервер электронной коммерции. Для него будет отдельное задание в cron, выполняющее синхронизацию постоянного хранилища с версией в памяти (это может происходить каждые несколько секунд или асинхронно после каждой операции). Кроме того, он будет отвечать на запросы с сайта электронной коммерции, обслуживать хранилище в оперативной памяти и при необходимости обращаться к веб-серверу (и, если не все данные поместятся в памяти, он же возьмет на себя поддержку подкачки)

5.3. ОПИСАНИЕ API СТРУКТУРЫ ДАННЫХ: НЕПЕРЕСЕКАЮЩЕЕСЯ МНОЖЕСТВО

В нашем примере структура данных должна предлагать несколько важных операций.

Абстрактная структура данных	

API	`class DisjointSet {`
	`  init(U);`
	`  findPartition(x);`
	`  merge(x, y);`
	`  areDisjoint(x,y);`
	`}`

Контракт с клиентом	Непересекающееся множество отслеживает взаимоотношения между элементами во вселенной `U`. Отношения не определяются структурой данных; эту задачу должен решать клиент.

	Однако предполагается, что такое отношение ® обладает свойствами рефлексивности, симметричности и транзитивности; то есть для элементов x, y, z вселенной U: • x ® x; • если x ® y, то y ® x; • если x ® y и y ® z, то x ® z. Гарантии, предоставляемые классом: • возможность добавить связь между любыми двумя элементами; • если два элемента в какой-то момент объединяются (то есть добавляется отношение между ними), они становятся частью одного непересекающегося множества; • если существует цепочка элементов x_1, x_2 ... x_n такая, что x_1 объединен с x_2, x_2 объединен с x_3 и т. д., то x_1 и x_n окажутся в одном разделе; • если два элемента находятся в разных разделах, то не может быть третьего элемента, принадлежащего обоим непересекающимся множествам, в которых находятся первые два элемента

Первый очевидный шаг — инициализация экземпляра на этапе создания. Без потери общности можно ограничиться случаем, когда вселенная U, то есть множество всех возможных элементов, известна заранее и статична. Также предположим, что изначально каждый элемент находится в отдельном разделе. Эти ограничения легко преодолеть за счет разумного использования динамических массивов и собственных методов класса.

Наконец, на протяжении всей главы будет предполагаться, что элементы нашей вселенной U — целые числа от 0 до $n - 1$. Это довольно слабое ограничение, потому что каждый фактический элемент U можно связать с индексом.

Итак, инициализация сводится к выделению в памяти места для полей, необходимых классу, и к назначению каждому элементу своего раздела.

Вызов метода `findPartition` для элемента x из U возвращает раздел, которому принадлежит x. Возвращаемый результат может не иметь смысла за пределами экземпляра структуры данных: думайте об этом методе как о *защищенном (protected)*[1] или даже прикиньте возможность сузить его область видимости, объявив приватным (private).

Вот две основные операции, которые нам нужны:

- проверка принадлежности двух элементов в U, x и y, разным разделам (`areDisjoint`);
- объединение разделов двух элементов, x и y, в общий (`merge`).

[1] Определение защищенной области видимости зависит от языка программирования. Здесь предполагается, что защищенный метод или атрибут доступен только классу, где он объявлен, а также его подклассам. Приватный метод, наоборот, не виден никаким классам, наследующим класс, в котором он объявлен.

5.4. ПРОСТЕЙШЕЕ РЕШЕНИЕ[1]

Самое прямолинейное решение нашей задачи — представить каждый раздел списком (или массивом), как показано на рис. 5.4. Каждый элемент должен иметь указатель на начало списка.

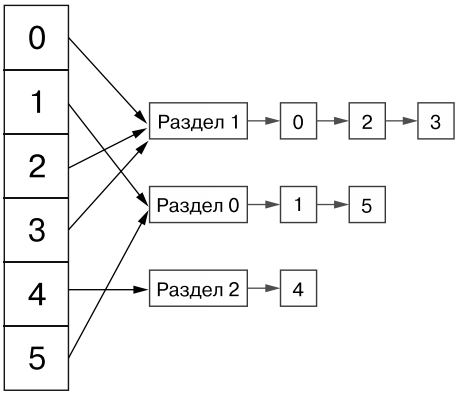


Рис. 5.4. Представление непересекающихся множеств списками. Каждый элемент массива хранит указатель на начало связанного списка. Каждый связанный список, в свою очередь, представляет множество. Здесь множества пронумерованы произвольно, потому что индекс не несет никакой информации о множестве (и не может быть получен)

Чтобы узнать, находятся ли два элемента в одном разделе, нужно получить список, на который ссылается один элемент, и проверить, ссылается ли на тот же список другой[2].

Чтобы объединить два раздела, P_1 и P_2, представленные двумя списками, L_1 и L_2, нужно обновить указатель `next`[3] последнего элемента в L_1, чтобы он указывал на начало L_2 (или, наоборот, последнего элемента списка L_2, чтобы он указывал на начало L_1). Эту операцию, изображенную на рис. 5.5, можно выполнить за постоянное время, если сохранить дополнительный указатель на конец каждого списка. К сожалению, мы еще не закончили: для всех элементов в L_2 также нужно обновить их указатели на список в ассоциативном массиве, чтобы они указывали на начало нового объединенного списка.

В худшем случае время выполнения этой операции линейно зависит от количества элементов, потому что может понадобиться обновить до $n - 1$ элементов (где n — общее количество элементов во вселенной `U`).

Есть один способ немного уменьшить ожидаемое количество присваиваний: всегда добавляя самый короткий из двух списков, можно гарантировать, что не придется обновлять указатели у более чем $n/2$ элементов. К сожалению, это не улучшает асимптотического времени выполнения[4].

---

1 Этот раздел посвящен теории.

2 Для реализации, вероятно, потребуется использовать здесь проверку равенства ссылок.

3 Чтобы вспомнить, как организованы связанные списки, или если вы не уверены, что означает указатель next, обращайтесь к приложению B.

4 Не забывайте, что постоянные коэффициенты не имеют значения в анализе «O большого», поэтому O(n / 2) = O(n).

Теперь обратимся к коду, чтобы лучше понять, как все это работает.

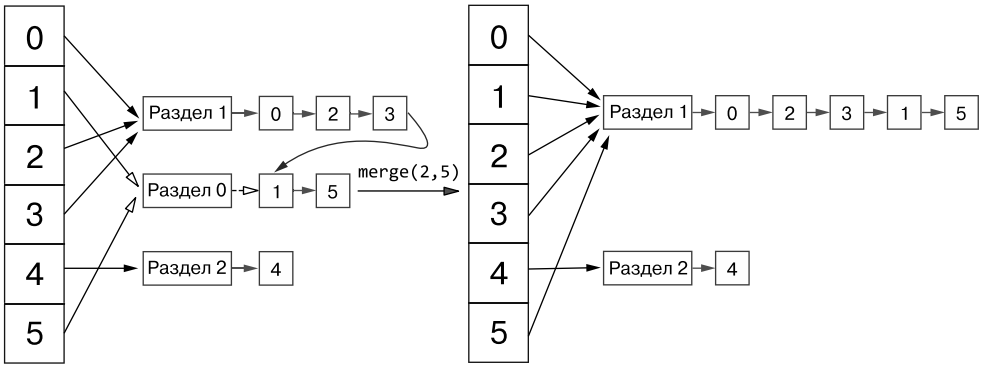


Рис. 5.5. Объединение двух разделов. *Слева:* один из двух списков добавляется в конец другого созданием нового ребра, связывающего «хвост с головой», и удалением ссылок из массива на второй список. *Справа* показан результат после слияния с обновленными указателями в элементах массива, принадлежащих добавленному списку (элементы 1 и 5)

5.4.1. Реализация простейшего решения

Начнем с псевдокода определения класса и конструктора (листинг 5.1). Все методы, представленные в следующих разделах, будут принадлежать этому классу `DisjointSet`.

Листинг 5.1. Простейшее решение, конструктор

```
class DisjointSet
  #type HashMap[Element, List[Element]]
  partitionsMap

function DisjointSet(initialSet=[])
  this.partitionsMap ← new HashMap()
  for elem in initialSet do
    throw-if (elem == null or partitionsMap.has(elem))
    partitionsMap[elem] ← new Set(elem)
```

Конструктор принимает список элементов и по умолчанию инициализирует непересекающееся множество пустым набором

Создаем новый ассоциативный массив с элементами в множестве

Выполняем обход элементов списка в аргументе

Добавляем связь между текущим элементом и новым множеством, содержащим только текущий элемент

Генерируем исключение, если встречено значение null или повторяющийся элемент

Инициализация реализуется просто: список, переданный в аргументе, проверяется на наличие дубликатов, и из него инициализируется непересекающееся множество.

В реальной реализации следует побеспокоиться о сравнении элементов. В зависимости от языка программирования можно использовать операцию сравнения ссылок, оператор равенства или метод, определенный в классе элементов. Представленный

здесь код предназначен только для иллюстрации базового решения, поэтому не будем заботиться о таких деталях.

Сначала инициализируется ассоциативный массив, индексирующий элементы и отображающий их в соответствующие им разделы (строка 2).

Далее выполняется обход элементов в `initialSet` и проверяется, определены ли они и нет ли дубликатов, а затем их разделы инициализируются одноэлементными множествами, содержащими сами элементы (изначально все элементы находятся в своих отдельных множествах).

Теперь, реализовав инициализацию непересекающегося множества, добавим пару полезных методов. Например, общедоступное свойство `size`, хранящее количество записей в локальном ассоциативном массиве разделов.

Примеры реализации этих методов можно найти в репозитории. Здесь же сосредоточимся на основных методах API, начиная с метода `add`, показанного на рис. 5.6 и в листинге 5.2.

Листинг 5.2. Простейшее решение, add

```
function add(elem)
  throw-if elem == null
  if partitionsMap.has(elem) then
    return false
  partitionsMap[elem] ← new Set(elem)
  return true
```

Принимает элемент и возвращает true тогда и только тогда, когда элемент был успешно добавлен в множество, иначе возвращается false

Проверка допустимости элемента

Если элемент уже присутствует в структуре данных, возвращается false и никакие изменения не применяются

Иначе элемент добавляется в ассоциативный массив, ему назначается отдельное множество и возвращается true

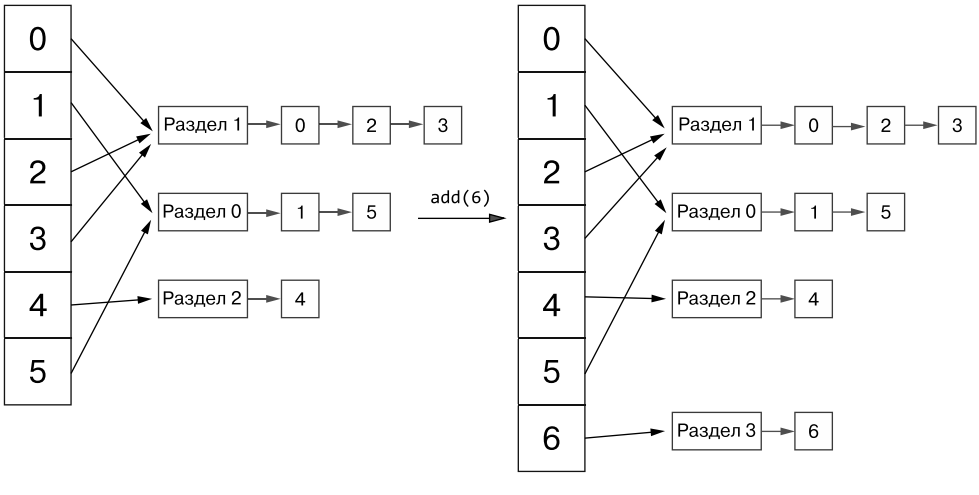
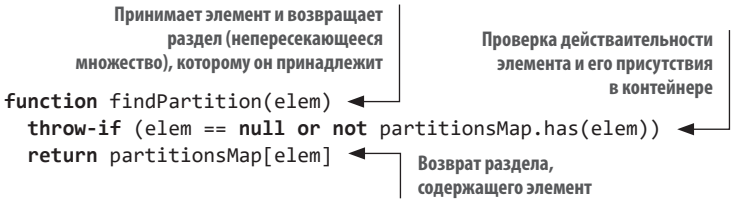


Рис. 5.6. Добавление нового элемента в контейнер. Если новый элемент не является дубликатом какого-либо элемента, имеющегося в настоящее время в контейнере, его можно добавить, создав новый отдельный раздел, содержащий только вновь добавленный элемент

Этот метод позволяет вселенной прирастать новыми (уникальными) элементами, которые могут быть добавлены в любое время. Каждый раз, когда добавляется новый элемент, для него создается новый отдельный раздел, содержащий только этот элемент. И конечно же, перед добавлением нужно убедиться, что в аргументе передан действительный элемент и он не был добавлен в нашу вселенную ранее.

Теперь мы подошли к самому интересному: к методу `findPartition`, показанному в листинге 5.3.

Листинг 5.3. Простейшее решение, findPartition

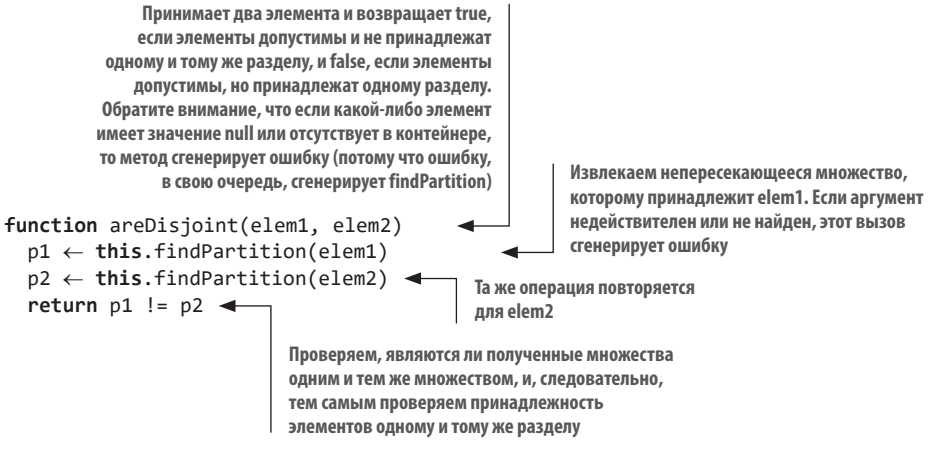


В этой базовой реализации метод очень прост: после обычной проверки (включая проверку действительности элемента и его присутствия в непересекающемся множестве) он просто возвращает раздел, содержащий этот элемент.

Как уже упоминалось, эта реализация метода `findPartition` имеет постоянное время выполнения (при условии, что хеш для `elem` можно вычислить за постоянное время).

Другой простой в реализации метод, показанный в листинге 5.4, проверяет принадлежность двух элементов одному и тому же разделу.

Листинг 5.4. Простейшее решение, areDisjoint



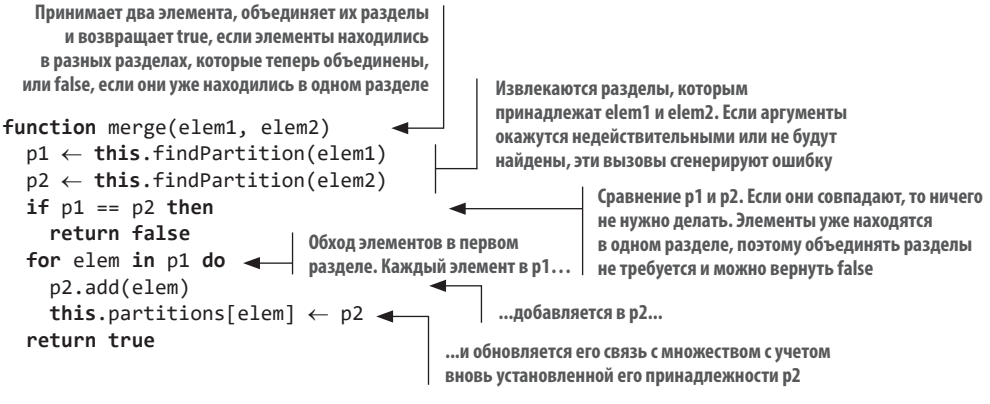
Здесь достаточно просто повторно использовать `findPartition`, вызвав этот метод для обоих элементов и проверив, возвращают ли эти два вызова один и тот же раздел. Обратите внимание, что, повторно используя `findPartition`, можно с уверенностью утверждать, что реализацию `areDisjoint` не придется менять при изменении способа

хранения наших элементов или реализации `findPartition` (при условии, что его интерфейс остается прежним, а разделы можно сравнивать оператором неравенства).

Более того, мы решили реализовать проверку принадлежности элементов разным разделам, а не одному и тому же разделу: это связано с типичными способами использования непересекающихся множеств. Обычно вызывает интерес, не принадлежат ли два элемента одному и тому же разделу, и, если это так, два раздела объединяются. Но в зависимости от того, как вы собираетесь использовать этот контейнер, возможно, удобнее будет обратное и ничто не помешает определить метод `samePartition`.

Все методы, представленные выше, выполняются за постоянное время независимо от размера контейнера. Теперь пришло время реализовать метод слияния двух разделов, показанный в листинге 5.5 (и на рис. 5.5). Как уже известно, метод слияния `merge` требует $O(n)$ присваиваний в худшем случае.

Листинг 5.5. Простейшее решение, merge



Этот метод сложнее предыдущих. И все же благодаря повторному использованию `findPartition` он выглядит довольно простым.

Сначала проверяется принадлежность элементов одному и тому же разделу. С этой целью для обоих вызывается метод `findPartition`, и полученные результаты сравниваются. Вызовы позаботятся также о проверке аргументов.

После выяснения, что слияние действительно нужно выполнить, в методе объединяются два множества, тем самым корректируются указатели в ассоциативном массиве разделов. Если бы разделы вместо множества `Set` были реализованы в виде связанных списков, то можно было бы просто добавить ссылку на начало одного списка из конца другого. Применение множества `Set` вынуждает добавлять элементы по одному. Для этого требуется дополнительное количество присваиваний, зависящее от объема множества (в худшем случае), что не меняет порядка выполнения функции; ведь все равно нужно обновить ссылки в элементах в одном из двух списков (то есть множеств).

Здесь показано простейшее решение, которое всегда перемещает элементы из первого раздела во второй. В репозитории на GitHub[1] вы найдете немного улучшенную версию предложенной реализации, которая проверяет, какое множество меньше, и добавляет его элементы в большее множество; однако это очень незначительное улучшение простейшей версии, и время работы остается линейным при минимальном размере множеств.

5.5. ИСПОЛЬЗОВАНИЕ ДРЕВОВИДНОЙ СТРУКТУРЫ[2]

Подведем итоги, чего мы с вами достигли в базовой реализации: удалось написать метод `findPartition` с постоянным временем выполнения и метод `merge` с линейным временем в наихудшем случае.

Можно ли добиться большего и оптимизировать время выполнения не только для `findPartition`, но и для всех операций с непересекающимся множеством? Как оказывается, можно!

5.5.1. От списка к деревьям

Идея проста: вместо списков, представляющих разделы, можно использовать деревья, как показано на рис. 5.7. В таком случае каждый раздел однозначно идентифицируется корнем связанного с ним дерева. Главное преимущество деревьев перед списками: если дерево сбалансировано, то любая операция над деревом выполняется за логарифмическое время (а не за линейное, как в случае со списками).

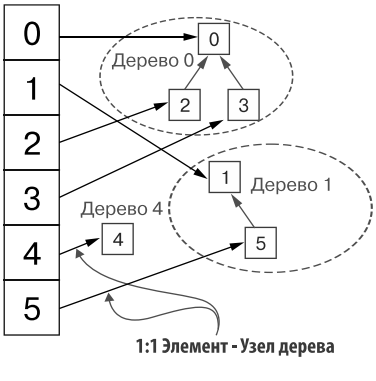


Рис. 5.7. Представление непересекающихся множеств с использованием деревьев. Деревья названы по их корням, потому что корень дерева используется в роли уникального идентификатора раздела (мы можем сделать это, предполагая уникальность элементов). Каждый элемент в массиве ссылается на узел в дереве: в простейшей реализации существует прямая связь между элементами и узлами дерева. То есть для того, чтобы добраться до корня дерева, может понадобиться пересечь все дерево снизу вверх (и в среднем половину высоты дерева)

Для объединения двух разделов нужно установить корень одного дерева как дочерний элемент корня другого дерева; пример показан на рис. 5.8.

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#disjoint-set.

[2] Этот раздел посвящен теории и объясняет некоторые продвинутые концепции.

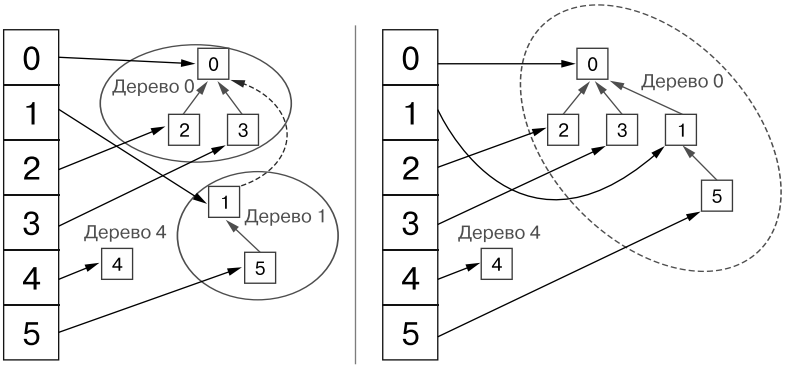


Рис. 5.8. Объединение двух множеств с использованием представления в виде деревьев. Требуется создать только одну новую ссылку (плюс выполнить обход некоторых узлов дерева). На рисунке *слева* добавляется одно ребро от корня дерева 1 к корню дерева 0, чтобы объединить их. *Справа* показано, как это меняет структуру данных: теперь есть только два дерева, но высота дерева 0 стала больше

Таким образом достигнуто огромное улучшение по сравнению с простейшим решением, потому что отпадает необходимость обновлять в ассоциативном массиве разделов ссылки элементов на объединенное множество. Вместо этого каждый узел в дереве поддерживает ссылку на своего родителя (сохранять ссылки на дочерние элементы не требуется, потому что в данном случае они бесполезны).

Корни деревьев, как уже упоминалось, однозначно идентифицируют каждый раздел. Итак, когда потребуется узнать, к какому разделу принадлежит элемент, можно просто извлечь узел дерева, на который он указывает, и перейти к корню этого дерева. Если в методе `areDisjoint` сделать так для обоих элементов, а затем сравнить найденные корни, то можно выяснить, принадлежат ли два элемента одному и тому же разделу (то есть равны ли найденные корни).

Таким образом, для слияния двух разделов теперь требуется постоянное количество изменений плюс некоторое количество операций, необходимых для поиска двух корней. Поиск множества, которому принадлежит элемент (или проверка принадлежности двух элементов одному и тому же разделу), в среднем требует логарифмического времени (об этом рассказывалось, когда мы обсуждали деревья[1]) и линейного в худшем случае. Так происходит потому, что при объединении разделов может не повезти с выбором корня дерева, устанавливаемого в качестве дочернего по отношению к другому. Каждый раз, случайно выбирая, какой корень будет потомком другого, можно сделать наихудший сценарий маловероятным... но не невозможным (хотя и крайне редким). Поэтому сохраняется риск оказаться в граничной ситуации, подобной той, что изображена на рис. 5.9. Это означает, что наихудший для слияния

[1] Загляните в подраздел B.1.3 приложения B.

сценарий по-прежнему требует O(n) операций поиска и, что еще хуже, теперь даже `findPartition` имеет линейное время выполнения.

Прежде чем задуматься, что можно предпринять, рассмотрим код улучшенной версии.

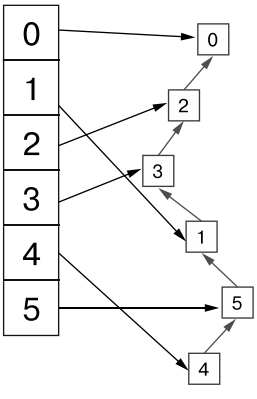


Рис. 5.9. Наихудший сценарий реализации простого дерева: высота получившегося дерева равна n, общему количеству элементов, потому что дерево выродилось в список

5.5.2. Реализация версии с деревьями

Рассмотрим подробно, как работает эта улучшенная реализация. Большая часть кода осталась неизменной по сравнению с предыдущим разделом, поэтому мы не будем его здесь показывать. В методах, подвергшихся изменениям, выделим сами изменения, чтобы вы могли быстро сравнить две версии.

Элементы в ассоциативном массиве разделов будут теперь ссылаться не на фактические множества, а на родительский элемент в дереве. Вот почему, как можете увидеть в репозитории книги, можно переименовать `partitionsMap` в `parentMap`, чтобы явно обозначить его предназначение.

При инициализации каждому элементу присваивается ссылка на самого себя как на своего родителя. Причина такого решения вскоре будет объяснена.

То же относится и к методу `add`, который в остальном остается неизменным.

Метод `findPartition` (листинг 5.6) требует незначительных изменений для правильной работы. Вот два замечания по его реализации:

- в базовой реализации мы станем возвращать не список, а элемент в корне дерева раздела;
- возвращаемое значение `findPartition`, вероятно, не будет иметь смысла для внешнего вызывающего кода, однако на самом деле ожидаемое место использования этого метода — внутри, в вызовах методов `merge` и `areDisjoint`.

После получения родителя элемента выполняется проверка, является ли он самим этим элементом. Если элемент является своим родителем, то это означает, что корень

дерева раздела достигнут. Ведь именно так инициализируется такое поле, и при слиянии родитель корня никогда не изменяется.

Листинг 5.6. Реализация на основе деревьев, findPartition

```
class DisjointSet
  #type HashMap[Element, Tree[Element]]
  parentsMap

function findPartition(elem)
  throw-if (elem == null or not parentsMap.has(elem)))
  parent ← this.parentsMap[elem]
  if parent != elem then
    parent ← this.findPartition(parent)
  return parent
```

Принимает элемент и возвращает другой элемент, находящийся в корне дерева раздела, которому принадлежит elem

Проверка допустимости элемента

Извлечение родителя элемента

Если родителем текущего элемента является сам elem, значит, корень дерева уже достигнут; иначе...

...нам нужно рекурсивно подняться к корню в поиске родительского раздела

В этой точке parent — это корень дерева раздела, в котором находится elem

В противном случае, если у текущего элемента есть родитель, начинается подъем по дереву к его корню рекурсивным вызовом `findPartition` (строка 5), и по завершении возвращается его результат.

Эта новая реализация `findPartition`, как уже отмечалось, имеет непостоянное время выполнения. Она реализует рекурсивные вызовы, количество которых соответствует расстоянию до корня дерева разделов. Пока нет возможности сделать какие-либо предположения о деревьях. Поэтому можно считать, что число вызовов пропорционально количеству элементов во вселенной `U`, хотя это наихудший случай и в среднем можно ожидать более высокой производительности.

Может показаться, что мы только ухудшили производительность структуры данных. Но давайте определим новую реализацию операции `merge` (листинг 5.7), чтобы увидеть преимущества использования деревьев.

Листинг 5.7. Реализация на основе деревьев, merge

```
function merge(elem1, elem2)
  p1 ← this.findPartition(elem1)
  p2 ← this.findPartition(elem2)
  if p1 == p2 then
    return false
  this.parentsMap[p2] ← p1
  return true
```

Принимает два элемента, объединяет их разделы и возвращает true тогда и только тогда, когда элементы находятся в разных разделах, теперь объединенных, или false, если они уже находились в одном разделе

Извлекаются разделы, которым принадлежат elem1 и elem2. Если аргументы окажутся недействительными или не будут найдены, эти вызовы сгенерируют ошибку

Сравнение p1 и p2. Если они совпадают, то ничего не нужно делать. Элементы уже находятся в одном разделе, поэтому объединять разделы не требуется и можно вернуть false

Назначает p1 родителем p2, и с этого момента p1 будет корнем для p2 и опосредованно для всех элементов в разделе p2

Сравнив две реализации, легко заметить, что эта проще, хотя в ней изменились лишь несколько последних строк. Самое замечательное, что отпала необходимость перебирать список элементов! Чтобы объединить два раздела, достаточно добраться до

корней обоих деревьев, а затем назначить один корень родителем другого. Но найти эти корни методу все-таки необходимо.

Новая строка выполняется за постоянное время, поэтому основной вклад во время выполнения этого метода вносят два вызова `findPartition`. Как уже известно, им требуется время, пропорциональное высоте дерева, и в худшем случае оно может быть линейным. Однако в среднем случае, и особенно на ранних этапах после инициализации, высота деревьев будет незначительной.

Таким образом, в этой реализации непересекающегося множества все операции в худшем случае по-прежнему требуют линейного времени, а в среднем — только логарифмического, даже операции, которые выполнялись за постоянное время в простейшей реализации. Определенно, это не выглядит воодушевляющим достижением, если сосредоточиться на худших случаях. Однако, если взглянуть с другой стороны, нам удалось получить более сбалансированный набор операций, которые дадут существенный выигрыш в контекстах, где `merge` является обычной операцией. (В приложениях, где чаще происходит чтение, а слияние выполняется редко, простейшая реализация может оказаться предпочтительнее.)

Но прежде, чем отказаться от решения на основе деревьев, прочтите следующий раздел, он того стоит.

5.6. ЭВРИСТИКА ДЛЯ УЛУЧШЕНИЯ ВРЕМЕНИ ВЫПОЛНЕНИЯ[1]

Следующий шаг в стремлении к оптимальной производительности — добиться логарифмического времени выполнения `findPartition` даже в худшем случае. К счастью, это довольно легко! В приложении B обсуждаются сбалансированные деревья; загляните туда, если чувствуете, что недостаточно хорошо понимаете, о чем идет речь.

Проще говоря, в реализацию на основе деревьев легко добавить хранение *ранга* (размера) каждого дерева, а в метод `merge` — дополнительные операции с постоянным временем выполнения, обновляющие ранг только в корнях деревьев.

При объединении двух деревьев можно выбирать на роль дочернего дерево с наименьшим количеством узлов, как показано на рис. 5.10.

По индукции можно доказать, что это дерево также будет иметь наименьшую высоту, а значит, новое дерево будет иметь ту же высоту, что и старое, либо его высота будет увеличена на 1. Также можно доказать, что высота дерева не может увеличиться более чем в логарифмическое число раз.

Поскольку логарифм растет очень медленно (например, ln(1000) ~= 10, ln(1 000 000) ~= ~= 20), на практике это уже хороший результат, достаточный для большинства приложений.

---

[1] Этот раздел посвящен теории и объясняет некоторые продвинутые концепции.

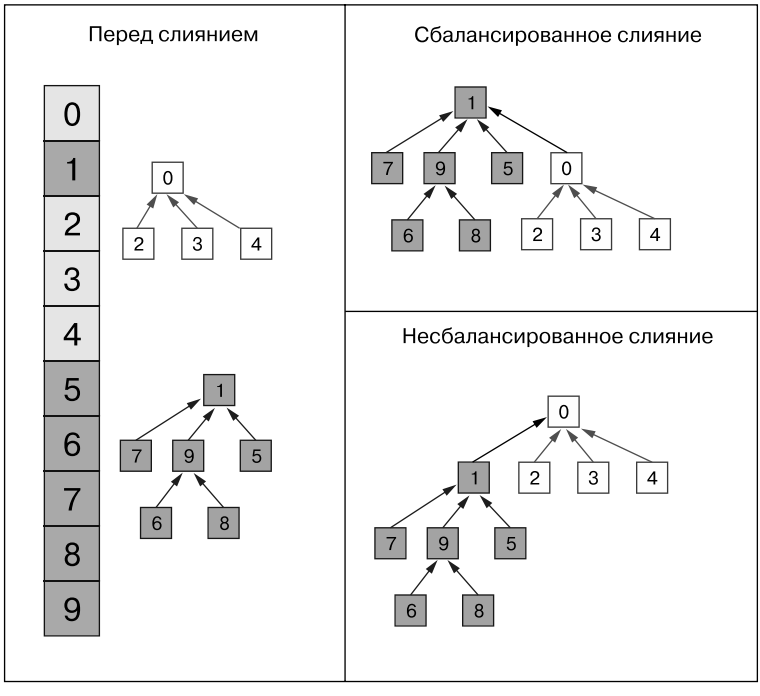


Рис. 5.10. Слияние двух деревьев: примеры сбалансированного и несбалансированного слияния. Стрелки из массива опущены для удобства, потому что каждый элемент массива указывает на соответствующий элемент дерева (то есть все красные элементы указывают на красное дерево и т. д.)

Однако, если вы пишете какой-то по-настоящему критический код, например код ядра операционной системы или встроенного ПО, можно сделать еще лучше.

Почему? Просто потому, что вы можете. А иногда и потому, что это нужно. Если сэкономить 0,001 мс на операции, повторяющейся миллиард раз, суммарная экономия составит 16 мин вычислений.

ПРИМЕЧАНИЕ

В большинстве случаев производительность не единственная характеристика, на которую обращают внимание разработчики. Многое зависит от того, экономятся ли эти 16 мин в ходе вычислений, длящихся час или занимающих сутки (излишне говорить, что в последнем случае такой выигрыш не имеет значения). А также от того, какую цену приходится платить за экономию. Если это изменение кода делает его хрупким, сложным и трудным для сопровождения или просто требует недель разработки, то вам придется взвесить все за и против, прежде чем пойти по этому пути. К счастью, для непересекающихся множеств такого не случается, и сжатие путей легко реализовать, получив при этом большой выигрыш.

Посмотрим, как добиться еще большего улучшения, прежде чем углубиться в код.

5.6.1. Сжатие пути

Как отмечалось в предыдущем разделе, можно добиться большего, чем позволяют простые сбалансированные деревья и методы с логарифмическим временем выполнения.

Чтобы достигнутые результаты сделать еще лучше, можно использовать эвристику, называемую *сжатием пути* (path compression). Эта простая идея показана на рис. 5.11: для каждого узла в деревьях можно хранить ссылку не на родителя, а на корень дерева. В конце концов, нам не требуется следить за историей выполненных слияний; достаточно быть в курсе, какой элемент является корнем раздела в настоящий момент, и желательно иметь возможность выяснять это как можно быстрее.

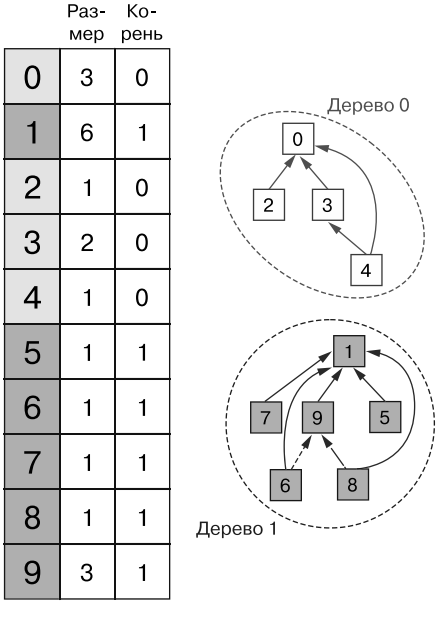


Рис. 5.11. Непересекающееся множество, представленное деревом, поддерживающим сжатие пути. Внутреннее представление показано рядом с массивом элементов. В древовидном представлении пунктирные стрелки — это ссылки на родителей, а сплошные — на корень множества. Структура изначально содержит два множества, окрашенных в светло-красный и темно-синий цвета, корнями которых являются соответственно элементы 0 и 1

Однако для обновления указателей на корень в процессе слияния потребуется уже не логарифмическое, а линейное время, прямо пропорциональное количеству узлов в дереве.

Но посмотрим, что получится, если обновлять указатели в узлах дерева не сразу. Тогда в следующий раз, когда будет вызван `findPartition` для одного из элементов дерева — назовем его `x`, — ему потребуется пройти по дереву от `x` до старого корня x_R, а затем от x_R до нового корня `R`.

Имейте в виду, что указатели на элементы в старом дереве могли быть до слияния синхронизированы (и тогда методу потребуется выполнить всего два перехода, чтобы добраться до нового корня; рис. 5.12) или нет.

Поскольку методу все равно придется идти вверх по дереву, можем затем повторить шаги от вершины `R` вниз до `x` и обновить указатели на корень во всех этих элементах. Такие действия не повлияют на асимптотическую производительность `findPartition`,

потому что, повторяя тот же путь, мы просто удваиваем количество шагов (а постоянные множители не имеют значения в асимптотическом анализе, как описывается в приложении Б).

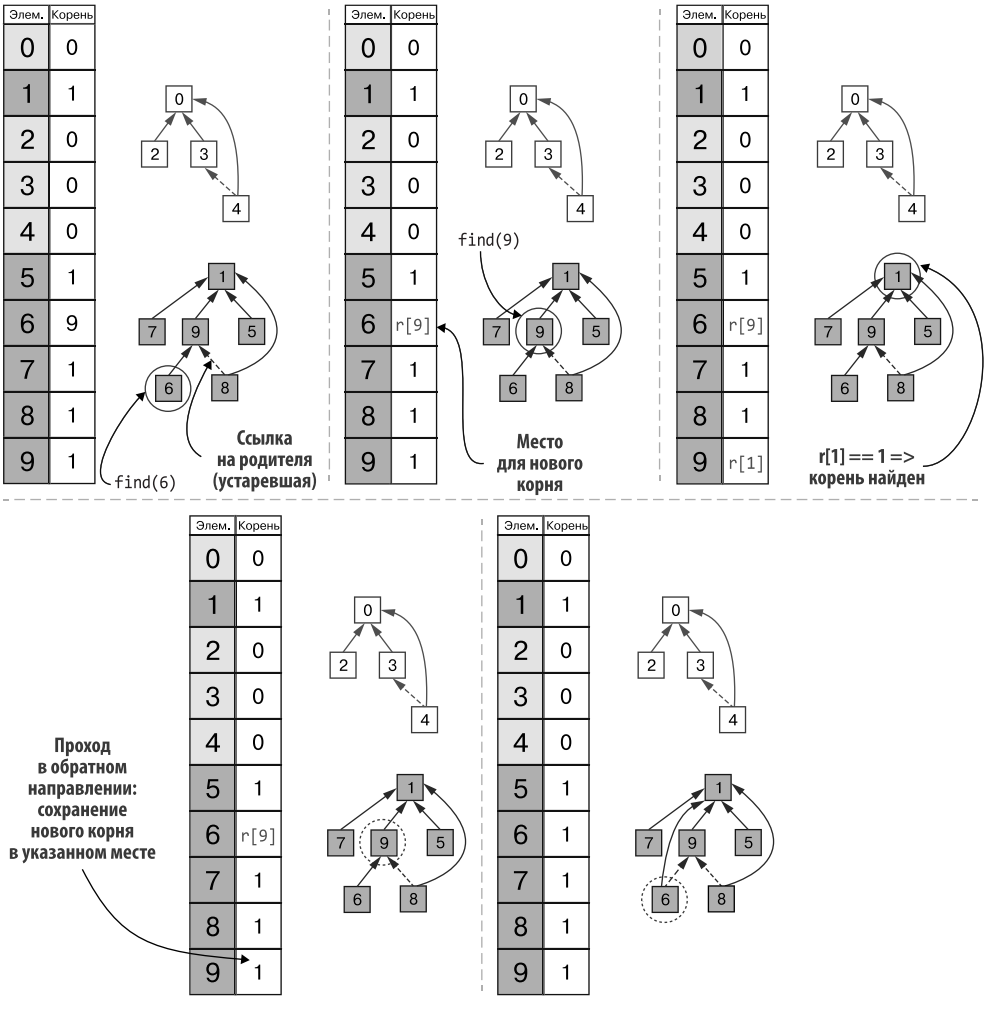


Рис. 5.12. Вызов find на том же непересекающемся множестве, что изображено на рис. 5.11. Обратите внимание, что темно-синее дерево не синхронизировано. Если вызвать find для элемента 6, то алгоритм будет медленно ползти вверх по синему дереву, пока не найдет его корень (третья диаграмма). Затем (нижние диаграммы) алгоритм возвращается назад, обновляя указатели на корень в промежуточных элементах 9 и 6

Но после выполнения этих дополнительных шагов, в следующем вызове `findPartition`, для любого элемента на пути от `x` до `root(x)` будет точно известно, что указатели обновлены и поиск корня займет только один шаг.

На этом этапе важно оценить, сколько раз в среднем потребуется обновлять указатели на корень в одной операции — в амортизированном анализе — для определенного количества k операций. Такой анализ алгоритма выглядит немного сложнее.

Но не будем вдаваться в его детали, потому что уже доказано, что текущее амортизированное время m вызовов `findPartition` и `merge` для набора из n элементов потребует $O(m\,\texttt{Ack(n)})$ обращений к массиву.

Здесь `Ack(n)` — аппроксимация *обратной функции Аккермана*, растущей так медленно, что ее можно считать константой (ее значение будет меньше 5 для любого целого числа, которое можно сохранить на компьютере).

Итак, в результате удалось получить амортизированную константу для всех операций с этой структурой данных! Если вас не впечатлил этот результат, то попробуйте его осмыслить еще раз, потому что на самом деле это действительно большое достижение!

Пока неизвестно, является ли это нижней границей структуры данных Union-Find (состоящей из непересекающихся множеств). Было доказано[1], однако, что $O(m\,\texttt{InvAck(m,n)})$ — строгая нижняя граница, где `InvAck(m, n)` — истинная обратная функция Аккермана.

Я понимаю, что учитывать нужно довольно много. Но не отчаивайтесь: как оказывается, чтобы реализовать эвристику сжатия пути, достаточно лишь нескольких небольших изменений.

5.6.2. Реализация балансировки и сжатия пути

Теперь обсудим окончательную реализацию структуры непересекающихся множеств, включая эвристики «балансировки дерева по рангу» и «сжатия пути».

Каждый элемент должен хранить некоторую информацию о своем поддереве, поэтому используем вспомогательный (приватный) класс `Info` для сбора всей информации, как показано в листинге 5.8.

Листинг 5.8. Класс Info

```
class Info
  function Info(elem)
    throw-if elem == null
    this.root ← elem
    this.rank ← 1

class DisjointSet
  #type HashMap[Element, Info]
  parentsMap
```

Конструктор класса Info принимает элемент непересекающегося множества

Проверяет аргумент

Первоначально элемент находится в собственном отдельном дереве и является его корнем

Ранг поддерева первоначально равен 1, потому что оно содержит только один элемент

[1] Существует масса литературы, посвященной этой теме. Но имейте в виду, что это довольно сложное чтиво, хотя и очень интересное.

Класс `Info` моделирует (содержит информацию) узел дерева раздела. Это всего лишь контейнер для двух значений: корня и ранга (то есть размера) дерева с корнем в текущем элементе.

В свойстве `root` будут храниться не ссылки на другие узлы, а индекс самого элемента, который затем мы используем в качестве ключа в `HashMap`, как уже делали это в предыдущих разделах.

Если бы на самом деле моделировалась древовидная структура данных, то такая организация привела бы к несовершенной инкапсуляции. Но мы используем класс `Info` лишь как кортеж для хранения свойств элемента.

В большинстве реализаций непересекающихся множеств для этого задействуются два массива. Поскольку рассматриваемая сейчас реализация не требует, чтобы ключи были целыми числами, и использует ассоциативные массивы, то можно было бы определить два ассоциативных массива для корней и рангов элементов. Однако в этом случае информация об элементах сохранялась бы трижды: дважды как ключ в каждом ассоциативном массиве и один раз как корень дерева (конечно, эта последняя запись может хранить некоторые ключи несколько раз, а другие — ни разу).

С использованием этой дополнительной обертки и единого ассоциативного массива «с информацией» элементы сохраняются только один раз — в качестве ключей.

Объекты будут сохраняться как ссылки с минимальными накладными расходами, а неизменяемые значения, и особенно строки, будут сохраняться по значению. Таким образом, простое избавление от необходимости сохранять каждый элемент в потенциале дает ощутимую экономию памяти.

Теоретически можно сделать еще лучше: размещать элементы в объектах-обертках и использовать эти обертки в качестве ключей. В таком случае каждый ключ будет сохраняться только один раз и использовать ссылки на обертки в качестве как ключей в ассоциативных массивах, так и значений.

Имеет ли смысл усложнять реализацию ради экономии на накладных расходах в решении с обертками? Это зависит от предположений о типах и размерах ключей. В большинстве случаев выигрыш получится незначительным, поэтому стоит обязательно проанализировать приложение и выполнить его профилирование и только потом приступать к такой оптимизации.

Но вернемся к нашей реализации. Как уже говорилось, исправления минимальны. И в конструкторе, и в методе `add` нужно изменить только самую последнюю строку:

```
parentsMap[elem] = new Info(elem)
```

Необходимо лишь создать новый экземпляр `Info`, связанный с каждым элементом.

В `findPartition` все определенно становится намного интереснее (листинг 5.9).

Как говорилось в начале раздела, при использовании эвристики сжатия пути ссылки на корень в элементах не обновляются в `merge`, но обновляются в `findPartition`. Итак,

главное отличие от старой версии — сохранение в строке 6 результата рекурсивных вызовов `findPartition` в ссылке на корень в текущем элементе. Весь остальной код остается неизменным.

Листинг 5.9. Решение на основе дерева с эвристиками, findPartition

```
function findPartition(elem)
  throw-if (elem == null or not parentsMap.has(elem)))
  info ← this.parentsMap[elem]
  if (info.root == elem) then
    return elem
  info.root ← this.findPartition(info.root)
  return info.root
```

Принимает элемент и возвращает другой элемент — корень дерева раздела, в котором находится заданный элемент

Проверка допустимости элемента

Извлечение информации о текущем элементе

Если элемент является корнем дерева, то его можно вернуть

Иначе следует выполнить рекурсивный поиск корня и попутно обновить ссылку на корень в текущем элементе, чтобы она указывала на фактический корень

Само собой разумеется, что большая часть изменений приходится на метод `merge` (листинг 5.10).

Листинг 5.10. Решение на основе дерева с эвристиками, merge

```
function merge(elem1, elem2)
  r1 ← this.findPartition(elem1)
  r2 ← this.findPartition(elem2)
  if r1 == r2 then
    return false
  info1 ← this.parentsMap[r1]
  info2 ← this.parentsMap[r2]
  if info1.rank >= info2.rank then
    info2.root ← info1.root;
    info1.rank += info2.rank;
  else
    info1.root ← info2.root;
    info2.rank += info1.rank;
  return true
```

Принимает два элемента, объединяет их разделы и возвращает true тогда и только тогда, когда два элемента до объединения находились в разных разделах, и false, если они уже были в одном разделе

Извлечение корней деревьев, которым принадлежат элементы elem1 и elem2. Если аргумент недействителен, то эти вызовы сгенерируют ошибку

Сравнение r1 и r2. Если они совпадают, то ничего не нужно делать, потому что элементы уже находятся в одном разделе и остается только вернуть false

В этой точке известно, что разделы нужно объединить, поэтому извлекается информация об обоих корнях

Проверяем, имеет ли первое дерево больший ранг (больше элементов). Меньшее дерево станет поддеревом большего дерева

Изменяется ссылка на корень в меньшем дереве

Устанавливается ранг в корне равным сумме рангов обоих деревьев. Ранг в другом корне обновлять не требуется, потому что он никогда больше проверяться не будет

Мы все так же, как и раньше, извлекаем корневые элементы деревьев и проверяем их совпадение.

Потом нужно получить информацию для обоих корней и проверить, какое дерево больше. После сравнения меньшее дерево станет дочерним, и в нем нужно переназначить ссылку на корень. Кроме того, необходимо обновить ранг в корне большего дерева; оно теперь будет включать все элементы нового потомка.

Пример практической реализации эвристики можно найти в репозитории книги на GitHub.

Это все, что нужно изменить, чтобы добиться гигантского повышения производительности. Простота кода показывает, насколько разумно это решение, и в следующих разделах вы также увидите, почему очень важно сделать его правильным.

5.7. ПРИМЕНЕНИЕ

Непересекающиеся множества имеют широкий круг применения, именно поэтому они так подробно исследуются в этой книге.

5.7.1. Графы: связанные компоненты

Для трассировки *связанных компонент* в *неориентированных графах*, то есть связанных между собой областей графа, существует простой алгоритм, использующий непересекающиеся множества.

Связанные компоненты обычно вычисляются с использованием поиска в глубину (Depth First Search, DFS), однако для трассировки компонент можно использовать непересекающиеся множества, заполняя их в процессе сканирования всех ребер графа. Пример показан в листинге 5.11.

Листинг 5.11. Вычисление связанных компонент в графе с помощью непересекающихся множеств

```
disjointSet = new DisjointSet(graph.vertices)
for edge in graph.edges do
  disjointSet.merge(edge.source, edge.destination)
```

Создание нового непересекающегося множества, в котором каждая вершина графа первоначально находится в отдельном разделе

Объединение разделов, в которых находятся вершины, связанные ребром

Обход ребер в графе

В конечном итоге каждый раздел с вершинами в `disjointSet` будет содержать связанные компоненты.

Стоит отметить, что этот алгоритм нельзя использовать для ориентированных графов и тесно связанных компонент.

5.7.2. Графы: алгоритм Краскала для минимального остовного дерева

Остовное дерево связного неориентированного *графа* `G` — это дерево, узлы которого являются вершинами графов, а ребра — подмножеством ребер `G`. Если `G` связан, то он обязательно имеет хотя бы одно остовное дерево, но их может быть больше, если граф имеет циклы (рис. 5.13).

Среди всех возможных *минимальным остовным деревом* (Minimum Spanning Tree, MST) является дерево с минимальной суммой весов ребер.

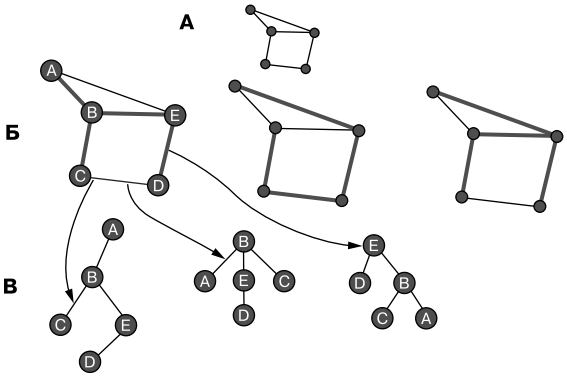


Рис. 5.13. Пример графа с несколькими остовными деревьями. *А*. Неориентированный связный граф с циклами. *Б*. В графе есть циклы, соответственно, существует несколько остовных деревьев, охватывающих все узлы. Здесь показано несколько примеров, в каждом из которых присутствует минимальный набор ребер (выделены жирными линиями), «охватывающих» все вершины. *В*. Для каждого набора ребер можно получить несколько деревьев в зависимости от корня дерева и порядка следования потомков. (Здесь показаны лишь некоторые примеры. Но обратите внимание, что деревья могут быть не только двоичными)

Обсуждение алгоритма Краскала выходит за рамки этой книги, поэтому просто отмечу, что он строит MST для графа по такой формуле.

1. Первоначально вершины находятся в отдельных множествах.
2. Создаются непересекающиеся множества для вершин графа.
3. Выполняется обход ребер графа в порядке возрастания весов.
4. Если вершины, соединяемые ребром, находятся в разных разделах, то они объединяются в один раздел.
5. Если все вершины оказались в одном разделе, алгоритм останавливается.

MST определяется списком ребер, который инициирует в пункте 4 операции слияния непересекающихся множеств.

5.7.3. Кластеризация

Кластеризация часто используется в роли обучения без учителя[1]. Задача состоит в том, чтобы получить разбиение множества точек на несколько обычно не пересекающихся подмножеств, как показано на рис. 5.14.

Существует несколько типов алгоритмов кластеризации. Обсуждение классификации алгоритмов выходит за рамки этой главы (оно будет иметь место в главе 12), тем не менее отмечу один конкретный класс этих алгоритмов.

---

[1] Машинное обучение без учителя связано с осмыслением «немаркированных» данных (то есть данных, которые не были классифицированы) с целью поиска в них структуры.

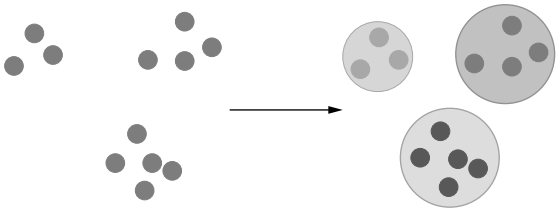


Рис. 5.14. Пример кластеризации. *Слева* — набор исходных двумерных точек. У нас нет дополнительной информации о точках или об отношениях между ними. После кластеризации *(справа)* можно сделать вывод о некоторых отношениях между точками. В частности, точки оказались сгруппированными в три подмножества, внутри которых они, как кажется, демонстрируют более высокую корреляцию

Агломеративная иерархическая кластеризация начинается с каждой точки, находящейся в своем отдельном кластере (разделе), и непрерывно объединяет точки (и их кластеры) попарно, пока все кластеры не окажутся слиты в один. На рис. 5.15 показан пример, как это работает. Алгоритм хранит историю этого процесса, и на любом его шаге можно получить снимок созданных кластеров. Точный момент, когда делается снимок, контролируется несколькими параметрами и определяет результат работы алгоритма.

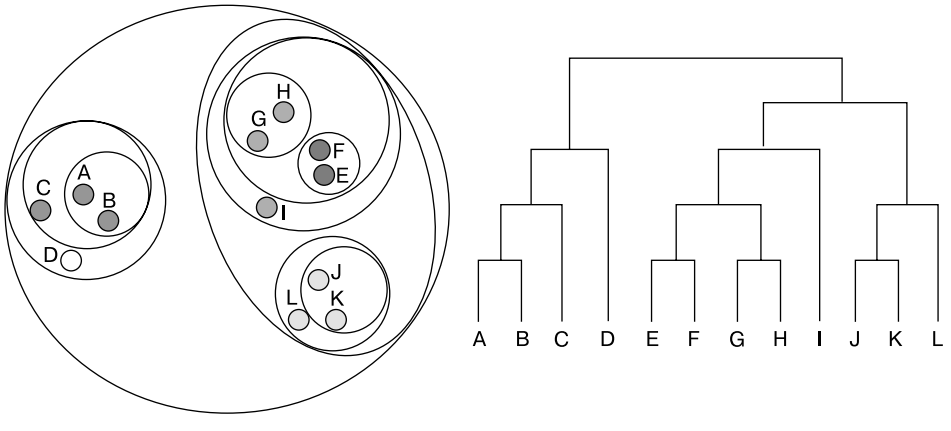


Рис. 5.15. Пример иерархической кластеризации. *Слева* показаны набор данных (двумерных точек) и последовательность их группировки в виде эллипсов. Из рисунка можно сделать вывод, что, например, A и B группируются до того, как C добавится к ним, чтобы сформировать больший кластер. Следовательно, связь между A и B предполагается более сильной, чем связь между A и C или B и C. *Справа* тот же процесс показан с использованием дендрограммы[1]. Обратите внимание, что оба рисунка могут быть результатом агломеративной или разделительной кластеризации: первая создает дендрограмму, начиная снизу, вторая — начиная с вершины

[1] Дендрограмма (от греч. dendro — «дерево» и gramma — «рисунок») — это древовидная диаграмма, используемая для иллюстрации расположения кластеров, полученных в результате иерархической кластеризации.

Описание алгоритма должно насторожить: на каждом шаге нужно найти две точки, принадлежащие двум разным кластерам. Легко представить, какая структура данных лучше всего подходит для вычисления и поиска этой информации. В главе 13 будет описано практическое применение непересекающихся множеств как части алгоритма распределенной кластеризации.

5.7.4. Унификация

Унификация — это процесс решения уравнений в форме символьных выражений. Один из способов решения таких уравнений — поиск эквивалентных членов в обеих частях уравнения и их удаление.

Конечно, стратегии решения зависят от того, какие выражения (члены) могут встречаться в уравнениях и как их сравнивать (или когда их можно считать равными). Например, они могут считаться равными, если имеют одинаковые значения, или же равными могут считаться символические выражения, если они эквивалентны, возможно, за вычетом некоторой замены переменных.

Очевидно, что непересекающиеся множества идеально подходят для реализации высокопроизводительных алгоритмов, решающих эту задачу.

РЕЗЮМЕ

- Непересекающиеся множества позволяют создавать все более сложные и эффективные решения простым добавлением небольших дополнительных изменений.
- Иногда можно согласиться на неоптимальную реализацию, если она достаточно эффективна, а производительность некритична.
- В некоторых случаях можно даже согласиться на простейшее решение с линейным временем выполнения, но это настолько фундаментальная основа многих алгоритмов работы с графами, что ее нужно максимально оптимизировать.
- Нам известна нижняя теоретическая граница времени выполнения операций с непересекающимся множеством, но мы не знаем, существует ли алгоритм, преодолевающий эту границу, или хотя бы какой-то другой алгоритм, более быстрый, чем уже известные.
- *Обратная функция Аккермана*, значение которой не превышает 5 для любого целого числа, которое можно представить на компьютере, отображает порядок величины времени выполнения операции слияния непересекающихся множеств. Для слияния двух подмножеств в среднем потребуется не более пяти замен.

6 Префиксные деревья, компактные префиксные деревья: эффективный поиск строк

В этой главе

- ✓ Уникальные особенности работы со строками.
- ✓ Префиксные деревья как способ эффективного поиска и индексации строк.
- ✓ Компактные префиксные деревья как продвинутая модель префиксных деревьев с эффективным использованием памяти.
- ✓ Использование префиксных деревьев для решения задач, связанных со строками.
- ✓ Использование префиксных деревьев для реализации эффективной проверки орфографии.

Приходилось ли вам в спешке отправлять электронное письмо или сообщение в мессенджер и через секунду замечать опечатку? Со мной такое случалось много раз! Однако в последнее время в почтовых клиентах и браузерах у нас появился ценный союзник: программы проверки орфографии! Если вам интересно узнать больше о том, как они работают и как их эффективно реализовать, то эта глава послужит хорошей отправной точкой.

В главе 3 были описаны сбалансированные деревья. Они предлагают наилучший вариант контейнеров, идеально подходящих для эффективного хранения динамически изменяющихся данных, в которых часто необходимо выполнять поиск. В приложении В описываются и сравниваются имеющиеся у нас варианты контейнеров,

обеспечивающие быстрый поиск, быструю вставку или быстрое удаление. Так вот: деревья предлагают наилучший компромисс между всеми этими операциями.

Сбалансированные деревья, в частности, гарантируют логарифмическое время выполнения всех основных операций в худшем случае. В общем случае, когда ничего не известно о данных, которые нужно хранить и (позже) искать, это действительно лучшие структуры данных.

Но являются ли они столь же эффективными, если заранее известно, что в контейнере будут храниться только определенные типы данных? Оказывается, иногда, когда имеется больше информации о данных, как, например, в этом случае, можно использовать специализированные алгоритмы, превосходящие универсальные.

Возьмем, к примеру, сортировку. Если известно, что в роли ключей применяются целые числа из ограниченного диапазона, можно использовать алгоритм *RadixSort*, обладающий производительностью выше линейно-логарифмической и игнорирующий нижнюю границу сортировки путем сравнения[1].

Точно так же, если известно, что программа должна сортировать строки, можно использовать специализированные алгоритмы, такие как *трехсторонняя быстрая сортировка строк*. Подобные алгоритмы оптимизированы для такого типа данных и работают лучше, чем обычный алгоритм быстрой сортировки (или любой алгоритм сортировки на основе сравнения).

В этой главе мы проанализируем конкретный подкласс контейнеров — строковые контейнеры и выясним, как оптимизировать их в отношении потребления памяти и времени выполнения. Для этого введем новые специализированные структуры данных: префиксные деревья и компактные префиксные деревья. Затем используем их для реализации эффективной проверки орфографии.

6.1. ПРОВЕРКА ОРФОГРАФИИ

Но сначала сформулируем задачу, которую будем решать: проверку орфографии.

Эта задача не особенно нуждается в представлении, верно? Хотелось бы реализовать программное обеспечение, принимающее слова и возвращающее `true` или `false`, в зависимости от того, является ли входное слово допустимым для английского языка[2] или нет соответственно.

---

[1] Была доказана невозможность сортировки списка из n элементов менее чем за $O(n \log(n))$ операций, если используемый метод основан исключительно на сравнениях. RadixSort способен достигнуть $O(n \log(k))$ в сортировке n целых чисел, которые могут принимать любое из k возможных значений. Для больших значений k, когда $k \sim= n$, алгоритм ведет себя не лучше, чем общий линейно-логарифмический.

[2] Разумеется, можно написать программу проверки орфографии для любого языка; здесь выбран английский просто из-за удобства.

Формулировка получилась немного расплывчатой, и это оставляет открытой дверь для многих (в том числе и неэффективных) решений. Попробуем уточнить контекст, чтобы прояснить требования. Предположим, мы разрабатываем для социальной сети новый клиент со строгими ограничениями на потребляемые ресурсы (например, для мобильной ОС). При этом проверка вводимой информации должна выполняться оперативно и выдавать классическое красное волнистое подчеркивание слов с ошибками. Каждый раз, когда заканчивается набор слова (то есть каждый раз, когда набирается разделитель слов, например пробел, запятая и т. д.), программа-клиент должна проверить наличие опечаток.

Из-за ограничений на потребление ресурсов важно, чтобы проверка орфографии выполнялась без задержек; необходимо максимально уменьшить влияние проверки с точки зрения потребления процессора и памяти.

Кроме того, желательно, чтобы программа проверки орфографии могла обучаться — например, чтобы пользователи могли добавлять в исключения свои имена или названия своих городов/клубов/имена артистов и т. д.

6.1.1. Прнцесса, Деймон и эльф звходят в бар

Заметили опечатки в заголовке этого раздела?[1] Опечатки только и ждут, когда вам понадобится в спешке отправить сообщение (независимо от используемого носителя), и, к сожалению, после его отправки обратного пути не будет — вы не сможете ничего исправить[2].

Вот почему необходимы средства проверки орфографии, ясно выделяющие опечатки, и в настоящее время такие инструменты встроены даже в браузеры.

В конце концов, средства проверки орфографии имеют довольно простую организацию: это обертка вокруг *словаря*[3], а метод клиента, проверяющий орфографию, просто вызывает метод `contains` контейнера и, если получает отрицательный результат, добавляет визуальное выделение, чтобы показать ошибку.

Наш контейнер тоже будет иметь простой API. Это универсальный контейнер, поддерживающий поиск, по аналогии с API двоичных деревьев поиска или рандомизированных декартовых деревьев (см. раздел 3.4).

Вы уже можете подумать о том, как реализовать этот контейнер с помощью знакомых нам инструментов. Например, для поиска в статическом наборе можно выбрать хеш-таблицу, а при жестких требованиях к потреблению памяти и допустимости

---

1 У вас в голове зазвенел звоночек? Все правильно...

2 Впрочем, твиты можно удалить, а если вы будете достаточно быстры, то их отправку никто даже не заметит.

3 Здесь под этим термином понимается абстрактная структура данных, называемая словарем, которая, кстати, используется для моделирования цифрового эквивалента реальных словарей. Подробнее о словарях рассказывается в главе 4.

некоторой неточности можно даже взять за основу фильтр Блума. Однако это не наш случай, потому что необходимо обеспечить поддержку открытого динамического набора, а, как известно, единственной структурой данных, обеспечивающей наилучший компромисс для всех операций, является дерево.

Простого двоичного дерева, возможно, вполне достаточно для поддержки всех операций, предоставляемых словарями. На рис. 6.1 показано, как могут выглядеть такие деревья. Здесь демонстрируется только небольшая часть поддерева, содержащая несколько похожих слов (чуть ниже я объясню, почему это важно).

Насколько быстрыми могут быть операции с таким деревом? Предполагаем, что дерево сбалансировано, значит, его высота будет логарифмически пропорциональна количеству содержащихся в нем слов. Поэтому каждому методу — `contains`, `insert`, `remove` и т. д. — потребуется обойти в среднем (и не более) $O(\log(n))$ узлов.

Однако до сих пор, анализируя деревья, мы предполагали, что роль ключей в них играют целые числа или другие типы данных, которые можно проверить за постоянное время и которые занимают постоянный объем памяти.

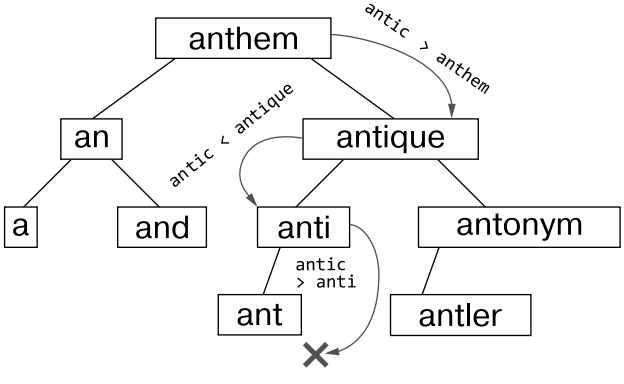


Рис. 6.1. Двоичное дерево поиска, хранящее часть словаря, и шаги, которые необходимо выполнить, чтобы отыскать слово "antic". В этом примере поиск возвращает отрицательный результат, потому что искомое слово отсутствует в показанном дереве

Для строк это предположение не соответствует действительности. Каждый узел должен иметь возможность хранить строку неограниченной длины, поэтому общий объем памяти, занимаемый деревом, определяется суммой всех длин ключей:

$$E\left[S(n)\right] = E\left[\sum_{i=0}^{n-1}\left|k_i\right|\right] = \sum_{i=0}^{n-1}E\left[k_i\right] \approx \sum_{i=0}^{n-1}L = nL.$$

Если предположить, что средняя длина строк в дереве равна L, то ожидаемое значение $S(n)$ — объем памяти, необходимый для хранения дерева, — пропорционален $n \times L$. Если максимальную длину строк обозначить как m, то $S(n) = O(n \times m)$ является строгой верхней границей для наихудшего случая.

Точно так же, взявшись оценивать время выполнения метода `search`, нельзя игнорировать длину строк. Например, вызов `search("antic")` в дереве на рис. 6.1 начнет поиск с корня и сравнит `"antic"` с `"anthem"`, что повлечет за собой сравнения по крайней мере четырех символов, перед тем как выяснится, что эти слова не совпадают. Затем метод перейдет к правой ветви и снова сравнит две строки, `"antic"` и `"antique"` (еще пять сравнений символов), и, поскольку они не совпадают, обход продолжится в левом поддереве и т. д.

Следовательно, в худшем случае методу `search` потребуется выполнить `T(n) = = O(log(n)*m)` сравнений.

6.1.2. Сжатие — ключ к успеху

Как показывает этот краткий анализ, дерево не подходит для решения рассматриваемой задачи ни с точки зрения занимаемой памяти, ни с точки зрения производительности. Если внимательно посмотреть на дерево на рис. 6.1, то можно заметить множество ненужных накладных расходов: все слова начинаются с символа `"a"`, но он присутствует во всех узлах поддерева, и на каждом шаге обхода дерева будет участвовать в сравнении с искомым (или вставляемым) текстом.

В пути поиска слова `"antic"` все четыре показанных и проверенных узла имеют один и тот же префикс `"ant"`. Было бы неплохо иметь возможность каким-то образом сжать эти узлы и сохранить общий префикс только один раз с дополняющими его окончаниями в каждом узле.

6.1.3. Описание и API

Структура данных, которая будет представлена в следующем разделе, была создана с учетом этих потребностей. Она способна предложить эффективный способ реализации другой операции: поиска в контейнере всех ключей, начинающихся с одного и того же префикса.

Из примера в предыдущем подразделе уже можно видеть, что, имея возможность хранить общие префиксы строк только один раз, мы могли бы быстро извлекать все строки, начинающиеся с этих префиксов.

В табл. 6.1 показан общедоступный API для абстрактной структуры данных (Abstract Data Structure, ADT), предлагающий все типовые операции с контейнером, а также две новые: извлечение всех строк, начинающихся с определенного префикса, и поиск самого длинного префикса, хранящегося в контейнере.

Из предыдущего примера становится понятно, что хорошим именем для этой структуры данных могло бы быть `PrefixTree`, но `StringContainer` кажется более обобщенным (в отношении абстрактной структуры данных нам все равно, как она реализована — с использованием дерева или какой-либо другой конкретной структуры данных). Это название точнее передает суть контейнера, его назначение для

хранения строк. А факт поддержки поиска по префиксу является почти естественным следствием разработки контейнера для строк.

Таблица 6.1. API и контракт для StringContainer

Абстрактная структура данных	

API	`class StringContainer {`
	`  insert(key)`
	`  remove(key)`
	`  contains(key)`
	`  longestPrefix(key)`
	`  keysStartingWith(prefix)`
	`}`

Контракт с клиентом	Помимо операций, свойственных обычным контейнерам, эта структура позволяет искать самый длинный префикс и возвращать все хранимые в нем строки, начинающиеся с определенного префикса

Теперь, зафиксировав API и описав ADT, которые будут использоваться для решения задачи проверки орфографии, можно углубиться в детали и рассмотреть несколько конкретных структур данных, основываясь на которых можно реализовать это ADT.

6.2. ПРЕФИКСНОЕ ДЕРЕВО

Первая реализация `StringContainer`, которую мы проиллюстрируем, основывается на применении *префиксного дерева*, или *trie*. Кстати, все остальные структуры данных, демонстрируемые в следующих разделах, тоже основаны на префиксных деревьях, поэтому мы не можем начать с чего-то другого.

И первое, что нужно знать о префиксных деревьях, — их название в английском языке — trie — произносится как «трай» («попытка»). Его автор[1], Рене де ла Брианде (René de la Briandais), выбрал этот термин, потому что он похож на tree («дерево»), а также потому, что является частью слова re*trie*val («извлечение»), обозначающего основную операцию этого контейнера; своеобразное произношение этого слова отчасти задумано как каламбур, а отчасти — чтобы избежать путаницы со словом tree.

Первоначально префиксные деревья разрабатывались как компактный и эффективный способ поиска строк в файлах; идея этой структуры данных, как показано в предыдущем разделе, заключается в том, чтобы уменьшить избыточность за счет однократного сохранения общих префиксов.

Этого нельзя достичь с помощью простого двоичного дерева поиска или только двоичного дерева, поэтому потребовался сдвиг парадигмы: де ла Брианде использовал

[1] *Briandais de la R.* File searching using variable length keys. Доклад, представленный 3–5 марта 1959 года на Западной объединенной конференции по вычислительным машинам. ACM, 1959.

n-ичные деревья, в них ребра отмечены всеми символами алфавита, а узлы просто соединяют пути.

У узлов также есть небольшая, но очень важная функция: они хранят информацию о соответствии ключа и пути от корня до текущего узла.

Прежде чем перейти к более формальному описанию, взглянем на рис. 6.2. Здесь изображена структура типичного префиксного дерева, содержащего слова `"a"`, `"an"`, `"at"` и `"I"`.

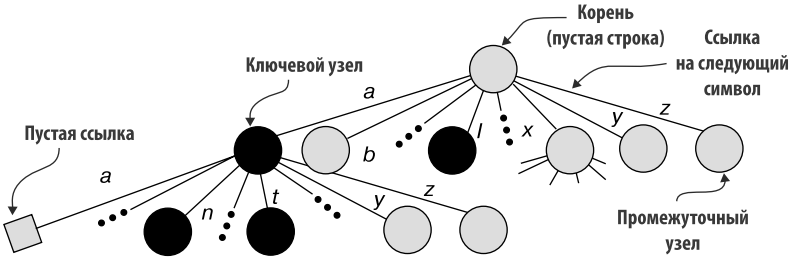


Рис. 6.2. Структура префиксного дерева. Слова кодируются в дереве с помощью ребер, каждое ребро соответствует одному символу, а каждая вершина n связана с одним словом, полученным путем соединения символов в ребрах на пути от корня к n. Корневой узел соответствует пустой строке (поскольку на пути к нему не пройдено ни одного ребра), крайний левый лист соответствует слову "aa" и т. д. Не все пути составляют осмысленные слова, и не все узлы хранят связанные с ними слова. Только черные узлы (называемые ключевыми узлами) отмечают слова, хранящиеся в дереве, а пустые узлы, также известные как промежуточные, соответствуют префиксам слов, хранящимся в дереве. Обратите внимание, что все листья должны быть ключевыми узлами

Если рис. 6.2 кажется непонятным, то знайте, что вы не одиноки в этом непонимании! В классической реализации узлы префиксного дерева имеют по одному ребру для каждого возможного символа алфавита: некоторые из этих ребер указывают на другие узлы, но большинство из них (особенно на нижних уровнях дерева) являются пустыми ссылками[1].

Тем не менее такое конкретное визуальное представление префиксного дерева выглядит ужасно: слишком много ссылок и слишком много узлов, что вызывает ощущение хаоса.

По этой причине будем использовать альтернативное представление, показанное на рис. 6.3, в котором изображаются только ссылки на фактические узлы, без пустых ссылок.

---

[1] Как будет показано ниже, это верно для всех символов *c*, для которых нет суффикса текущего узла, следующим символом которого является *c*.

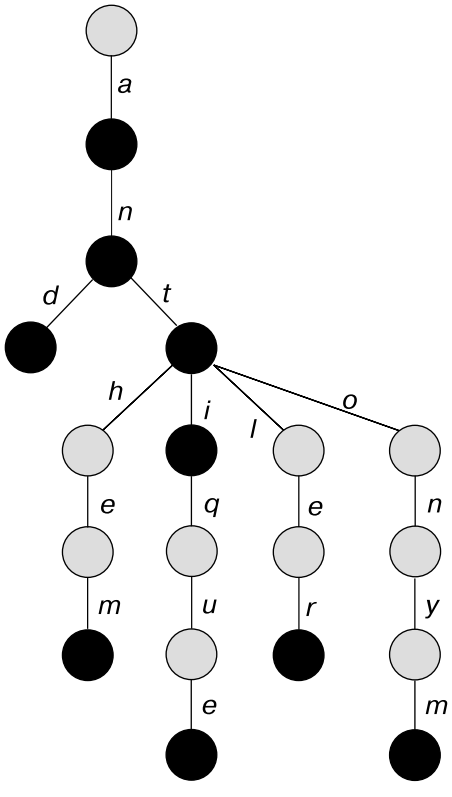


Рис. 6.3. Более компактное представление префиксного дерева. Этот пример содержит те же элементы, что и двоичное дерево поиска на рис. 6.1

Формально для заданного алфавита Σ с $|\Sigma| = k$ символами префиксное дерево является k-ичным[1] деревом, в котором каждая вершина имеет (не более) k потомков, каждый из которых отмечен другим символом из Σ; ссылки на дочерние узлы могут указывать на другой узел или быть пустыми (`null`).

Однако, в отличие от k-ичных деревьев поиска, ни один узел в префиксном дереве не хранит связанный с ним ключ. Символы в префиксном дереве фактически хранятся на границах между узлом и его потомками.

В листинге 6.1 показана возможная реализация класса `Trie` (на объектно-ориентированном псевдокоде); полная реализация доступна в репозитории книги на GitHub[2].

---

[1] Хотя обычно мы говорим об n-ичных деревьях, но в данном случае символ n зарезервирован для обозначения количества элементов в контейнере (или в общем случае размера входных данных). Чтобы избежать путаницы, мы будем использовать k для обозначения размера алфавита и, следовательно, термин «k-ичный».

[2] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#trie.

Листинг 6.1. Класс Trie

```
#type Char[]
Alphabet

class Node
  #type boolean
  keyNode

  #type HashMap<Char, Node>
  children

  function Node(storesKey)
    for char in Alphabet do
      this.children[char] ← null
    this.keyNode ← storesKey

class Trie
  #type Node
  root

  function Trie()
    root ← new Node(false)
```

В простейшей версии узлы префиксного дерева могут содержать только самую малую часть информации — `true` или `false`. Узел `N`, отмеченный значением `true`, называется *ключевым узлом*, потому что последовательность символов на ребрах в пути от корня до `N` соответствует слову, которое фактически хранится в дереве. Узлы со значением `false` называются *промежуточными узлами*, потому что соответствуют промежуточным символам, составляющим одно или несколько слов, хранящихся в дереве.

Как видите, в префиксных деревьях отсутствует обычная двойственность между листьями и внутренними узлами, зато появляется другое (ортогональное) различие. Однако, оказывается, все листья в правильно сформированном минимальном дереве являются ключевыми узлами: как будет показано далее, лист не имеет смысла в качестве промежуточного узла.

Тот факт, что слова определяются путями, означает, что все потомки узла имеют общий префикс: путь от корня до их общего родителя. Например, на рис. 6.3 можно увидеть, что все узлы имеют общий префикс `"an"` и все узлы, кроме одного, имеют общий префикс `"ant"`. Эти два слова, кстати, тоже хранятся в дереве, потому что узлы в конце путей к ним являются ключевыми.

Корень во всех отношениях выступает промежуточным узлом, связанным с пустой строкой; он мог бы быть ключевым узлом, только если бы пустая строка принадлежала корпусу, содержащемуся в дереве.

Для проверки орфографии было бы вполне достаточно хранить логические значения в каждом узле (в конце концов, нам нужно только знать, присутствует ли слово в словаре), но префиксные деревья часто используются для хранения или индексации слов в текстах. В таких случаях нужно знать либо количество вхождений слова в тексте,

либо позиции, в которых они появляются. В первой ситуации можно хранить счетчик в каждом узле, и только ключевые узлы будут иметь положительное значение. В последней ситуации можно хранить список позиций вхождений.

6.2.1. Почему снова лучше?

Сначала оценим объем занимаемой памяти: почему префиксное дерево на рис. 6.3 лучше двоичного дерева поиска на рис. 6.1?

Сделаем некоторые арифметические прикидки! Но прежде оговорим условия, они приведены в списке.

- Учитываться будут только строки и символы ASCII, поэтому будем считать, что для хранения каждого символа требуется всего 1 байт (при использовании Юникода ситуация не сильно изменится; скорее, даже получится некоторая экономия, этот вариант будет самым дешевым) плюс (в двоичном дереве поиска) 1 байт для пустого символа, завершающего каждую строку.
- В префиксном дереве явно хранятся только ссылки на фактические узлы и фиксированное количество байтов для пустых ссылок в двоичном дереве поиска (ровно столько же памяти занимают непустые ссылки).
- Как уже упоминалось, каждый узел в префиксном дереве имеет $|\Sigma|$ ссылок, где $|\Sigma|$ — размер алфавита. Это означает, что в префиксных деревьях, и особенно в узлах на нижних уровнях, большинство ссылок являются пустыми. Действительно, в листинге 6.1 можно увидеть, как все эти ссылки инициализируются в конструкторе `Node`.
- У каждого узла в префиксном дереве нужно учесть фиксированный объем памяти для хранения списка потомков (можно представить, что все ссылки хранятся в хеш-таблице), а также переменный объем, зависящий от количества фактических потомков.
- Каждая ссылка в двоичном дереве занимает 8 байт (64-битные ссылки), а каждая ссылка в префиксном дереве занимает 9 байт (8 для ссылки плюс 1 для символа, с которым она связана).
- Для каждого узла в двоичном дереве поиска потребуется столько байтов, сколько символов содержится в его ключе, плюс 4 байта[1] для самого объекта `Node`.
- Каждому узлу в префиксном дереве требуется 1 бит (для хранения логического значения) плюс такой же постоянный объем, как и для двоичного дерева поиска; округлим до 5 байт.

С учетом этих предпосылок можно заметить, что двоичное дерево поиска (BST) на рис. 6.1 имеет 9 узлов (ключами которых являются строки) и, следовательно, $2 \times 9 = 18$ ссылок, тогда как префиксное дерево (trie) на рис. 6.3 имеет 19 узлов

[1] Это количество взято совершенно произвольно; реальные объекты в языках программирования могут занимать больше 4 байт (например, в Java или C++ от 8 до 16 байт).

и 18 ссылок. Корневой узел BST содержит ключ `"anthem"` и занимает 27 байт (4 для самого узла, 7 для строки, 2 × 8 для ссылок). Точно так же его левый потомок с ключом `"an"` занимает 23 байта. Проще говоря, размер вычисляется как 21 байт на узел плюс длина строки. Для всего дерева, учитывая, что оно имеет 9 узлов и занимает в общей сложности 47 байт для ключей, нужно 227 байт.

Теперь проверим префиксное дерево: каждый узел занимает 5 байт, а каждая ссылка 9 байт — всего 257 байт.

Таким образом, этой версии префиксного дерева может потребоваться чуть больше памяти, чем соответствующему двоичному дереву поиска. Рассмотренные величины зависят от многих факторов, в первую очередь от накладных расходов на хранение объектов. Поскольку в этом префиксном дереве больше узлов, то и получается, что чем больше накладные расходы на хранение объектов, тем больше будет разница в размере используемой памяти.

Однако очевидно, что форма дерева и фактическое количество узлов играют важную роль. В примере на рис. 6.3 ключи хранят только короткие общие префиксы. Оказывается, префиксные деревья более эффективны при хранении ключей с длинными общими префиксами. На рис. 6.4 показано, как в этих случаях баланс может сместиться в пользу префиксных деревьев. На рис. 6.4 показано, что большинство узлов в префиксном дереве — черные (ключевые узлы), и когда соотношение ключевых узлов и промежуточных более точное, выше становится и эффективность потребления памяти префиксным деревом. Так происходит потому, что, когда в одном пути имеется больше одного ключевого узла, в этом пути хранится не меньше двух слов (одно из которых является префиксом другого). В двоичном дереве поиска для этих двух слов потребуется два узла, хранящих полные строки.

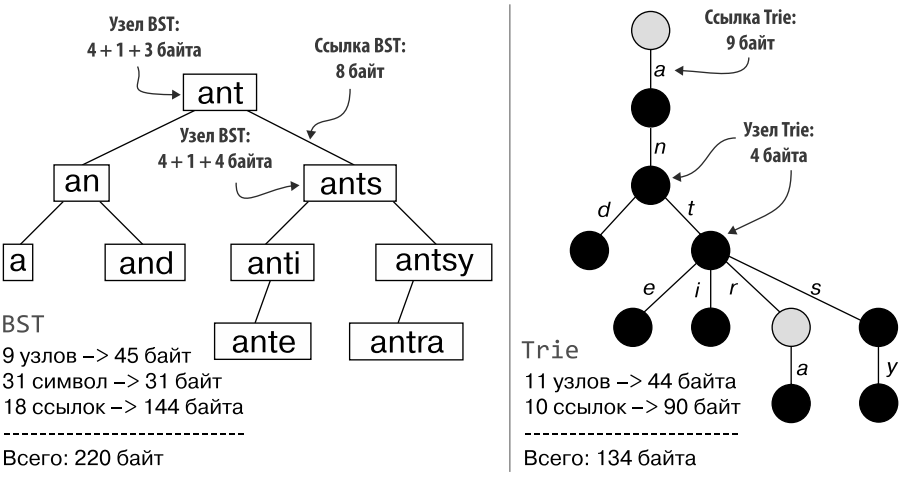


Рис. 6.4. Сравнение подходов на основе двоичных деревьев поиска и префиксных деревьев. Чем больше общих символов хранится в ключах в префиксном дереве (то есть чем длиннее общий префикс), тем эффективнее префиксные деревья в смысле потребления памяти

Еще одним признаком эффективного расходования памяти является ветвление глубоких узлов. В такой ситуации префиксное дерево «сжимает» пространство, необходимое для хранения двух строк с общим префиксом, сохраняя общий префикс только один раз.

В итоге, когда во втором примере сохраняется всего девять слов, при тех же предположениях, что и выше, разница составляет 220 байт против 134, то есть префиксное дерево экономит почти 40 % пространства. Если причислить к накладным расходам на узлы по 8 байт, то разница составит 256 байт против 178, а экономия — около 30 %, что все равно выглядит впечатляюще. Для больших деревьев со словарями или индексами длинных текстов могут потребоваться сотни мегабайтов.

На рис. 6.4 показан наилучший сценарий для префиксных деревьев, но очевидно, что ситуация не всегда складывается так хорошо. На рис. 6.5 можно оценить другой пограничный случай, близкий к наихудшему сценарию, когда самым длинным префиксом в (вырожденном) префиксном дереве является пустая строка. В подобных ситуациях информация хранится очень неэффективно; к счастью, однако, вероятность возникновения таких пограничных случаев в реальных приложениях чрезвычайно мала.

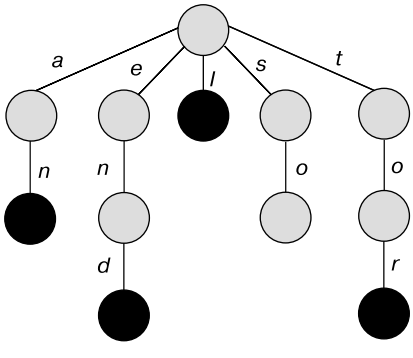


Рис. 6.5. Вырожденное префиксное дерево, в котором ни одна строка не имеет общего префикса с другой строкой

Такое префиксное дерево крайне неэффективно расходует память. Итак, в худшем случае префиксные деревья сопоставимы с двоичными деревьями поиска по потреблению памяти.

А что можно сказать о времени работы? Ответ на этот вопрос отложим до рассмотрения отдельных методов, чтобы было понятнее, откуда именно берутся параметры и результаты.

6.2.2. Поиск

Начнем с поиска. Допустим, у нас есть правильно сконструированное префиксное дерево. Как проверить, содержит ли оно определенный ключ?

Оказывается, найти что-то в префиксном дереве проще, чем в двоичном дереве поиска. Основное различие деревьев состоит в том, что обходить префиксное дерево

приходится по одному символу (искомого ключа), каждый раз следуя по ссылке, отмеченной очередным символом.

И строки, и префиксные деревья — это рекурсивные структуры, единицей итерации в которых является одиночный символ; каждую строку на самом деле можно описать так:

- пустая строка, `""`;
- конкатенация символа `c` и строки `s'`: `s=c+s'`, где `s'` — строка на один символ короче, чем `s`, и может быть пустой строкой.

ПРИМЕЧАНИЕ

В большинстве языков программирования одинарные кавычки не допускаются в именах переменных, поэтому в листинге 6.2 вместо s' мы используем имя tail, обозначающее конец текущей строки s на рисунках.

Например, строка `"home"` состоит из символа `"h"`, соединенного со строкой `"ome"`, которую можно представить как `"o" + "me"`, и т. д., пока не дойдем до символа `"e"`, который можно записать как символ `'e'`, объединенный с пустой строкой `""`.

Префиксное дерево, в свою очередь, хранит строки как пути от корня до ключевых узлов. Префиксное дерево `T` можно описать как корневой узел, связанный (не более чем) с $|\Sigma|$ более коротких префиксных деревьев. Если префиксное поддерево `T'` соединено с корнем ребром, отмеченным символом `c` ($c \in \Sigma$), то для всех строк `s` в `T'` конкатенация `c + s` принадлежит `T`.

Например, на рис. 6.3 корень имеет только одно исходящее ребро, отмеченное буквой `"a"`. Если рассматривать `T'` как единственное префиксное поддерево корня, то `T'` содержит слово `"n"`, а это означает, что `T` содержит `"a" + "n" = "an"`.

Поскольку и строки, и префиксные деревья являются рекурсивными структурами, было бы естественно реализовать метод поиска рекурсивно. То есть достаточно рассмотреть только случай поиска первого символа строки `s=c+s'`, начиная с корня `R` префиксного дерева `T`. Если `c`, первый символ `s`, совпадает с выходом `R`, то можно продолжить поиск `s'` в префиксном (под)дереве `T'`. Предполагается, что корень поддерева не содержит `null` (как мы вскоре увидим, это вполне обоснованное предположение).

Если в какой-то момент `s` окажется пустой строкой, то это будет обозначать, что весь путь, соответствующий `s`, пройден. Останется только проверить текущий узел, чтобы убедиться, что строка `s` хранится в дереве.

Если вместо этого будет достигнут узел, не имеющий исходящего ребра, совпадающего с текущим символом `c`, то можно будет с уверенностью утверждать, что строка `s` отсутствует в префиксном дереве.

Эти примеры проиллюстрированы на рис. 6.6 и 6.7, но сначала мы рассмотрим реализацию метода поиска, на который затем будем ссылаться при их обсуждении.

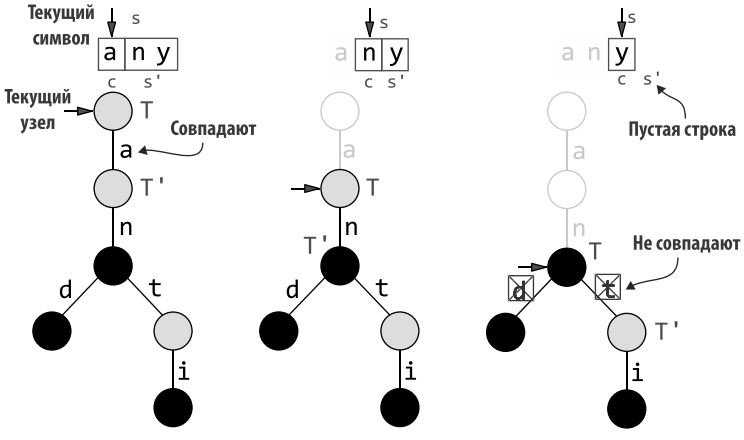


Рис. 6.6. Неудачный поиск в дереве. На каждом шаге искомый ключ со строкой s разбивается на c + s′ — конкатенацию ее первого символа и остальной части строки. Затем первый символ сравнивается с исходящими ребрами текущего узла, и, если совпадение обнаруживается, обход префиксного дерева продолжается. В этом примере поиск завершается ошибкой, потому что последний символ в строке не соответствует ни одному исходящему ребру

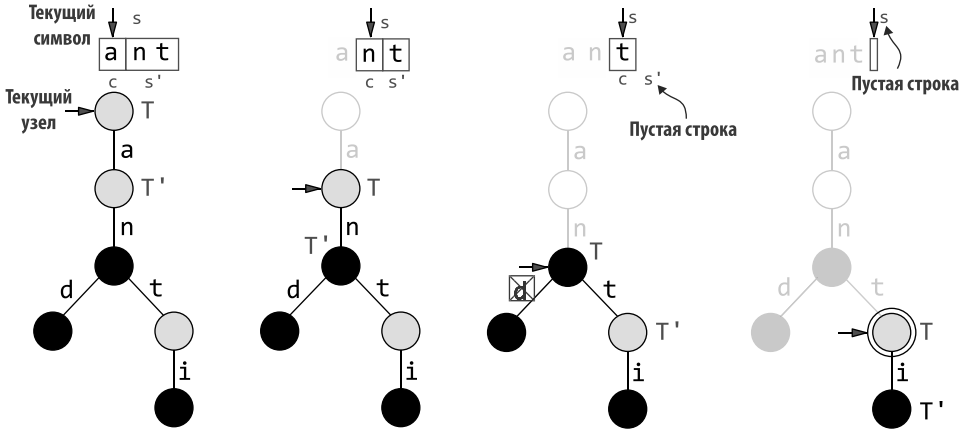


Рис. 6.7. Еще один пример неудачного поиска в префиксном дереве. Здесь поиск завершается ошибкой, потому что путь, соответствующий искомой строке, заканчивается промежуточным узлом

В листинге 6.2 показана рекурсивная реализация метода поиска. В каждом вызове он проверяет совпадение первого символа в искомой подстроке с исходящим ребром

текущего узла, а затем рекурсивно выполняет поиск конца текущей строки в поддереве, на которое ссылается это ребро. На рис. 6.6 и 6.7 показана параллель между движением «вправо» в искомой строке и перемещением вниз по префиксному дереву.

Разумеется, этот метод можно также реализовать с применением цикла. Представленная здесь реализация лучше подходит для случаев, когда имеется компилятор, поддерживающий оптимизацию хвостовой рекурсии[1]. Но, как отмечается в приложении Д и в главе 3, если вам не нравится рекурсия или вы не уверены, что компилятор применит оптимизацию хвостового вызова, то я бы посоветовал создать итеративные версии этих методов, чтобы избежать переполнения стека.

В листинге 6.3 показан метод поиска из открытого API префиксного дерева, который вызывает метод из листинга 6.2. Для других операций мы не будем демонстрировать подобные методы-обертки, если они окажутся такими же тривиальными, как `search`.

Как уже упоминалось, поиск может завершиться неудачей в двух случаях: первый, показанный на рис. 6.6, — когда встречается узел, не имеющий исходящего ребра для следующего символа. В примере с вызовом `trie.search("any")` это происходит, когда очередь доходит до последнего символа `'y'` (как показано на правой диаграмме на рис. 6.6). В листинге 6.2 это соответствует условию в строке 5, возвращающему `true`.

Листинг 6.2. Метод search

```
function search(node, s)
  if s == "" then
    return node.keyNode
  c, tail ← s.splitAt(0)
  if node.children[c] == null then
    return false
  else
    return search(node.children[c], tail)
```

Поиск search принимает узел префиксного дерева и искомый строковый ключ s. Возвращает true, если s имеется в дереве, и false в противном случае. Предполагается, что в аргументе node никогда не будет передано значение null, что является вполне обоснованным предположением, если этот метод реализован как приватный, вызываемый методом поиска из общедоступного API префиксного дерева

Проверяется, пустая ли искомая строка. Если да, то, учитывая рекурсивную природу метода, можно утверждать, что пройден весь путь в префиксном дереве

Достигнут целевой узел для заданного ключа, однако утверждать, что искомая строка присутствует в префиксном дереве, можно, только если текущий узел является ключевым

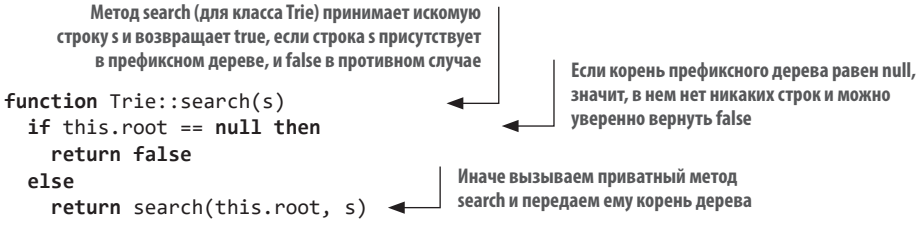
Поскольку s не пустая строка, ее можно разбить на голову, символ c (первый символ в s) и хвост с остальной частью строки

Иначе рекурсивно продолжаем поиск tail в поддереве с корнем в children[c]

Если текущий узел не имеет исходящего ребра для символа c, то это означает, что дальнейшее движение вниз по дереву невозможно и s отсутствует в префиксном поддереве с корнем в node

[1] Как объясняется в приложении Д, если рекурсивный вызов является последней операцией в рекурсии, то компиляторы могут оптимизировать целевой машинный код, реорганизовав его с применением циклов вместо вызовов функций. Но имейте в виду, что не все компиляторы поддерживают эту оптимизацию.

Листинг 6.3. Метод Trie::search




```
function Trie::search(s)
  if this.root == null then
    return false
  else
    return search(this.root, s)
```

Второй случай, когда поиск может завершиться неудачей, — если `search` рекурсивно вызывается для пустой строки после выбора подходящего исходящего ребра текущего узла. Это означает, что достигнут узел дерева, путь которого от корня описывает искомый ключ. Например, как показано на рис. 6.7, метод `search` обходит дерево ссылка за ссылкой и проверяет ключ символ за символом, пока условие в строке 2 (листинг 6.2) не вернет `true` и выполнение не будет передано в строку 3.

Результат строки 3 — единственная разница между успешным и неудачным поиском. Посмотрите пример на рис. 6.7: успешный поиск слова `"ant"` проделает те же самые шаги, с той лишь разницей, что конечный узел (обозначенный буквой `T` на диаграмме справа) должен быть ключевым узлом.

Обратите внимание, что в листинге 6.3 можно избежать проверки наличия префиксного дерева или обработки корня как особого случая, потому что в конструкторе префиксного дерева (см. листинг 6.1) корень создается как пустой узел. При условии тщательной реализации всех методов это лишний раз доказывает правильность нашего предположения о том, что аргумент `node` в вызове `search` (и всех других методов) никогда не будет равен `null`.

Метод `search` — наиболее важный метод для префиксных деревьев, потому что на нем основываются все остальные методы. Поиск настолько важен для этих реализаций, что в листинге 6.4 представлен вариант `searchNode`, возвращающий найденный узел, а не просто `true` или `false`. В следующих разделах будет показано, как можно его использовать для реализации метода `remove`.

Листинг 6.4. Метод searchNode

```
function searchNode(node, s)
  if s == "" then
    return node
  c, tail ← s.splitAt(0)
  if node.children[c] == null then
    return null
  else
    return search(node.children[c], tail)
```

Пришло время задуматься, насколько быстро работает `search`. Количество выполняемых рекурсивных вызовов ограничено меньшим из двух значений: максимальной высотой дерева и длиной искомой строки. Последняя обычно меньше первой, но

в любом случае можно утверждать, что для строки длиной m будет сделано не более $O(m)$ вызовов, независимо от количества ключей, хранящихся в префиксном дереве.

Тогда главным становится вопрос: сколько времени занимает каждый шаг? Как оказывается, это время определяется тремя факторами:

- стоимостью сравнения двух символов: можно предположить, что она равна $O(1)$;
- стоимостью поиска следующего узла: для алфавита Σ размером k она может быть в зависимости от реализации:
  - константой (амортизированной или в худшем случае[1]), $O(1)$, при использовании хеш-таблиц для ребер;
  - логарифмической в худшем случае, $O(\log(k))$, при использовании сбалансированного дерева;
  - линейной в худшем случае, $O(k)$, при использовании простых массивов.

  В большинстве случаев можно принять амортизированное постоянное время;
- стоимостью перехода по ссылке и разделения строки на голову и хвост. Вот здесь нужно быть очень осторожными. Применение простейшего решения для извлечения подстроки в каждом узле приведет к снижению производительности в большинстве языков программирования. Строки обычно реализуются как неизменяемые объекты, поэтому для извлечения подстроки в каждом вызове потребуются линейное время и дополнительная память $O(m)$. К счастью, эта задача решается просто: в рекурсивный вызов можно передать ссылку на начало строки и индекс следующего символа, и в таком случае эту операцию можно считать $O(1)$.

Поскольку так можно реализовать каждый вызов, чтобы добиться амортизированного постоянного времени, то весь поиск занимает $O(m)$ амортизированного времени выполнения.

6.2.3. Вставка

Как и `search`, метод `insert` проще определить с использованием рекурсии. В нем возможны два разных сценария:

- в префиксном дереве уже есть путь, соответствующий вставляемому ключу, и в этом случае нужно только изменить последний узел в пути, сделав его ключевым (или, если индексируется текст, добавить новую запись в список индексов для слова);
- в префиксном дереве есть путь только для подстроки ключа, и тогда нужно добавить новые узлы.

---

[1] Множество ключей в хеш-таблице, то есть алфавит, является статическим и известно заранее, поэтому можно использовать идеальное хеширование и получить постоянное время поиска в худшем случае. Подробности ищите в приложении B.

На рис. 6.8 показан пример рассмотренной ранее ситуации, здесь вызов `insert("anthem")` для префиксного дерева с рис. 6.7 приведет к добавлению новой ветви к одному из листьев.

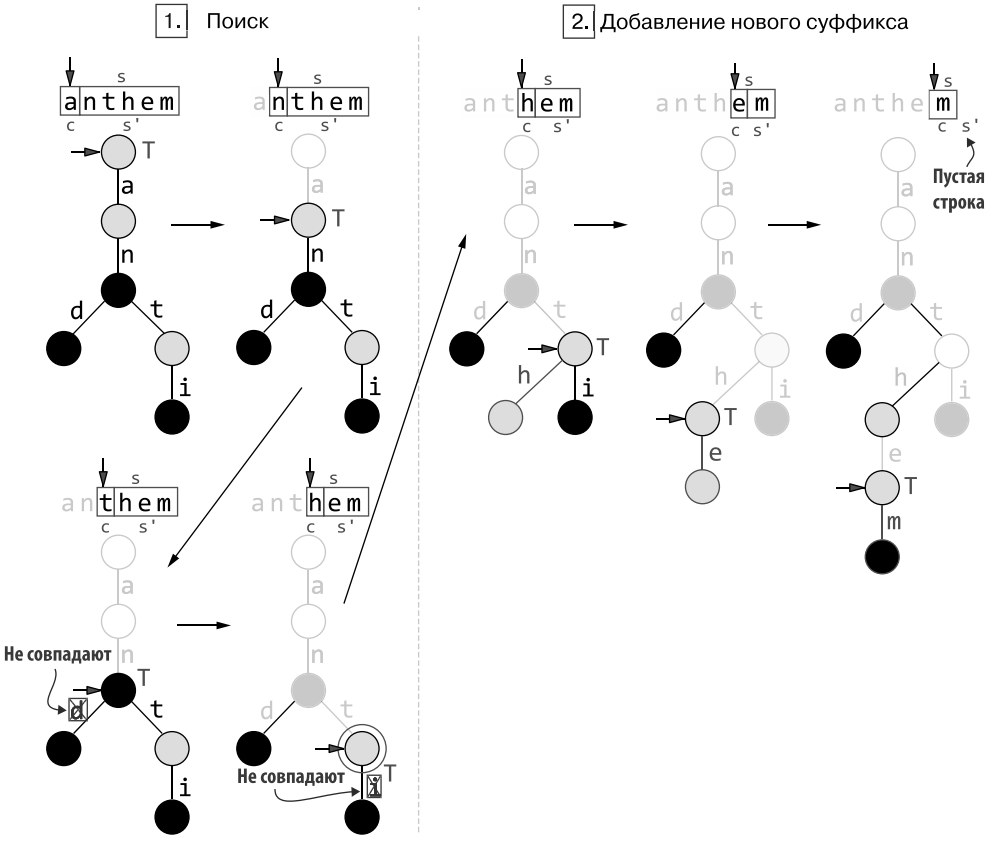


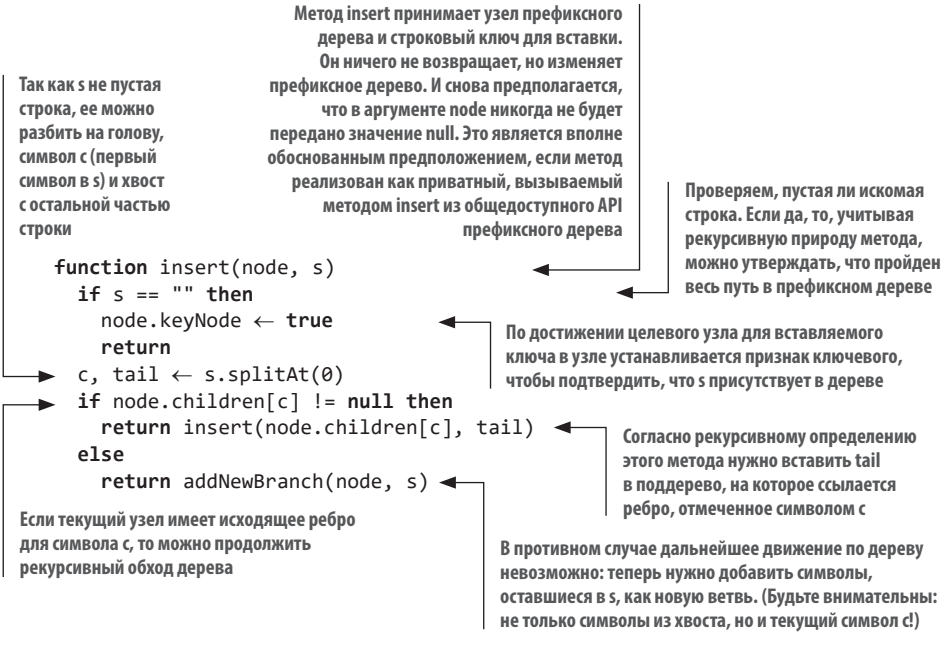
Рис. 6.8. Пример метода insert. Вызов trie.insert("anthem") сначала отыскивает самый длинный префикс из s в префиксном дереве ("ant"); затем, на втором шаге, к узлу, соответствующему самому длинному общему префиксу, добавляется новый путь для остатка в s ("hem")

Метод, как показано в листинге 6.5, состоит из двух основных шагов. На первом шаге выполняется переход в дереве по ссылке, соответствующей следующему символу в добавляемом ключе. Так будет продолжаться, пока не исчерпается вся входная строка или не обнаружится узел, не имеющий исходящего ребра, который соответствует следующему символу ключа.

В листинге 6.5 можно видеть, что первый шаг алгоритма реализован в строках с 1-й по 7-ю, в них продолжается обход дерева, и на основе символов во вставляемом ключе выбирается следующая ветвь. Все в точности как в методе `search`, с единственным отличием: если проверены все символы во входной строке (то есть пройден

весь путь от корня до целевого узла), то нужно просто установить в найденном узле признак ключевого узла. Это соответствует первому сценарию, описанному в начале раздела (не показан на рис. 6.8).

Листинг 6.5. Метод insert



Состояние, показанное на последней диаграмме в левой половине рис. 6.8, достигается, когда условие в строке 6 в листинге 6.5 стало ложным. В этом случае производится переход к строке 9 и в дерево добавляется новая ветвь с оставшимися символами: тем самым мы добавляем суффикс к строке, соответствующей пути от корня до текущего узла (в этом примере добавляется суффикс `"hem"` к уже имеющейся в дереве строке `"ant"`).

Последняя операция реализована с использованием (сюрприз!) рекурсии в другом вспомогательном методе, показанном в листинге 6.6.

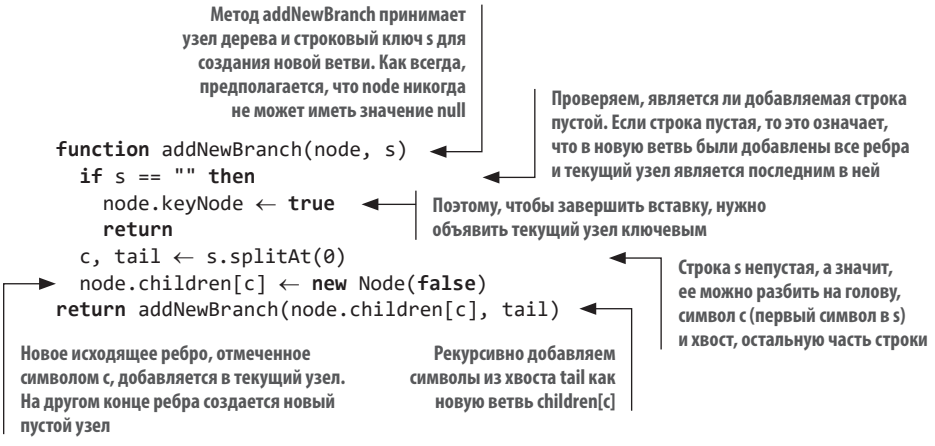
Метод имеет довольно простое определение. Как показано в правой половине рис. 6.8, он просто использует один символ из оставшейся строки, создает новое ребро, отмеченное этим символом, и новый пустой узел `N` на другом конце ребра, а затем рекурсивно добавляет конец строки в дерево `T` с корнем в `N`[1].

По аналогии с методом `search` можно доказать, что вставка тоже выполняется за амортизированное время $O(m)$, для случая, когда создание нового узла занимает

[1] Обратите внимание: поскольку узел N был только что создан, он будет единственным узлом в своем поддереве.

постоянное время (что так и есть, если для ребер использовать хеш-таблицы, и нет, если использовать простые массивы).

Листинг 6.6. Метод addNewBranch

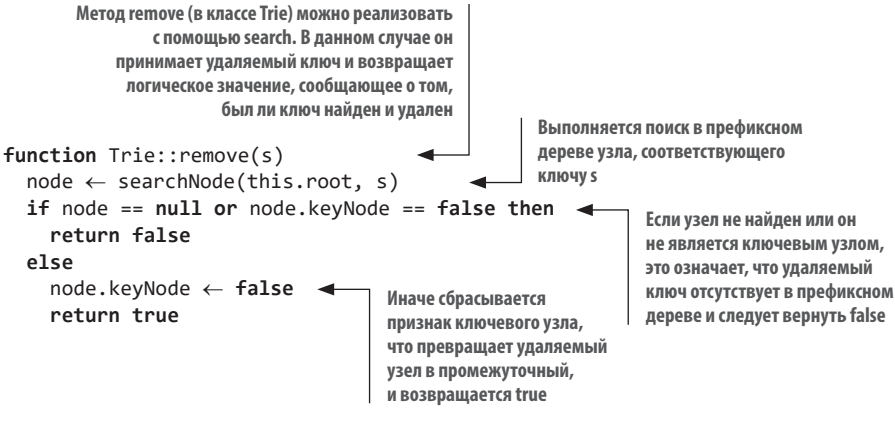


6.2.4. Удаление

В отношении удаления ключа из дерева у нас есть возможность выбора. Можем использовать более простой и дешевый алгоритм, позволяющий дереву только расти, или реализовать полноценный метод, более сложный и, возможно, более медленный на практике, но с наименьшим использованием памяти.

Разница между двумя вариантами заключается в том, что первый просто сбрасывает признак ключевого узла, превращая удаляемый узел в промежуточный, и никак не изменяет структуру дерева. Это легко реализовать повторным использованием метода `searchNode` из листинга 6.4, как показано в листинге 6.7.

Листинг 6.7. Метод Trie::remove (без усечения)



Какие недостатки имеет это решение? Взгляните на рис. 6.8. Если просто преобразовать ключевой узел N в промежуточный, может возникнуть одна из двух ситуаций:

- N является внутренним узлом и, соответственно, имеет дочерние узлы, хранящие ключи, доступ к которым можно получить, только пройдя через сам узел N;
- N является листом, и это означает, что в дереве нет других ключей, доступ к которым можно получить только через N.

Случай, когда N — лист, показан на рис. 6.9. После сброса признака ключевого узла в узле в конце пути в префиксном дереве появляется «висячая» ветвь, не содержащая ключа. Нет ничего страшного в том, чтобы оставить такую ветвь, потому что все методы префиксного дерева будут по-прежнему исправно работать. Но если данные часто меняются и операция удаления из префиксного дерева тоже выполняется достаточно часто, то объем памяти, расходуемой впустую на висячие ветви, может стать значительным.

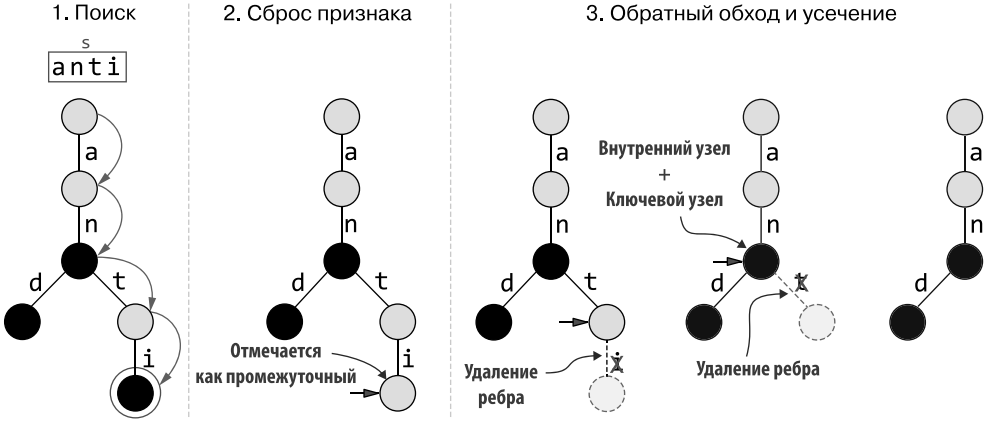


Рис. 6.9. Пример метода remove. *1*. Поиск ключа для удаления. *2*. Узел в конце пути отмечается как «промежуточный». *3*. Дерево усекается для удаления висячих ветвей

В таких случаях желательно иметь метод усечения, удаляющий висячие узлы в процессе обхода узлов в направлении, обратном направлению поиска. Если есть гарантия, что префиксное дерево «чистое», то есть не имеет висячих ветвей перед удалением текущего узла, то существует только два обстоятельства, при которых процедура усечения на этапе обратного обхода останавливается:

- достигнут ключевой узел. Очевидно, что нельзя удалить узел, содержащий ключ;
- достигнут внутренний промежуточный узел. Если после удаления последнего ребра в пути, по которому выполняется обратный обход, текущий узел оказывается листом, его можно удалить. В противном случае, если у узла есть другие потомки, то все его подветви будут содержать хотя бы один ключ (что вытекает из наших предположений), и, следовательно, текущий узел соответствует промежуточному символу в одной или нескольких строках, хранящихся в дереве, и его нельзя удалять.

В листинге 6.8 показана реализация метода, выполняющего удаление плюс усечение. Используйте рис. 6.9 для справки, работая с этим листингом.

Итак, можно заметить, что после достижения промежуточного узла (строка 4) узел в конце пути `"anti"` становится бесполезным листом. Метод возвращается к его родителю и удаляет ребро, отмеченное символом `"i"` (строки с 9-й по 12-ю в листинге 6.8). А затем узел в конце пути `"ant"`, который опять же является промежуточным узлом, тоже становится листом и, следовательно, может быть удален.

После возврата еще на один шаг метод обнаруживает, что родитель одновременно является ключевым узлом и имеет еще один дочерний узел, поэтому дальнейшее усечение дерева невозможно.

Листинг 6.8. Метод remove (с усечением)

Эта версия метода `remove`, очевидно, нуждается в реализации метода класса `Trie`, отличной от показанной в листинге 6.7. Но тогда метод API еще проще и превращается в элементарную обертку.

С точки зрения производительности к методу `remove` применимы те же соображения, что и к методам `search` и `insert`. Версия с усечением будет выполнять количество операций, равное $2m$ в худшем случае, — в два раза больше, чем простая версия без усечения, которая перебирает не более m ребер. Время выполнения тоже, вероятно, увеличится более чем в два раза, потому что, как можно судить по реализациям, разница в сложности кода имеет значение.

Однако при выборе более быстрой версии дерево может значительно вырасти, поэтому выбор между реализациями зависит от конкретных требований и контекста. Если набор данных динамически изменяется и ожидается большое количество вызовов `remove`, то, вероятно, лучше использовать версию с усечением. Если, напротив, соотношение вставок/удалений невелико или же есть вероятность, что удаленные строки придется вновь добавлять в дерево (часто или сразу после удаления), то лучше использовать более быструю (хотя и более беспорядочную) версию `remove`, показанную в листинге 6.7.

6.2.5. Самый длинный префикс

Методом `remove` завершается обзор классических операций с контейнерами. Но, как уже упоминалось, префиксные деревья могут предоставлять две операции нового типа, вызывающие к самой структуре данных наибольший интерес.

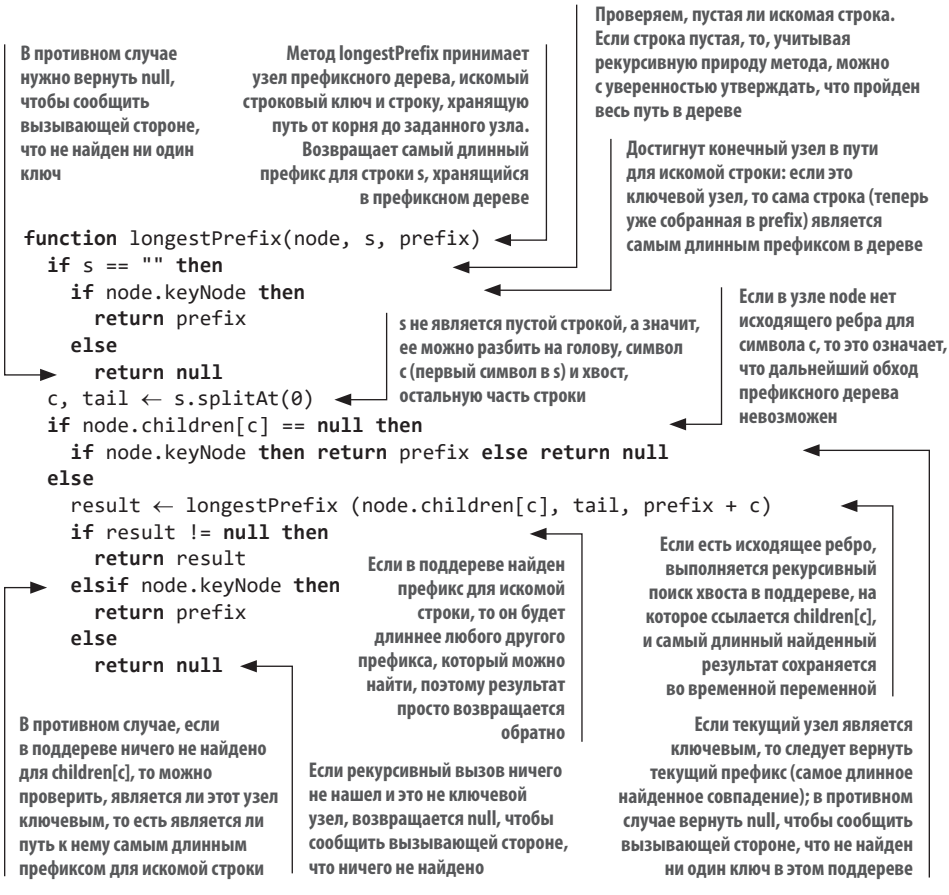
В этом разделе сосредоточимся на методе, возвращающем самый длинный префикс для искомой строки. Получив входную строку `s`, метод выполняет обход префиксного дерева, следуя за символами в `s` (насколько это возможно), и возвращает самый длинный найденный ключ.

Иногда, даже если искомый ключ отсутствует в префиксном дереве, бывает просто интересно получить соответствующий ему самый длинный префикс. Варианты использования этого метода будут представлены в подразделе, посвященном именно практическому применению.

Поиск самого длинного префикса почти полностью аналогичен работе метода `search`. Единственное отличие состоит в том, что для возврата из рекурсивной версии нужно проверить, найден ли искомый ключ, и если нет и текущий узел является ключевым, то вернуть ключ текущего узла. Это также означает, что метод должен возвращать строку, а не просто `true` или `false` и отслеживать пройденный путь, чтобы знать, что вернуть. Поскольку в процессе поиска запоминается пройденный путь, первый найденный ключевой узел будет содержать самый длинный префикс.

Листинг 6.9 поясняет эти идеи на примере реализации метода. Вспомните, как работает операция `insert`. Ее можно переосмыслить как двухэтапную операцию: поиск самого длинного общего префикса ключа, который уже присутствует в префиксном дереве, а затем добавление ветви с оставшимися символами. В качестве упражнения попробуйте переписать псевдокод для `insert`, используя метод `longestPrefix`.

Листинг 6.9. Метод longestPrefix



Как и в случае с другими методами, описанными к этому моменту, время выполнения этой операции линейно зависит от длины искомой строки: O(m), если $|s| == m$.

6.2.6. Ключи, соответствующие префиксу

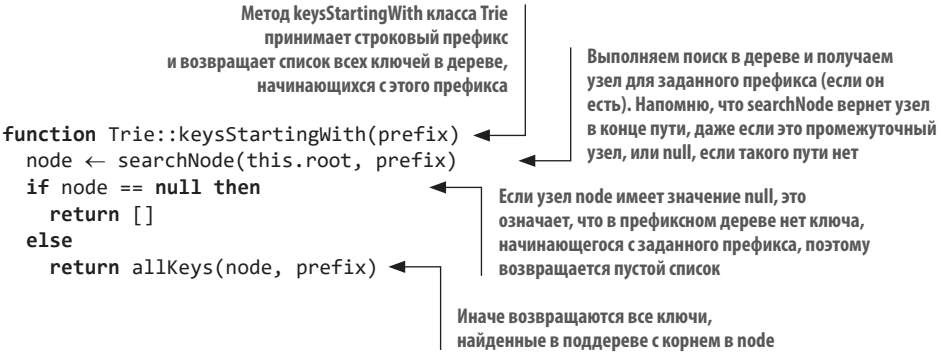
Последний метод, который мы рассмотрим, возвращает все ключи, соответствующие определенному префиксу.

Если остановиться и немного поразмышлять над определением префиксного дерева, то можно быстро додуматься до реализации этого метода. Даже само название этой структуры данных, *префиксное дерево*, предлагает решение. Мы уже знаем, что префиксное дерево стремится максимально компактно хранить строки с одним и тем же префиксом, преобразуя каждую строку в путь от корня до ключевого узла, а строки с одним и тем же префиксом совместно используют один и тот же путь в префиксном дереве.

Например, на рис. 6.3 все строки `"and"`, `"ant"`, `"anthem"` имеют общую часть пути, соответствующую общему префиксу `"an"`.

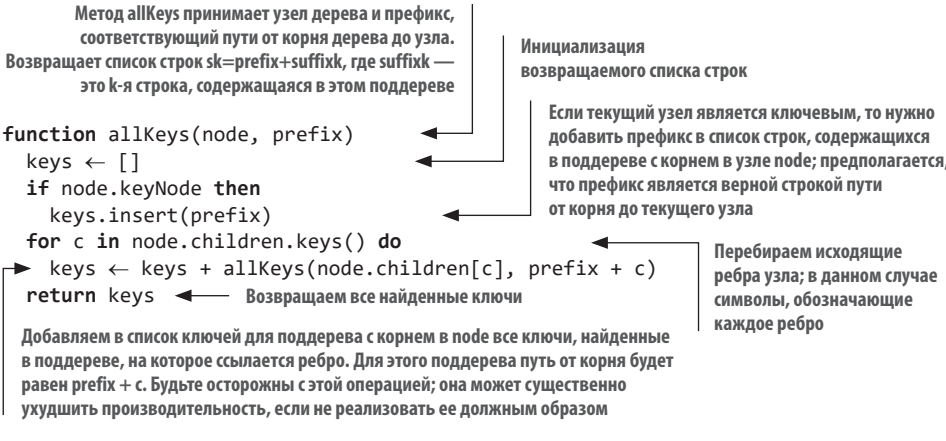
В листинге 6.10 показана реализация этого метода. Неудивительно, что он использует метод `searchNode`, описанный в листинге 6.4.

Листинг 6.10. Метод Trie:: keysStartingWith



Как видите, здесь присутствует вызов нового метода, который еще нужно описать: `allKeys`. Он выполняет обход (под)дерева и собирает все найденные ключи, как показано в листинге 6.11. Обход затрагивает все ветви каждого встреченного узла и прекращается только по достижении листа. Во втором аргументе метод принимает пройденный путь (строку) до узла, чтобы потом можно было определить, какой ключ вернуть.

Листинг 6.11. Метод allKeys



Выполняя асимптотический анализ этого метода, нужно быть особенно осторожными со строкой 6. В зависимости от используемого языка программирования и типа данных объединение списков может существенно ухудшить производительность, если реализовать его неправильно.

Наиболее эффективный способ накопления найденных ключей — передача в метод третьего параметра, аккумулятора, куда можно добавлять каждый ключ только один раз и в одном месте — в строке 4.

Исходя из этого предположения, время выполнения метода `allKeys` составляет O(j) для дерева с j узлами, и, следовательно, верхняя граница наихудшего случая для метода `keysStartingWith` составляет O($m + j$) для дерева с j узлами и префикса с m символами.

Но имейте в виду, что трудно, исходя из количества хранимых в дереве ключей, даже примерно оценить, сколько узлов оно будет иметь. Однако если известно, что дерево содержит n ключей, максимальная длина которых равна M, то можно утверждать: в худшем случае, который соответствует вырожденному дереву, где все слова имеют тот же префикс, что и искомый, и больше нет никаких символов, время выполнения для непустой строки составит O($m + n$ ($M - m$)).

В примере, изображенном на рис. 6.10, выполняется поиск всех ключей, соответствующих префиксу `"ant"`, поэтому $n = 6$, $m = 3$ и $M = 8$ (длина самого длинного ключа `"antidote"`).

Если попытаться отыскать все ключи, передав в префиксе пустую строку, метод вернет все ключи, имеющиеся в префиксном дереве, а время его выполнения будет равно O($n * M$), что также является верхней границей наихудшего случая.

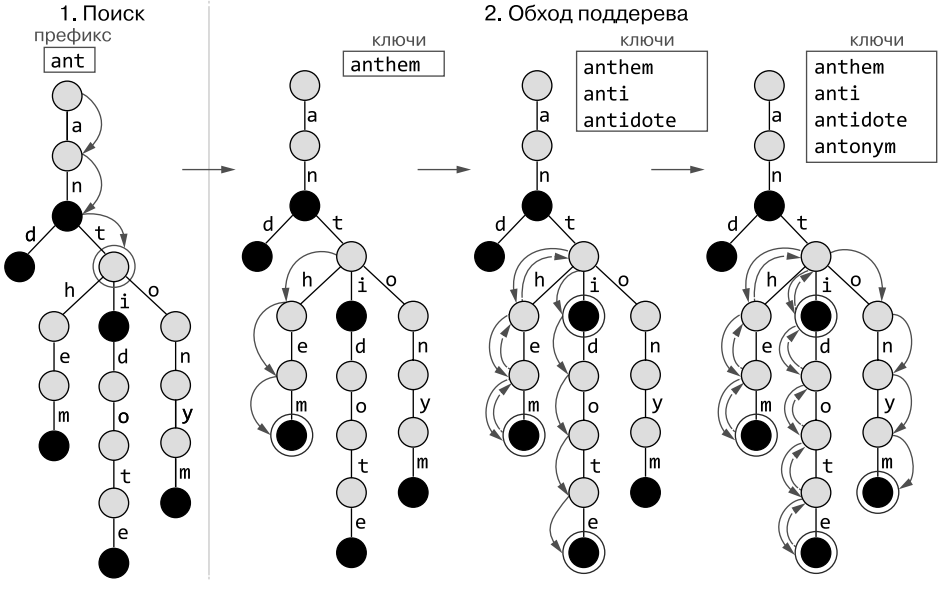


Рис. 6.10. Пример метода keysWithPrefix. *1*. Обход пути, соответствующего общему префиксу. *2*. Сбор всех ключей, найденных при обходе поддерева в конце пути для заданного префикса

6.2.7. Когда следует использовать префиксные деревья

Теперь, когда описаны все основные методы префиксных деревьев, было бы неплохо подвести итоги. В табл. 6.2 показана производительность методов префиксных деревьев и для сравнения — производительность эквивалентных методов сбалансированных двоичных деревьев поиска.

Таблица 6.2 помогает ответить на вопрос о производительности, который был обозначен в подразделе 6.2.1.

Таблица 6.2. Время выполнения операций с префиксными деревьями и сбалансированными двоичными деревьями поиска в предположении, что дерево хранит n ключей со средней длиной m; для простоты предполагается, что размер входного ключа равен m, где m ∈ O(M)

Метод	Двоичное дерево поиска	Двоичное дерево поиска + хеш	Префиксное дерево
`search`	$O(m \log(n))$	$O(m + \log(n))$	$O(m)$
`insert`	$O(m \log(n))$	$O(m + \log(n))$	$O(m)$
`remove`	$O(m \log(n))$	$O(m + \log(n))$	$O(m)$
`longestPrefix`	$O(m \times n)$	$O(m + n)$	$O(m)$
`keysWithPrefix`	$O(m \times n)$	$O(m + n)$	$O(m + n)$*

* В среднем.

Уже было показано, что префиксные деревья потребляют меньше памяти, чем двоичные деревья поиска, теперь мы также знаем, что они почти всегда работают быстрее.

Напомню, что обычно время работы BST выражается через n, количество сохраненных записей, однако в этом случае стоимость сравнения двух ключей явно равна не $O(1)$, а $O(m)$, где m — длина самого короткого из двух ключей.

В третьем столбце в табл. 6.2 показаны результаты для конкретного варианта двоичного дерева поиска, в каждом узле которого хранится хеш строки вместе с самим ключом. Такой подход требует дополнительной памяти $O(n)$ для хранения этих полей, зато позволяет быстро выполнить двухпроходное сравнение. Перед началом поиска для искомой строки `w` вычисляется `h(w)`. Затем в каждом узле сначала проверяется, соответствует ли `h(w)` хешу узла (что требует постоянного времени), и только если они равны, выполняется сравнение строк.

Прежде чем двигаться дальше, подытожим плюсы и минусы префиксных деревьев и сформулируем, когда же предпочтительнее использовать префиксные деревья вместо BST.

Положительные стороны, по сравнению с использованием двоичного дерева поиска или хеш-таблиц:

- время поиска зависит только от длины искомой строки;
- стоимость промахов при поиске ограничивается проверкой только нескольких символов (в частности, только самого длинного общего префикса в искомой строке и в корпусе, хранящемся в дереве);

- в префиксном дереве исключены конфликты ключей;
- нет необходимости предоставлять хеш-функцию или изменять хеш-функции по мере добавления новых ключей в дерево;
- префиксное дерево способно обеспечить алфавитный порядок следования записей по ключу.

Как бы привлекательно ни выглядел этот список, к сожалению, идеальной структуры данных не существует. И префиксные деревья тоже имеют некоторые недостатки:

- поиск в префиксных деревьях может работать медленнее, чем в хеш-таблицах, когда контейнер слишком велик, чтобы целиком уместиться в памяти. При использовании хеш-таблиц в таких случаях требуется меньше обращений к диску, вплоть до единственного, тогда как при использовании префиксного дерева потребуется $O(m)$ операций чтения с диска для строки длиной m;
- хеш-таблицы обычно размещаются в одном большом непрерывном блоке памяти, тогда как узлы дерева могут быть разбросаны по всему объему кучи. Соответственно, первые имеют лучшую локальность;
- идеальным вариантом использования префиксных деревьев является хранение текстовых строк. Теоретически можно преобразовать в строку любое значение, от числа до объекта, и сохранить его. Однако при хранении чисел с плавающей точкой возможны некоторые пограничные случаи, которые могут создавать длинные бессмысленные пути[1] из-за проблем с точностью представления таких чисел[2]. Например, это характерно для периодических или трансцендентных чисел или для результатов определенных операций с плавающей точкой, таких как 0,1 + 0,2;
- префиксные деревья имеют накладные расходы памяти для хранения узлов и ссылок. Как вы видели, некоторые реализации требуют, чтобы каждый узел хранил массив $|\Sigma|$ ребер, где Σ — используемый алфавит, даже если у узла мало или вообще нет дочерних узлов.

Таким образом, можно посоветовать использовать префиксные деревья, когда требуется часто выполнять поиск по префиксу (`longestPrefix` или `keysWithPrefix`). Если данные хранятся на медленных носителях, например на дисках, или когда важна локальность, используйте хеш-таблицы. В промежуточных случаях принять наилучшее решение поможет профилирование.

Префиксные деревья предлагают очень хорошую производительность для многих строковых операций. Однако из-за своей структуры они больше предназначены для хранения массивов дочерних элементов в каждом узле. Это может быстро привести к избыточному потреблению памяти. Общее количество ребер для дерева с n элементами в зависимости от степени перекрытия общих префиксов может колебаться между $|\Sigma| \times n$ и $|\Sigma| \times n \times m$, где m — средняя длина слова.

---

[1] http://stackoverflow.com/questions/588004/is-floating-point-math-broken/27030789#27030789.

[2] https://ru.wikipedia.org/wiki/IEEE_754-2008#Основные_и_взаимозаменяемые_форматы.

Вы уже видели, что для реализации узлов можно использовать ассоциативные массивы, в частности словари, сохраняя только непустые ребра. Но и это решение имеет свою цену. Она включает в себя стоимость не только доступа к каждому ребру (а может включать стоимость хеширования символа плюс стоимость разрешения конфликтов ключей), но и изменения размера словаря при добавлении новых ребер.

6.3. КОМПАКТНЫЕ ПРЕФИКСНЫЕ ДЕРЕВЬЯ

Для преодоления недостатков префиксных деревьев было разработано несколько альтернатив, в том числе *троичное дерево поиска* (Ternary Search Trie TST), уменьшающее потребление памяти за счет ухудшения производительности, и *компактное префиксное дерево* (radix trie).

Троичные деревья поиска снижают требования к памяти для хранения ссылок и освобождают от беспокойства об особенностях реализации для оптимизации хранения ребер. Однако количество узлов, которые нужно создать, по-прежнему находится на уровне количества символов, содержащихся в TST, то есть хранится весь корпус, $O(n \times m)$ для n слов со средней длиной m.

В префиксных деревьях большинство узлов не хранят ключи и являются лишь переходами на пути между ключами. Большинство этих переходов необходимы, но при хранении длинных слов складывается тенденция к созданию длинных цепочек внутренних узлов, каждый из которых имеет только одного потомка. Как было показано в подразделе 6.2.1, это основная причина, почему префиксным деревьям требуется слишком много памяти, иногда даже больше, чем двоичным деревьям поиска.

На рис. 6.11 показан пример префиксного дерева. Ничего особенного, обычное префиксное дерево. Как можно видеть, у промежуточных узлов всегда есть потомки (при условии, что висячие ветви отсекаются после удаления ключей): иногда только один потомок, иногда больше.

При наличии нескольких потомков у промежуточного узла имеется несколько ветвей, которые можно обойти. Однако если потомок один, то два узла начинают напоминать связанный список. Например, возьмем первые три узла, начиная от корня, на рис. 6.11: они кодируют префикс `"an"`, и поиск любой другой строки, начинающейся с `"a"`, но не заканчивающейся `"n"`, никуда не приведет.

Получается, что промежуточный узел является точкой ветвления, если он имеет более одного потомка. Это означает, что в дереве хранятся как минимум два ключа с общим префиксом, соответствующим этому узлу. Если так, то узел несет ценную информацию, которую нельзя сжать.

Ключевые узлы тоже хранят информацию, независимо от количества дочерних узлов. Они сообщают, что путь к ним составляет строку, которая хранится в дереве.

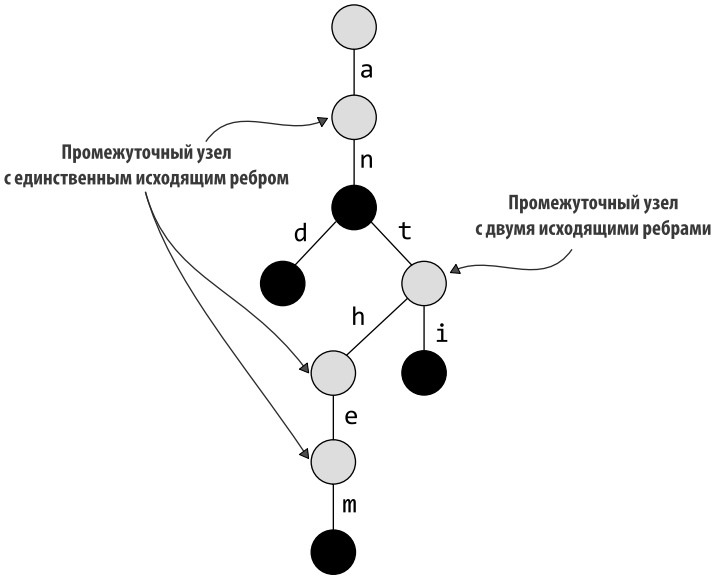


Рис. 6.11. Промежуточные узлы в префиксных деревьях имеют одно или несколько исходящих ребер (при условии, что реализовали усечение висячих ветвей после удаления). Видите разницу?

Компактные префиксные деревья (они же *radix trees*, они же *PATRICIA trees*[1]) основаны на идее существования возможности каким-либо образом сжать путь, ведущий к такого рода узлам, которые называются *сквозными узлами*.

Однако если промежуточный узел не хранит ключа и имеет только одного потомка, то он не несет релевантной информации, он лишь вынужденный шаг в пути.

Как? На рис. 6.12 показан процесс сжатия этих путей. Каждый раз, когда в пути присутствует сквозной узел, есть возможность сжать участок пути, связанный с этим узлом, в одно ребро, отмеченное строкой, составленной из меток исходных ребер.

Сколько может сэкономить это изменение? Чтобы получить общее представление, рассмотрим два дерева (рис. 6.12).

Исходное дерево имеет 9 узлов и 8 ребер, и с учетом допущений, сделанных в подразделе 6.2.1 в отношении 4-байтных служебных данных на узел, это означает $9 \times 4 + 8 \times 9 = 108$ байт. Сжатое дерево справа имеет 6 узлов и 5 ребер, но в этом случае каждое ребро несет не символ, а строку. Однако можно упростить операцию, учитывая ссылки на ребра и строковые метки отдельно. Соответственно, мы по-прежнему будем считать по 9 байт на ребро (поскольку теперь в стоимость ребра включается нулевой байт, отмечающий конец строки), но имеем право добавить сумму длин строк

[1] Оригинальное название этой структуры данных PATRICIA tree является аббревиатурой. *Morrison D. R.* PATRICIA — practical algorithm to retrieve information coded in alphanumeric // Journal of the ACM (JACM). — 1968. — Vol. 15.4 — P. 514–534.

как третий член в окончательное выражение; общий объем необходимой памяти составит $6 \times 4 + 5 \times 9 + 8 \times 1 = 77$ байт.

Другими словами, для этого простого префиксного дерева компактная версия требует на 30 % меньше памяти.

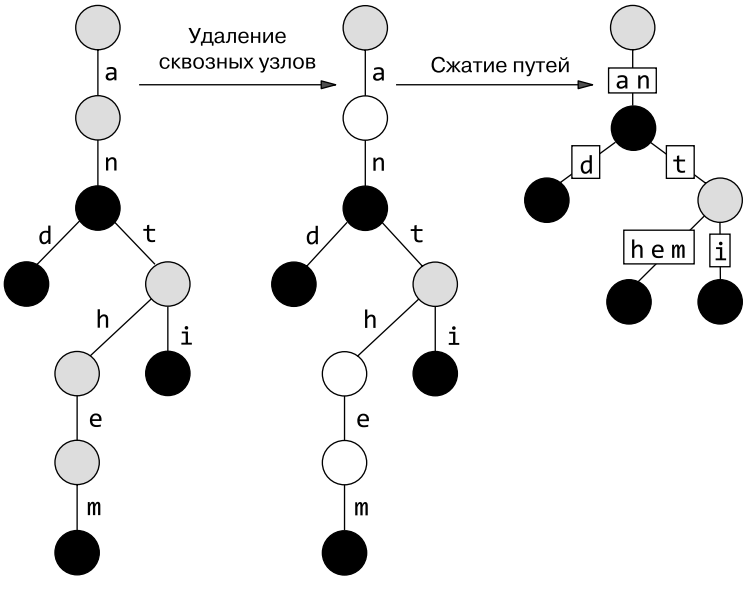


Рис. 6.12. Сжатие пути в префиксном дереве путем слияния ребер, смежных со сквозными узлами. Обратите внимание, что ребра в компактных префиксных деревьях отмечаются не только символами, но и строками

6.3.1. Узлы и ребра

Все операции с префиксными деревьями, описанные выше, можно похожим образом реализовать для компактных префиксных деревьев, только вместо ребер, отмеченных символами, нужно использовать ребра, отмеченные строками.

На высоком уровне логика методов останется почти такой же, как для обычных префиксных деревьев, однако, чтобы выяснить, какую ветвь выбрать для дальнейшего обхода, недостаточно просто проверить следующий символ в ключе, потому что ребра могут быть отмечены подстрокой, соответствующей нескольким символам в аргументе `s`.

Одно из важных свойств этих деревьев — никакие два ребра, исходящие из одного и того же узла, не имеют общего префикса. Это очень важно и позволяет эффективно хранить и проверять ребра.

Первый способ выполнения операций с такими деревьями состоит в том, чтобы хранить ребра в отсортированном порядке и использовать двоичный поиск для выявления ссылки, которая начинается со следующего символа `c` в ключе. Поскольку не может

быть двух ребер, начинающихся с c, то, найдя одно, можно сравнить остальные символы в его метке со следующими символами в искомой строке. Более того, двоичный поиск позволяет найти это ребро за логарифмическое время от количества ребер, а поскольку узел не может иметь больше k=|Σ| ребер (потому что может быть не более одного ребра, начинающегося с каждого символа в алфавите Σ), то худшее время выполнения двоичного поиска и выявления ребра-кандидата, как вы уже знаете, равно O(log(k)).

Так как k — это константа, не зависящая ни от количества ключей, хранящихся в дереве, ни от длины искомых/вставленных слов и т. д., с точки зрения асимптотического анализа можно считать, что O(log(k)) = O(1). Кроме того, в этом решении не требуется дополнительная память[1] для хранения ребер.

Этот способ показан на рис. 6.13, где также наглядно представлено, как работает двоичный поиск ребра-кандидата.

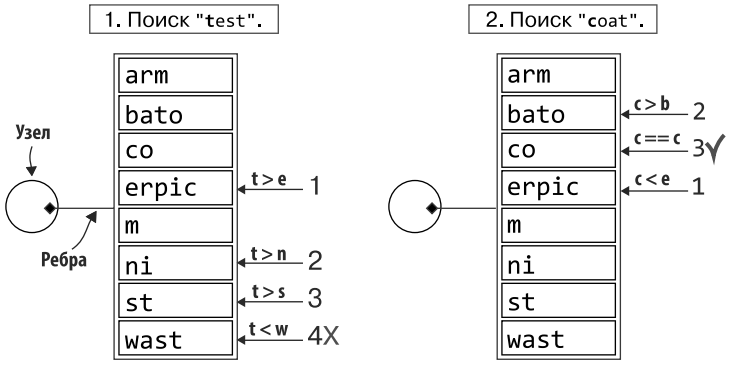


Рис. 6.13. Пример узла компактного префиксного дерева с ребрами, хранящимися в отсортированном массиве. *Слева:* двоичный поиск, завершившийся неудачей. *Справа:* успешный поиск. Сравнения выполняются только для первого символа каждой строки, однако при успешном поиске метка ребра является префиксом искомой строки

Обратите внимание, что совпадение между искомой строкой и меткой ребра не должно быть полным; вскоре вы увидите, что это значит для наших алгоритмов и как справляться с такими ситуациями.

Конечно, использование отсортированных массивов, как обсуждается в главе 4 и приложении B, означает логарифмический поиск, но линейную (читай медленную!) вставку. В асимптотическом анализе количество элементов можно считать константой, но с практической точки зрения такая реализация может значительно замедлить вставку новых ключей в большие префиксные деревья.

Альтернативные способы реализации словаря для ребер: сбалансированные деревья поиска, гарантирующие логарифмическое время поиска и вставки, или хеш-таблицы.

[1] За исключением постоянных накладных расходов на хранение объекта массива в большинстве языков.

Последний вариант изображен на рис. 6.14. Можно сохранить словарь, ключами которого являются символы, а значениями — полные строковые метки ребер узла вместе со ссылкой на дочерний узел, связанный этим ребром. Этот способ требует $O(k)$ дополнительной памяти для узла с k дочерними элементами, в худшем случае для каждого узла потребуется $O(|\Sigma|)$ дополнительной памяти.

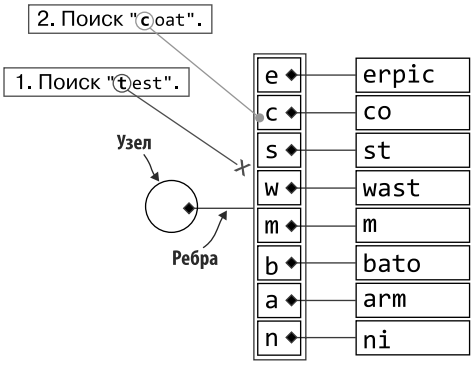


Рис. 6.14. Пример узла, в котором ребра, ведущие к дочерним узлам, хранятся в словаре. Ключами словаря являются символы — первые буквы меток, а значениями — полные метки и ссылки на конечный узел ребра. Здесь также показано упрощение поиска тех же строк, что и на рис. 6.13. Как и прежде, сравнения основаны на первом символе искомых строк

Несмотря на то что для хеш-таблиц требуется дополнительное место и некоторое время на выполнение операций вставки и удаления, этот способ позволяет выполнять поиск пути для ключа за постоянное время.

Независимо от реализации первым шагом будет сравнение первого символа входной строки с первым символом меток ребер.

В целом возможны четыре случая (показаны на рис. 6.15).

1. Строка s_E метки ребра точно соответствует подстроке `s` во входной строке, то есть `s` начинается с s_E и, соответственно, ее можно разбить на $s=s_E+s'$. В этом случае можно перейти по ребру к потомку и выполнить рекурсию по входной строке `s'`.
2. Существует ребро, метка которого начинается с первого символа в `s`, но s_E не является префиксом `s`; однако они должны иметь общий префикс s_P (длиной не менее одного символа). В таком случае дальнейшие действия зависят от выполняемой операции. Для поиска это означает неудачу, так как отсутствует путь к искомому ключу. Для вставки это означает необходимость распаковать ребро, чтобы вставить ребро s_E.
3. Входная строка `s` является префиксом метки ребра s_E. Это частный случай пункта 2 и предполагает аналогичное решение.
4. Наконец, при отсутствии совпадения с первым символом можно уверенно полагать, что дальнейшее движение вниз по дереву невозможно.

Теперь, прояснив общую организацию узлов компактного префиксного дерева, углубимся в алгоритмы. Учитывая все сказанное, их поведение естественным образом вытекает из методов префиксных деревьев.

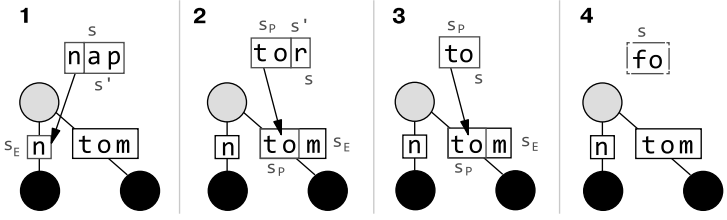


Рис. 6.15. Возможные результаты сравнения искомой строки со ссылками узла. *1*. Метка ребра полностью совпадает с частью искомой строки. *2*. Метка ребра и строка поиска имеют общий префикс, который короче обеих строк. *3*. Искомая строка является префиксом метки одного из ребер. *4*. Искомая строка не имеет общего префикса ни с одной из меток ребер

В листинге 6.12 показан псевдокод определений классов `RadixTrie` и `RTNode`, моделирующих эту новую структуру данных. Здесь добавлен класс, моделирующий ребра, чтобы сделать код более ясным. Интересно, заметили ли вы маленькую, но значимую деталь: отсутствие необходимости заранее определять фиксированный алфавит, имевшей место в обычных префиксных деревьях!

Полную реализацию вы найдете в репозитории книги на GitHub[1].

Листинг 6.12. Класс RadixTrie

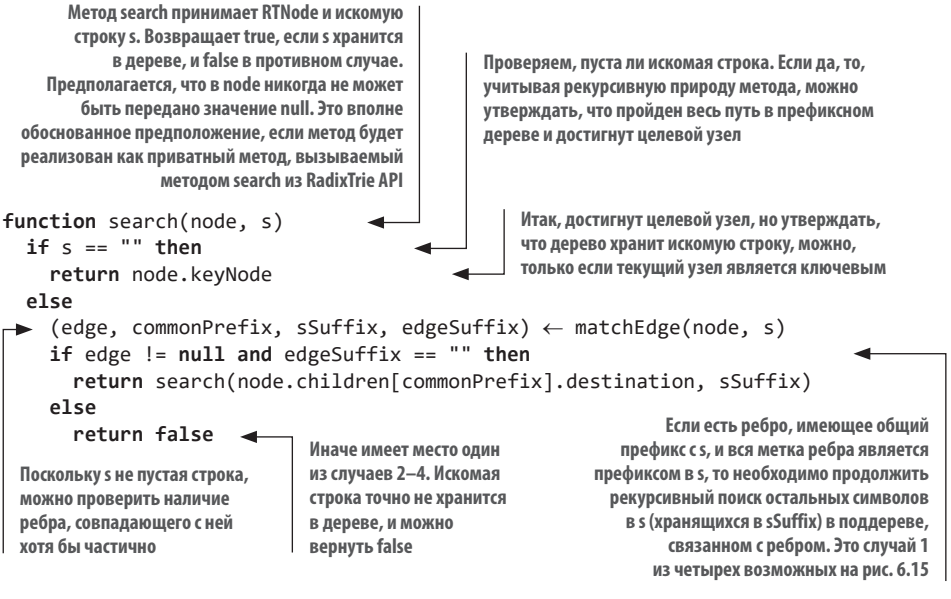
```
class RTEdge
  #type RTNode
  destination

  #type string
  label

class RTNode
  #type boolean
  keyNode

  #type HashMap<Char, RTEdge>
  children

  function RTNode(storesKey)
    children ← new HashMap()
    this.keyNode ← storesKey

class RadixTrie
  #type RTNode
  root

  function RadixTrie()
    root ← new RTNode(false)
```

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#radix-trie-aka-patricia-tree.

6.3.2. Поиск

Метод `search` (листинг 6.13) почти идентичен одноименному методу префиксного дерева; единственное отличие — порядок получения следующего ребра для продолжения обхода. Поскольку эта операция будет использоваться также в других методах, выделим ее в самостоятельный вспомогательный метод (листинг 6.14).

Листинг 6.13. Метод search компактного префиксного дерева



Этот метод просто ищет ребро, имеющее общий префикс с целевой строкой `s`. Напомню, что все ребра не могут иметь общий префикс, поэтому может быть не более одного ребра, начинающегося с того же символа, что и `s`.

Метод возвращает некоторую полезную информацию: самый длинный общий префикс между искомой строкой и меткой ребра, а также суффиксы этих двух строк (по отношению к их общему префиксу). Вызывающая сторона может использовать эти данные для выбора дальнейших действий.

В строке 6 листинга 6.13 эта информация используется для того, чтобы различать четыре случая, показанных на рис. 6.15. Единственный положительный случай для `search` — первый, поэтому нужно проверить, существует ли ребро, метка которого является префиксом для `s`.

Вспомогательный метод `matchEdge` реализуется просто, если предположить, что есть способ извлечения самого длинного общего префикса из двух строк. Это можно сделать, последовательно сравнивая символы с одинаковыми индексами в двух строках, пока не будет найдено несоответствие.

Листинг 6.14. Метод matchEdge компактного префиксного дерева

Метод matchEdge принимает RTNode и искомую строку. Возвращает кортеж с совпавшим ребром, если он есть, общий префикс для s и метки ребра, а также суффиксы этих строк. Предполагается, что в node никогда не может быть передано значение null, а в s — пустая строка

Поскольку строка по определению не пустая, в ней наверняка будет присутствовать первый символ с

Поиск в хеш-таблице ребра с меткой, начинающейся с символа с

Если такого ребра нет, значит, в дереве нет ребра, имеющего общий префикс с s. В таком случае следует вернуть null для ребра, пустую строку для общего префикса и соответствующие суффиксы

```
function matchEdge(node, s)
  c ← s[0]
  if node.children[c] == null then
    return (null, "", s, null)
  else
    edge ← node.children[c]
    prefix, suffixS, suffixEdge ← longestCommonPrefix(s, edge.label)
    return (edge, prefix, suffixS, suffixEdge)
```

Получаем исходящее ребро с меткой, начинающейся с символа с

Возвращаем вычисленные значения

Вычисляем наибольший общий префикс между s и меткой ребра, а также оставшиеся суффиксы

Предполагается, что такой метод (`longestCommonPrefix`) существует и он также возвращает суффиксы двух строк, то есть две строки, состоящие из символов в каждой из входных строк, оставшихся после удаления общего префикса.

На рис. 6.16 показан пример метода поиска в компактном префиксном дереве, полученном в результате сжатия префиксного дерева на рис. 6.3.

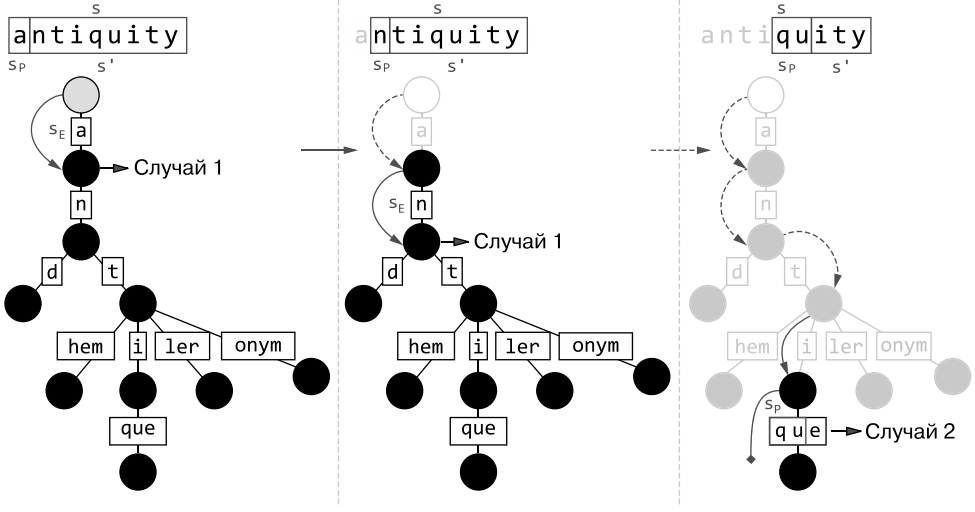


Рис. 6.16. Неудачный поиск строки "antiquity" в компактном префиксном дереве, соответствующем дереву на рис. 6.3. Первые две диаграммы показывают начальные шаги поиска, а последняя — заключительный шаг

6.3.3. Вставка

Как уже упоминалось, случаи 2 и 3 являются наиболее сложными, особенно для метода `insert`. После обнаружения частичного совпадения между ключом и ребром нужно разбить метку ребра, разделить ребро на два новых ребра и добавить посередине новый узел, соответствующий самому длинному общему префиксу s_P.

Этот порядок действий показан на рис. 6.17. После обнаружения общего префикса добавляем новый узел для разделения ребра, частично совпадающего со вставляемой строкой, а затем добавляем новую ветвь в этот новый узел.

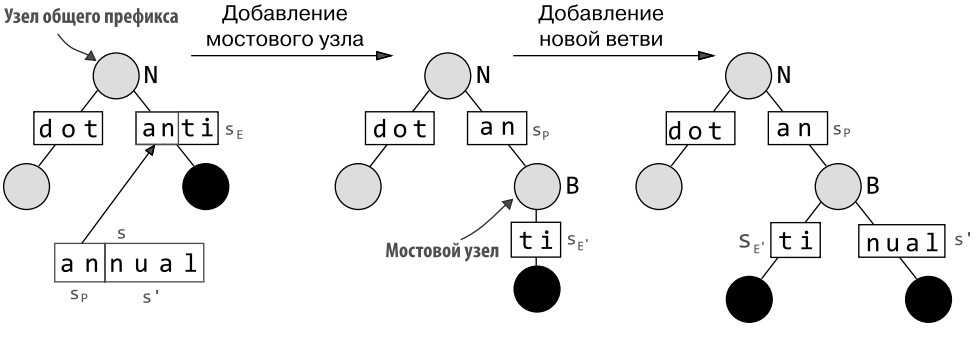


Рис. 6.17. Обработка случая 2 при сопоставлении ребер в ходе выполнения вставки. В этом примере добавляется слово "annual" к узлу, содержащему ребро с меткой "anti". В процессе вставки добавляется мостовой узел B, связанный с N ребром с общим префиксом "an", а затем два новых ребра, выходящих из B

Добавляемый узел называется *мостовым узлом*, потому что он будет служить мостом между существующим узлом, соответствующим общему префиксу двух строк, и путями, ведущими к конечным узлам для этих строк. Мостовые узлы — это, как легко понять, точки бифуркации, где разветвляется путь от корня.

Для простоты можно представить, что мы распаковываем ребро, ведущее к дочернему элементу, и возвращаемся к представлению обычного префиксного дерева с одним символом на ссылку. Затем проходим этот путь, пока не дойдем до конца общего префикса (скажем, до узла B), добавляем в B новую дочернюю ветвь и, наконец, снова сжимаем два подпути с двух сторон от B.

В листинге 6.15 показан псевдокод метода `insert`, а на рис. 6.18 и 6.19 — два примера работы этого метода с упрощенным деревом.

Метод `insert` следует той же логике, что и его версия для префиксного дерева в листинге 6.5. Он движется вниз по дереву, насколько это возможно (следуя самому длинному пути, охватывающему префикс вставляемой строки), а затем добавляет новую ветвь для нового ключа.

Листинг 6.15. Метод insert компактного префиксного дерева

Метод insert принимает RTNode и строковый ключ для вставки. Он ничего не возвращает, но изменяет префиксное дерево. И снова предполагается, что в аргументе node никогда не будет передано значение null, что является вполне обоснованным предположением, если этот метод реализован как приватный, вызываемый методом insert из общедоступного API префиксного дерева

Проверяем, пуста ли искомая строка. Если да, то, учитывая рекурсивную природу метода, можно утверждать, что пройден весь путь в префиксном дереве

Поскольку s не пустая, далее можно определить, существует ли ребро, совпадающее с ней хотя бы частично

По достижении целевого узла для вставляемого ключа в нем устанавливается признак ключевого узла, чтобы подтвердить, что s присутствует в дереве

```
function insert(node, s)
  if s == "" then
    node.keyNode ← true
  else
    (edge, commonPrefix, sSuffix, edgeSuffix) ← matchEdge(node, s)
    if edge == null then
      this.children[s[0]] ← new RTEdge(s, new Node(true))
    elif edgeSuffix == "" then
      insert(edge.destination, sSuffix)
    else
      bridge ← new Node(false)
      this.children[s[0]] ← new RTEdge(commonPrefix, bridge)
      bridge.children[edgeSuffix[0]] ←
          new RTEdge(edgeSuffix, edge.destination)
      insert(bridge, sSuffix)
```

Соответствует случаю 1. Существует ребро, метка которого является префиксом для s; нужно просто пойти дальше по ребру

Соответствует случаю 4 (рис. 6.15). Если нет ни одного ребра с общим префиксом, совпадающим хотя бы с первым символом в s, то нужно добавить новое ребро с меткой s, ведущее к новому ключевому узлу

Изменяем исходящее ребро в этом узле так, чтобы оно указывало на мостовой узел и содержало метку с общим префиксом

Наконец, нужно рекурсивно добавить оставшуюся часть ключа. Если sSuffix — пустая строка, это соответствует случаю 3, иначе — случаю 2

Иначе остаются только случаи 2 и 3. Метка ребра и s имеют общий префикс, но в метке есть символы, отсутствующие в s. Поэтому нужно разбить ребро и создать мостовой узел

Добавляем в мостовой узел ребро, ведущее к бывшему потомку текущего узла: меткой будет прежняя метка исходного ребра без commonPrefix

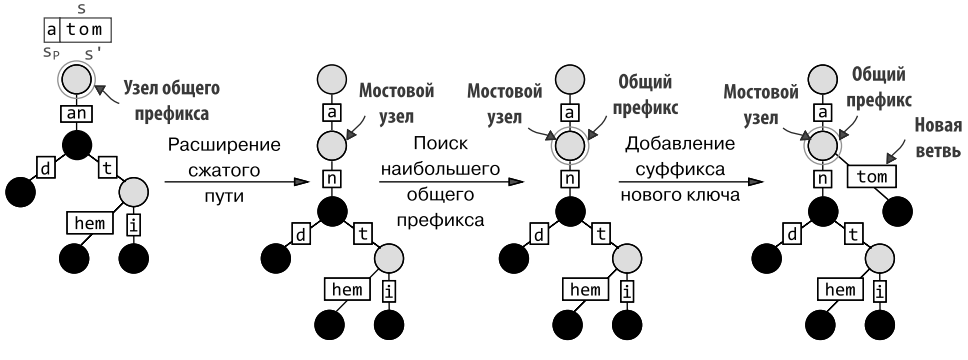


Рис. 6.18. Пример работы метода insert

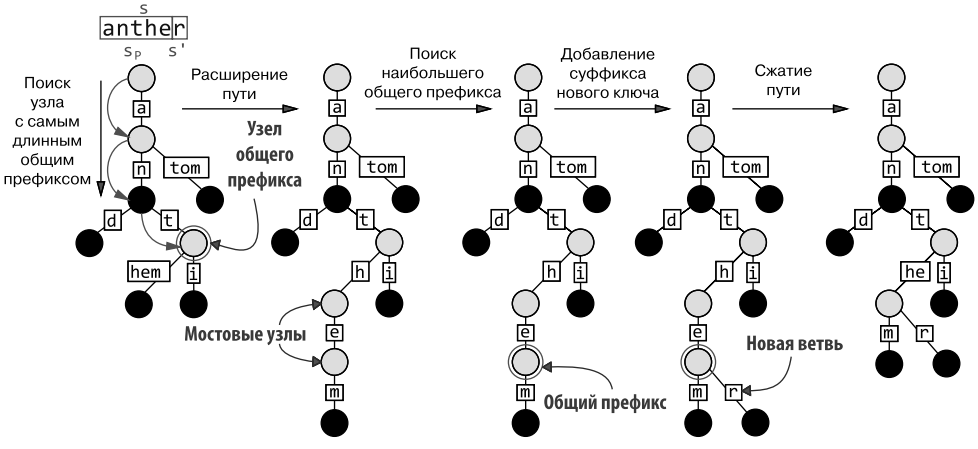


Рис. 6.19. Еще один пример работы метода insert с объяснением распаковки пути

Обход дерева становится более сложным, потому что в каждом узле нужно различать четыре возможных случая сопоставления меток ребер, и эта сложность отражается на длине метода. Кроме того, в случаях 2 и 3 нужно разбить ребро и добавить мостовой узел. Однако добавление новой ветви становится проще, потому что нужно лишь добавить новое ребро и один узел.

6.3.4. Удаление

По аналогии с поиском, единственное отличие методов `remove` в компактном и обычном префиксных деревьях связано с извлечением общего префикса. И это неудивительно, потому что удаление ключа можно рассматривать как успешный поиск с последующей очисткой удаляемого узла.

При удалении не нужно беспокоиться о разделении ребер или добавлении мостовых узлов. Если удаляемый ключ хранится в дереве, для него должен существовать путь, идеально соответствующий ему. Однако после удаления может понадобиться сжать последнюю часть удаленного пути, потому что превращение существующего ключевого узла в промежуточный узел может изменить структуру дерева, ведь при этом вводится сквозной узел (как рассказывалось в подразделе 6.3.1).

На рис. 6.20 показан пример работы метода `remove` компактного префиксного дерева.

Кроме того, может также понадобиться выполнить обычное усечение при удалении ключа-листа. Отличие от простого префиксного дерева состоит в том, что в компактном префиксном дереве понадобится удалить только одно ребро.

Пример на рис. 6.20 показывает оба случая, когда может понадобиться исправить родителя узла. Сначала в листе сбрасывается признак ключевого узла, а затем, после удаления узла из дерева, его родитель становится сквозным узлом, следовательно, его тоже можно удалить и сжать путь от его родителя до единственного дочернего узла.

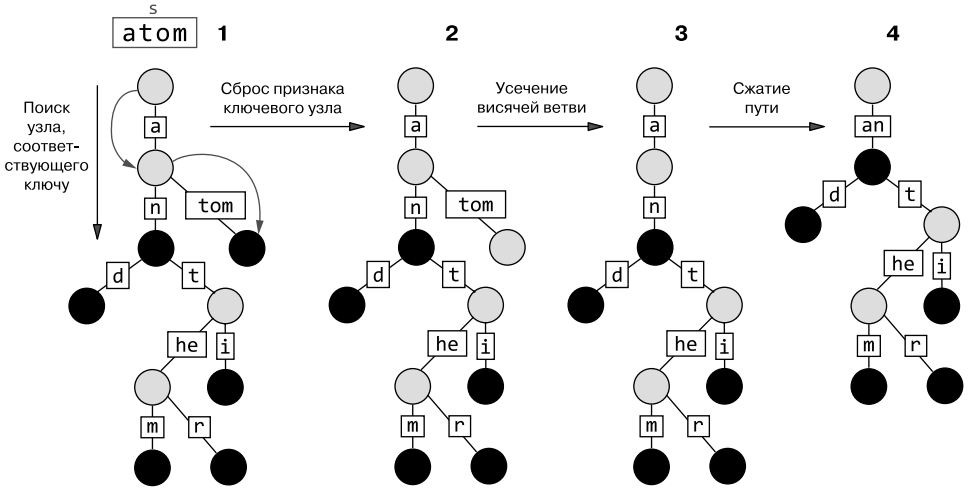


Рис. 6.20. Удаление слова "atom" из примера компактного префиксного дерева. *1*. Поиск узла, соответствующего удаляемому ключу. Путь должен полностью совпадать с ключом. *2*. Сбрасывание признака ключевого узла, в результате он становится промежуточным. Если узел является листом, то образуется висячая ветвь. *3*. Усечение висячей ветви. Если родитель удаленного узла имел только два дочерних узла, то теперь он стал сквозным узлом. *4*. Сжатие пути удалением сквозного узла и объединением ребер

В листинге 6.16 показан псевдокод этого метода, в псевдокоде используются две вспомогательные функции.

Листинг 6.16. Метод remove компактного префиксного дерева

Сохраняем во временной переменной узел на конце ребра

Метод remove принимает узел и строку, которую нужно удалить из поддерева с корнем в указанном узле. Возвращает пару логических значений; первое — признак успешного удаления ключа, второе — признак, что последняя ссылка превратилась в пустую висячую ветвь и ее следует усечь

Проверяем, пуста ли искомая строка. Если да, то можно утверждать, что пройден весь путь до требуемого узла

Сбрасываем признак ключевого узла, превращая узел в промежуточный

Поиск завершен, остается только вернуть результат вызывающей стороне, сообщив, что удаление выполнено успешно, и обозначив необходимость усечь этот узел (если это лист)

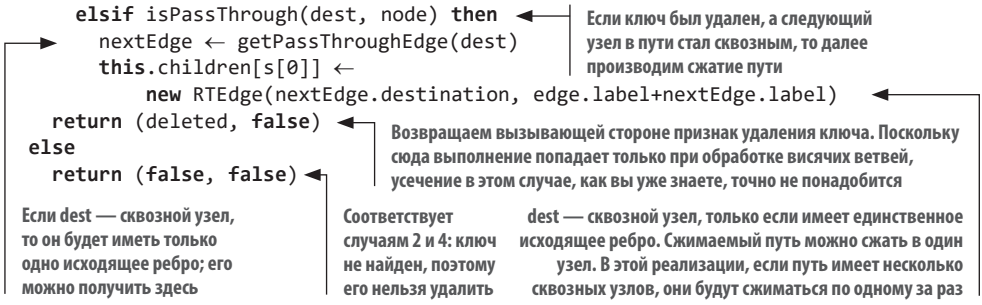
Поскольку s не пустая, далее можно определить, существует ли ребро, совпадающее с ней хотя бы частично

Соответствует случаю 1. Существует ребро, метка которого является префиксом для s; нужно просто пойти дальше по ребру

Рекурсивно вызываем remove для остатка строки и сохраняем результат

Если ключ был удален, а следующий узел в пути стал листом, который теперь можно обрезать, то далее удаляем ребро до этого листа

```
function remove(node, s)
  if s == "" then
    node.keyNode ← false
    return (true, node.children == 0)
  else
    (edge, commonPrefix, sSuffix, edgeSuffix) ← matchEdge(node, s)
    if edge != null and edgeSuffix == "" then
      dest ← edge.destination
      (deleted, shouldPrune) ← remove(dest, sSuffix)
      if deleted then
        if shouldPrune then
          node.children[s[0]] ← null
```



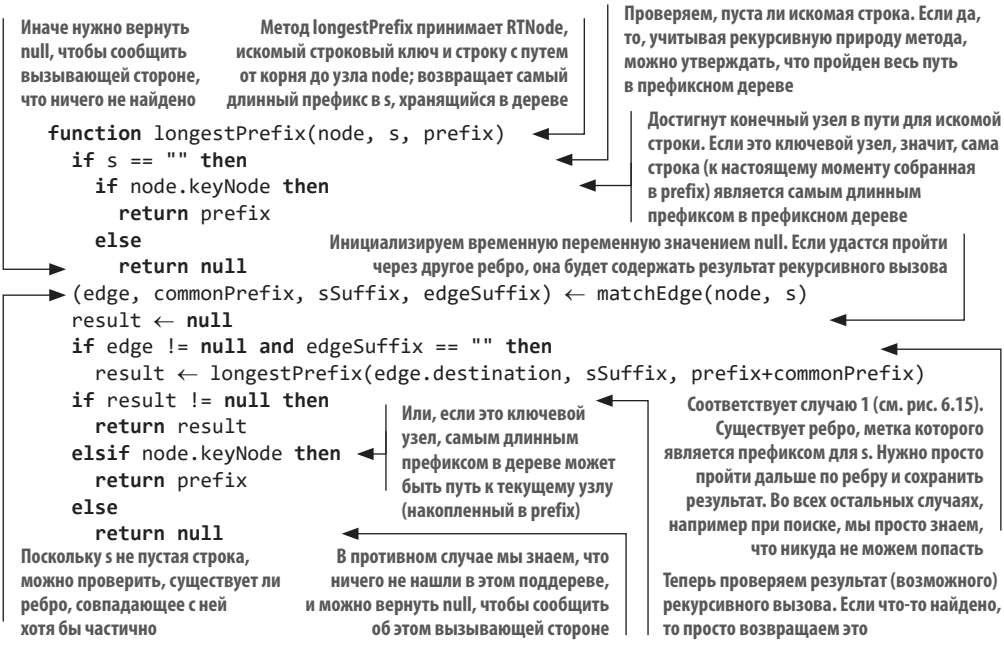
`isPassThrough` проверяет, является ли узел сквозным. Такое возможно, только когда узел не является ключевым, имеет единственное исходящее ребро и даже его родитель имеет единственное исходящее ребро (следовательно, этой функции также нужно передать родителя). Реализацию я оставляю вам в качестве упражнения.

Сквозной узел имеет только одно исходящее ребро, поэтому его поле `children` будет содержать ровно один элемент; `getPassThroughEdge` — обертка для получения этой записи.

6.3.5. Самый длинный общий префикс

Перенести этот метод из реализации обычного префиксного дерева несложно. Достаточно немного изменить алгоритм поиска, чтобы учесть другой способ сопоставления ребер. В листинге 6.17 представлен псевдокод для компактного префиксного дерева.

Листинг 6.17. Метод longestPrefix компактного префиксного дерева



6.3.6. Поиск ключей, начинающихся с префикса

Чтобы найти начальную точку для обхода и извлечь все ключи в поддереве с корнем в префиксе, в префиксном дереве можно использовать метод `search`.

К сожалению, в случае с компактным префиксным деревом ситуация немного сложнее, потому что префиксы, отсутствующие в дереве, могут частично совпадать с ребрами. Возьмем, к примеру, дерево на рис. 6.20, где такие префиксы, как `"a"` или `"anth"`, отсутствуют в дереве. Последний даже не имеет узла в конце соответствующего пути, однако в компактном префиксном дереве есть несколько слов, начинающихся с этих префиксов.

Если бы мы просто искали узлы, лежащие в конце путей, соответствующих этим строкам, мы бы отбросили все эти, в общем-то, верные результаты. Поэтому нам нужно написать специальную версию метода поиска для этой операции, которая будет различать случай 2 совпадения ребер без возможности дальнейшего обхода и случай 3, когда искомый фрагмент строки является правильным префиксом последнего ребра в пути и поэтому поддерево, на которое ссылается ребро, действительно будет содержать строки, соответствующие искомому префиксу. Разница между двумя случаями показана на рис. 6.21.

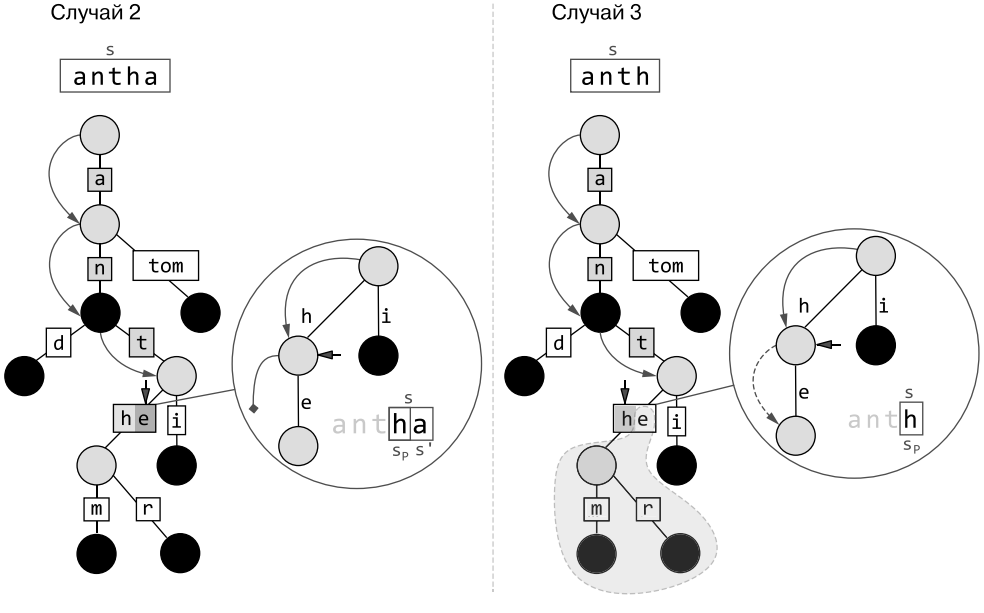


Рис. 6.21. Разница между случаями 2 и 3 при поиске узла, соответствующего более короткой строке, включающей префикс. *Слева:* в случае 2 следующее ребро совпадает не со всеми символами в оставшемся фрагменте строки, поэтому в соответствующем дереве searchNode(s) вернет null. *Справа:* в случае 3 есть полное совпадение фрагмента строки, которое заканчивается в середине ребра. В соответствующем префиксном дереве поиск вернет промежуточный узел, в частности сквозной; поэтому, поскольку в сквозных узлах не хранятся ключи, можно точно так же начать перечисление ключей с первого несквозного узла среди его потомков

Этот новый метод показан в листинге 6.18. Он получил имя `searchNodeWithPrefix`, чтобы его можно было отличить от метода `search`, отыскивающего точные соответствия. Метод API `keysStartingWith` и вспомогательный метод `allKeysInBranch`, напротив, в основном идентичны эквивалентным методам префиксных деревьев, поэтому я оставляю их реализацию вам в качестве практического упражнения.

Листинг 6.18. Метод searchNodeWithPrefix компактного префиксного дерева

```
function searchNodeWithPrefix(node, s)
  if s == "" then
    return node
  (edge, commonPrefix, sSuffix, edgeSuffix) ← matchEdge(node, s)
  if edge == null then
    return null
  elsif edgeSuffix == "" then
    return searchNodeWithPrefix(edge.destination, sSuffix)
  elsif sSuffix == null then
    return edge.destination
  else
    return null
```

Метод searchNodeWithPrefix принимает RTNode и искомый строковый ключ s; возвращает корень поддерева, содержащего все имеющиеся ключи с префиксом s

Проверяем, пуста ли искомая строка. Если да, то, учитывая рекурсивную природу метода, можно утверждать, что пройден весь путь в префиксном дереве: этот узел точно соответствует s

Соответствует случаю 1 (рис. 6.15). Существует ребро, метка которого является префиксом для s. Нужно просто продолжить рекурсивный обход и поиск оставшихся символов в строке

Поскольку s не пустая строка, можно проверить, существует ли ребро, совпадающее с ней хотя бы частично

Соответствует случаю 4 (см. рис. 6.15). Искомый префикс отсутствует в дереве

Соответствует случаю 4. Далее нет путей, начинающихся с s в префиксном (под)дереве с корнем в node

Соответствует случаю 3. В конце пути для искомого префикса нет узла, хранящего ключ, однако в несжатом префиксном дереве здесь был бы сквозной узел. То есть все префиксы, хранящиеся в поддереве с корнем в узле, будут начинаться с s+edgeSuffix, и эти и только эти хранимые строки будут иметь префикс s

На этом завершается краткий обзор основных методов компактного префиксного дерева и обсуждение структур данных для эффективного поиска строк. Если вас заинтересовала эта тема, то я предлагаю вам самим дополнительно взглянуть на суффиксные деревья и суффиксные массивы — две интересные структуры данных, широко используемые в таких областях, как биоинформатика. Их обсуждение, к сожалению, выходит за рамки этой главы.

6.4. ВАРИАНТЫ ПРИМЕНЕНИЯ

Теперь, познакомившись с двумя конкретными структурами данных, реализующими абстрактную структуру `StringContainer`, можно перейти к изучению вариантов их практического применения.

Как это обычно бывает со структурами данных, дело не в том, что появилось что-то новое, что нельзя реализовать без применения префиксных деревьев, а в том, чтобы добиться выполнения некоторых операций лучше или быстрее с их применением, чем с помощью других структур.

Это особенно верно в случае префиксных деревьев, так как они специально созданы для ускорения обработки запросов на основе строк. Поскольку одним из основных вариантов применения префиксного дерева является реализация текстовых словарей, пробным камнем часто становятся хеш-таблицы.

6.4.1. Проверка орфографии

Вернемся к нашему основному примеру! В главе 4 показано, что в ранних версиях программ проверки орфографии использовались фильтры Блума, но через некоторое время их заменили более эффективные альтернативные варианты — префиксные деревья.

Очевидно, что первый шаг к созданию механизма проверки орфографии — вставка всех ключей из нашего словаря (здесь имеется в виду «орфографический словарь», а не структура данных!) в префиксное дерево.

Имея готовое префиксное дерево, реализовать обратную связь, просто выделяющую опечатки, очень просто. Нужно лишь выполнить поиск и, если он завершился неудачей, сделать вывод, что в слове есть опечатка.

Но представьте, что, кроме этого, принято решение реализовать вывод вариантов исправления опечатки. Как можно это сделать с помощью префиксного дерева?

Предположим, что проверяется слово `w`, состоящее из m символов, и можно принять предложения, отличающиеся от `w` не более чем на k символов: другими словами, нам нужны слова, для которых *расстояние Левенштейна* (также известное как расстояние редактирования) не превышает k.

Чтобы найти эти слова в дереве, начнем обход дерева от корня и в процессе обхода сохраним массив из m элементов, где i-й элемент этого массива — наименьшее расстояние редактирования, необходимое, чтобы признать совпадение ключа в текущем узле с первыми i символами в искомой строке.

Для каждого узла `N` проверяем массив, содержащий расстояния редактирования:

- если все расстояния в массиве больше максимально допустимого, то можно остановиться, ибо нет смысла продолжать обход поддерева (потому что расстояния могут только увеличиваться);
- в противном случае запоминаем последнее расстояние редактирования (для всей строки поиска), и если оно лучшее из найденных нами к этому моменту, то связываем его с текущим ключом узла и сохраняем.

Закончив обход, сохраним ключ, ближайший к искомой строке, и расстояние до него.

На рис. 6.22 показан упрощенный пример работы алгоритма. Он использует префиксное дерево, но этот же алгоритм с небольшими изменениями можно легко реализовать с применением компактного префиксного дерева. На самом деле в этом алгоритме нас больше всего интересуют ключевые узлы, а не промежуточные.

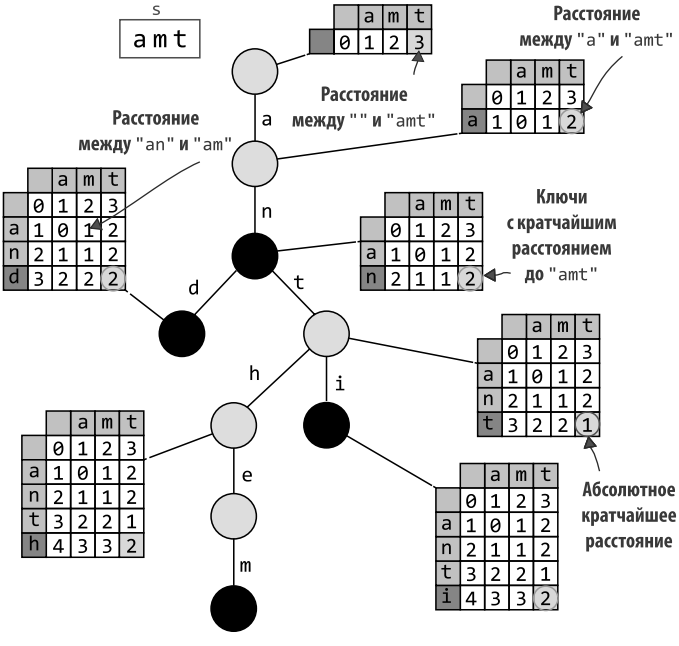


Рис. 6.22. Поиск вариантов исправления опечаток по буквам для слова s="amt" с использованием префиксного дерева. Обратите внимание, что для каждого узла вычисляется только последняя строка каждой таблицы на основе предыдущей (соответствующей родителю узла). Путь с ближайшим расстоянием до "amt" соответствует слову "ant", но в этом префиксном дереве соответствующий узел не является ключевым, поэтому считается, что слово "ant" отсутствует в дереве и не может быть возвращено в результате! Зато на расстоянии 2 находятся три ключа: "an", "and", "anti". Все они могут быть возвращены

Алгоритм начинает обход дерева с корня, соответствующего пустой строке (потому что путь, ведущий к ней, тоже пустой). На каждом шаге сравниваем целевое слово `s` (`"amt"` в этом примере) со словом, соответствующим текущему узлу; в частности, вычисляем расстояние между каждым префиксом `s` и словом, связанным с текущим узлом.

Итак, для корня расстояние между пустой строкой и пустым префиксом `"amt"` (который, очевидно, тоже является пустой строкой) равно 0 (потому что они совпадают). Расстояние между `""` и `"a"` равно 1, потому что нужно добавить один символ к первой строке, чтобы получить вторую, и т. д.

После вычисления вектора расстояний выполняется переход через любое исходящее ребро, и процесс повторяется для следующих узлов. В данном случае имеется только одно ребро, связанное со строкой `"a"`. Можно заполнить следующую строку в таблице, используя только предыдущую строку, и сравнить последний символ

(тот, который отмечает последнее пройденное ребро) с каждым символом в `s` (обратите внимание, что для столбца с пустой строкой расстояние всегда будет равно длине пути).

Следовательно, вторая строка в нашей таблице будет начинаться с `1` и иметь `0` в ячейке `[1,1]`, потому что обе строки начинаются с `"a"`. Для следующего символа в `s` нет соответствующего символа в ключе узла (потому что он короче), поэтому нужно добавить 1, чтобы получить его расстояние, и то же самое для последнего символа. Фактически, если рассматривать префикс `"am"`, расстояние до `"a"` равно 1, а для `"amt"` — 2.

Обратите внимание, что стоимость сравнения двух строк всегда содержится в нижней правой ячейке, поэтому для `"a"` и `"amt"` это расстояние равно 2.

Алгоритм продолжает обход ветвей, пока не будет достигнута точка, в которой стоимость уже не может уменьшиться (то есть когда длина пути уже равна строке `s` или длиннее ее, нет смысла продолжать спуск по ветви после обнаружения ключевого узла), или все расстояния в последней строке больше заданного пользователем порога максимально допустимого расстояния. Это особенно полезно при поиске длинных строк, которые в противном случае привели бы к обходу большей части дерева (в то время как наиболее вероятные верные варианты исправления находятся на расстоянии 2–3 символов).

Как показано на рис. 6.22, наименьшее расстояние имеет путь `"ant"`. Однако здесь есть подвох! В действительности это префиксное дерево не содержит ключа `"ant"`, соответственно, мы не можем принять это значение во внимание.

На расстоянии 2 есть несколько ключей, и любой из них или все они могут использоваться как подсказки.

Как быстро можно найти возможные варианты замены? На данный момент, после обсуждения в разделе 6.1, вы знаете, что поиск строки в префиксном дереве выполняется быстрее, чем в альтернативных структурах данных, даже в худшем случае: $O(m)$ сравнений для строки длиной m, в то время как для хеш-таблиц или двоичных деревьев поиска это время составит $O(m + \log(n))$ в лучшем случае.

6.4.2. Сходство строк

Сходство двух строк является мерой расстояния, которое их разделяет. Обычно это некоторая функция количества изменений, необходимых для преобразования одной строки в другую.

Вот два примера таких оценок расстояния:

- *расстояние Левенштейна*, количество односимвольных правок;
- *расстояние Хэмминга*, количество позиций, в которых строки различаются.

Как было показано в предыдущем подразделе, средства проверки орфографии используют сходство строк для выбора наилучших вариантов исправления опечаток.

Но в последнее время стала набирать популярность еще одна область применения оценки расстояний: сопоставление последовательностей ДНК в биоинформатике. Эта задача требует больших вычислительных ресурсов, поэтому использование неподходящих структур данных может сделать ее решение невозможным.

Когда нужно сравнить только две строки, наиболее эффективным способом является прямое вычисление расстояния Левенштейна. Однако, если нужно сравнить одну строку с n другими строками, чтобы найти наилучшее совпадение, вычисление n раз расстояния Левенштейна становится нецелесообразным. Время работы составит $O(n \times m \times M)$, где m — длина искомой строки, а $O(M)$ — средняя длина n строк в корпусе.

Как оказывается, большего успеха можно добиться с помощью префиксных деревьев. Использование того же алгоритма, что и для проверки орфографии (без усечения по пороговому значению), для вычисления всех расстояний потребует времени, равного $O(m \times N)$, где N — общее количество узлов в дереве. И хотя построение дерева может занять до $O(n \times M)$, это произойдет только один раз в начале, и если скорость поиска достаточно высока, стоимость первоначального построения будет амортизирована.

Теоретически N может быть $O(n \times M)$, как было показано в разделе 6.2, если никакие две строки в корпусе не имеют одинакового префикса. На практике, однако, весьма вероятно, что N на порядок меньше, чем $n \times M$, и ближе к $O(M)$. Более того, как вы видели в предыдущем подразделе, если установить порог для максимально допустимого расстояния, то есть для наибольшей разницы между двумя строками, и запоминать лучший найденный результат, то можно еще больше сократить количество проверяемых узлов во время поиска.

6.4.3. Сортировка строк

Burstsort[1] — это алгоритм пакетной сортировки с эффективным кэшированием, работающий аналогично алгоритму поразрядной сортировки MSD (Most Significant Digit — «наиболее значащая цифра»). Однако *пакетная сортировка* эффективнее с точки зрения кэширования и даже быстрее, чем поразрядная сортировка!

Оба вида сортировки имеют одинаковое асимптотическое время выполнения $O(n \times M)$, совпадающее с теоретической нижней границей для сортировки n строк длиной M, но пакетная сортировка дает результаты в два раза быстрее благодаря локальности ссылок и лучшему распределению памяти.

---

[1] *Sinha R., Zobel J.* Efficient trie-based sorting of large sets of strings // Proceedings of the 26th Australasian computer science conference. — Vol. 16. — Australian Computer Society, Inc., 2003.

Подробное рассмотрение этого алгоритма выходит за рамки данной главы, но, чтобы вы могли получить представление о том, как работает пакетная сортировка, отмечу, что алгоритм Burstsort динамически строит префиксное дерево во время сортировки и использует его для разделения строк, назначая каждой строке корзину (аналогично поразрядной сортировке). Как уже упоминалось, оба алгоритма имеют одинаковую асимптотическую стоимость, потому что начальные символы каждой строки проверяются только один раз. Однако порядок доступа к памяти в алгоритме пакетной сортировки обеспечивает более эффективное использование кэша.

Алгоритм MSD обращается к каждому символу каждой строки один раз, а пакетная сортировка обращается к каждой строке целиком только один раз, при этом обращения к узлам префиксного дерева выполняются случайным образом.

Однако набор узлов префиксного дерева намного меньше набора строк, поэтому кэш используется более эффективно.

Если размер набора строк превышает размер кэша, то пакетная сортировка оказывается значительно быстрее, чем любые другие алгоритмы сортировки строк.

6.4.4. T9

Функция T9 стала настолько важной вехой в истории развития мобильных устройств, что мы до сих пор (ошибочно) называем новые мобильные средства проверки орфографии T9, хотя эта функция осталась в прошлом с появлением смартфонов.

Название T9 — это аббревиатура от Text on 9 keys («текст на 9 клавишах»), потому что алфавит (задолго до появления мобильных телефонов) был разделен на группы по 3–4 символа, умещавшиеся на цифровой клавиатуре телефона.

В оригинальной версии для стационарных телефонов подходящую цифру нужно было нажимать от одного до четырех раз, чтобы выбрать каждую букву. Например, кнопку 2 нужно было нажать один раз, чтобы набрать `'a'`, дважды, чтобы набрать `'b'`, и трижды, чтобы набрать `'c'`.

Идея T9 заключалась в том, чтобы позволить пользователю для каждой буквы в слове нажимать каждую клавишу только один раз при вводе *i*-й буквы, принадлежащей группе на *k*-й кнопке. Затем в T9 появилась возможность ввода комбинаций букв и даже целых слов, если в словаре имелось только одно возможное совпадение.

Например, при вводе 2-6-3 выбирались все три комбинации букв для формирования декартова произведения `[a,b,c]x[m,n,o]x[d,e,f]`, а T9 предоставляла правильные слова, такие как `[and, cod, con,...]`.

Это стало возможным благодаря использованию префиксного дерева и уточнению результатов поиска для каждой нажатой клавиши.

1. При нажатии кнопки 2 начинается параллельный обход поддеревьев в префиксном дереве, связанных с ребрами, отмеченными как `'a'`, `'b'`, `'c'` (все три

наверняка будут присутствовать в дереве для любого языка, использующего латинский алфавит).

2. После нажатия второй кнопки на клавиатуре для каждого из трех путей, с которых начался обход, T9 отмечает наличие дочерних элементов с метками `'m'`, `'n'` и `'o'`, и проверяет достигнутые узлы на этом втором уровне. Каждая комбинация представляет путь от корня до узла уровня 2. Скорее всего, не все 9 комбинаций будут иметь пути в дереве: например, маловероятно, что какое-либо слово будет начинаться с `"bn"`.
3. Процесс продолжается с нажатием любой следующей клавиши, пока не останется ни одного узла, достижимого через возможные пройденные пути.

Для этой конкретной задачи узлы в дереве, вероятно, должны хранить больше информации, чем простой логический признак для каждого ключа. Скорее, они будут хранить частоту встречаемости вводимого слова в корпусе (например, вероятность его употребления в английском языке). Благодаря этому при наличии нескольких вариантов можно вернуть наиболее вероятный: например, можно было бы ожидать, что T9 правильно выберет слово `and`, а не `cod`.

6.4.5. Автодополнение

За последние десять лет все познакомились с функцией автодополнения в окнах поиска. Более того, мы часто даже ожидаем, что окна поиска предоставят эту функцию по умолчанию.

Типичный процесс автодополнения выглядит следующим образом: пользователь на стороне клиента (обычно в браузере) вводит несколько первых букв, а функция автодополнения в окне поиска отображает несколько вариантов, начинающихся с уже введенных символов (ничего не напоминает? Например, «все ключи с префиксом»?). Если бы набор возможных значений, которые можно вставить в поле поиска, был статичным и небольшим, то его можно было бы передать клиенту вместе со страницей, кэшировать и использовать непосредственно на клиенте.

Однако часто наборы возможных значений оказываются очень велики и могут меняться со временем или даже запрашиваться динамически.

Поэтому в реальных приложениях клиент обычно отправляет (асинхронно) REST-запрос на сервер с уже введенными символами.

Сервер хранит префиксное дерево (или, что более вероятно, дерево PATRICIA — компактное префиксное дерево) с допустимыми словами, отыскивает строки, начинающиеся с комбинации символов, введенной к данному моменту, и действительный префикс, и затем возвращает определенное количество ключей из поддерева этой строки.

Получив ответ, программа-клиент, просто показывает список результатов с сервера в качестве предложений.

АВТОДОПОЛНЕНИЕ В СЕТИ

Вообще говоря, совершенно не обязательно, чтобы запрос выполнялся через конечную точку REST или отправлялся асинхронно. Однако первое служит признаком чистой архитектуры, а без второго функция автодополнения не имела бы большого смысла.

Чтобы избежать потери пропускной способности сети и вычислительной мощности сервера, запросы на автодополнение обычно отправляются раз в несколько секунд или если пользователь прекращает ввод. Когда приходит ответ, страница обновляет список отображаемых подсказок.

Но это решение все еще неидеально, потому что с HTTP/1.1 невозможно отменить запросы, которые уже были отправлены и достигли сервера. Кроме того, нет никакой гарантии получения ответов в том же порядке, в каком были отправлены запросы. Следовательно, если до того, как вернется предыдущий ответ, будет введено или стерто несколько символов, то необходимо сохранить какую-то версию ответов и никогда не заменять отображаемые результаты при получении устаревшего ответа.

Эта проблема будет смягчена в HTTP/2, где, кроме всего прочего и интересного, предусмотрена отмена запросов.

РЕЗЮМЕ

- Элементарные типы данных, такие как целые числа или числа с плавающей точкой, фундаментально отличаются от строк: для хранения любого целого числа требуется один и тот же объем памяти[1], но строки могут иметь произвольную длину и, следовательно, требовать для хранения произвольное количество байтов.
- Длина строки, с которой работает структура данных, является важным фактором при асимптотическом анализе; она оставляет место для дальнейшей оптимизации, которая невозможна с элементарными типами данных.
- Префиксные деревья позволяют эффективно хранить большие наборы строк и выполнять поиск в них, если предположить, что многие строки в наборе имеют общие префиксы.

[1] За исключением, например, целочисленного типа bignum в Python, который может представлять произвольно большие числа, используя переменное количество байтов для их хранения.

- Префиксы строк являются ключевым фактором для этой новой структуры данных, которая, в свою очередь, позволяет эффективно искать строки с общим префиксом или, наоборот, самый длинный префикс в наборе данных для заданной строки.
- Компактное префиксное дерево стремится по возможности сжимать пути, чтобы обеспечить более компактное представление префиксных деревьев без излишней сложности или потери производительности.
- Многие приложения в самых разных сферах, от проверки орфографии до биоинформатики, обрабатывающие строки, могут выиграть от использования префиксных деревьев.

7 Примеры использования: кэш LRU

В этой главе

- ✓ Как избежать повторных вычислений.
- ✓ Кэширование как решение.
- ✓ Описание различных типов кэшей.
- ✓ Разработка эффективного решения для кэша LRU.
- ✓ Обсуждение MFU и других вариантов управления приоритетом.
- ✓ Обсуждение стратегий кэширования.
- ✓ Рассуждения о параллелизме и синхронизации.
- ✓ Как кэширование может применяться в конвейере анализатора эмоциональной окраски.

Эта глава отличается от всего, что вы встречали в книге до сих пор. Мы не будем вводить здесь новые структуры данных, но покажем, как можно использовать уже описанные в главах 2–5 для создания еще более сложной структуры данных. Используя списки, очереди, хеш-таблицы и т. д. в качестве строительных блоков, можно создать расширенную структуру данных для быстрого доступа к значениям и результатам вычислений, которые извлекались недавно.

По сути своей кэш — это больше, чем просто структура данных; это сложный агрегат, состоящий из нескольких компонент. В простейшей форме кэш можно реализовать как ассоциативный массив, но по мере прочтения этой главы вы будете постепенно осознавать, что он может стать таким же сложным, как веб-сервис. Мы с вами будем вникать в детали работы все более и более сложных кэшей, и, изучив этот материал, вы сможете сформулировать обоснованное мнение по этой теме.

7.1. НЕ ВЫЧИСЛЯЙТЕ ОДНО И ТО ЖЕ ДВАЖДЫ

В своей повседневной работе разработчики программного обеспечения пишут приложения, большинство из которых выполняет очень простые задачи. Сложение двух чисел, или даже их деление, или сложение двух векторов (на современных графических процессорах GPU[1]) — тривиальные операции, довольно быстрые, не требуют оптимизации. (Конечно, так было не всегда, но если вы достаточно молоды, то вам никогда не приходилось иметь дело с процессорами x86.)

Тем не менее, какими бы быстрыми и оптимизированными ни были многоядерные процессоры или вычислительные кластеры, всегда были и будут вычисления и сложные операции, выполняющиеся так долго, что становится непростительным допускать расточительность и выполнять их по много раз, когда в действительности в этом нет необходимости.

Для примера вернемся к операции вычисления суммы векторов. Если векторы имеют миллиарды элементов (ну или слишком много, чтобы целиком уместиться в памяти GPU), то даже эта операция может оказаться довольно затратной. То же верно для повторного деления одних и тех же чисел миллиарды раз, когда можно обойтись несколькими сотнями: влияние такого сокращения на время работы приложений будет ощутимым.

Обработка чисел не единственная область, где важна оптимизация вычислений. Например, в веб-приложениях одной из самых затратных операций, безусловно, является доступ к базе данных, и особенно если это приложение выполняет итерации по курсору для подсчета или вычисления чего-либо.

Это не просто дорого (с точки зрения расходования ресурсов), но и медленно. Курсоры баз данных[2] могут включать обширные блокировки[3] (нередко целой таблицы или даже всей базы данных), если они не предназначены только для чтения, но даже

---

[1] Графические процессоры изначально разрабатывались для ускорения обработки и буферизации изображений в устройствах отображения. Но на рубеже веков они стали все чаще использоваться в качестве универсальных (с высокой степенью параллелизма) вычислительных устройств, и фактически для решения алгебраических задач с большим объемом вычислений, таких как машинное обучение, часто предпочтение отдается именно им, а не центральному процессору.

[2] Курсоры — это управляющие структуры, которые позволяют перемещаться по набору записей в таблице базы данных; для простоты их можно рассматривать как указатели на следующую строку в части таблицы. Они часто используются для последовательной обработки строк (либо чтения данных, либо их изменения), в отличие от пакетного чтения/записи, когда сразу читается/записывается группа записей.

[3] Блокировки — это еще одна конструкция в базах данных, которая предотвращает выполнение любых операций чтения/записи в необработанных данных/таблицах/БД во время изменения. Они обеспечивают согласованность и могут существенно повлиять на доступность. Как правило, не рекомендуется использовать курсоры, позволяющие обновлять данные, потому что они могут удерживать блокировки в течение длительного времени.

внесение в память (запись) отдельных записей может в некоторых базах данных потребовать блокировки страницы или всей таблицы.

Каждый раз, когда блокируются какие-то данные, все операции чтения этих данных вынуждены ждать снятия блокировки. Если подойти к этой задаче без должного внимания, то большое количество операций записи в БД может попросту «сжечь» ее[1], замедлить работу вашего приложения и вызвать задержки, раздражающие пользователей, или даже остановить БД, что, в свою очередь, вызовет прерывание HTTP-вызовов по тайм-ауту и, как результат, прекращение работы вашего веб-сайта/приложения.

Жуткая картина, правда? Ах, если бы только был способ избежать этого!

Многим компаниям, даже технологическим гигантам, в первые дни существования (и на заре эпохи Интернета) приходилось на собственном опыте сталкиваться с проблемами обеспечения бесперебойной работы по мере развития веб-сайтов. Пока они не нашли способов эти проблемы разрешать.

Одна из лучших возможностей уменьшить нагрузку на базу данных — исключить повторяющиеся за короткое время дорогостоящие вычисления результатов. Существует много других стратегий, ортогональных этой: от сегментирования[2] до ослабления ограничения согласованности (переход к согласованности в конечном итоге[3]). Они необходимы или по крайней мере полезны для достижения хорошей и стабильной производительности веб-приложения. Очевидно, что эта книга не самое подходящее средство их представления, но литературы по масштабируемости очень много, и если вам доведется работать в этой области, я рекомендую познакомиться с некоторыми из замечательных книг или веб-руководств, которые были опубликованы[4].

Вместо этого здесь мы сосредоточимся на кэшировании. Чтобы немного сузить рамки и сделать приводимый пример более понятным, рассмотрим правдоподобную ситуацию, когда действительно можно использовать кэширование.

Представьте, что у вас есть служба-агрегатор, собирающая новости о компаниях из социальных сетей и выставляющая им оценку, например, по положительным или

---

1 Не поймите буквально! Это просто выражение из профессионального жаргона.

2 Суть сегментирования (sharding) заключается в разбиении данных, пользователей или транзакций (или всех сразу) на группы, каждая из которых закрепляется за отдельными машиной/кластером/центром обработки данных. Этот прием помогает уравновесить нагрузки и позволяет использовать более дешевые серверы и базы данных меньшего размера для каждой группы, что в конечном счете упрощает и удешевляет масштабирование приложений.

3 Ослабление требования к согласованности по времени. Идея объясняется далее в этой главе.

4 Например, https://www.manning.com/books/principles-of-cloud-design или https://www.manning.com/books/progressive-web-apps, где также есть хорошие главы о кэшировании.

отрицательным отзывам людей (в науке о данных это называется *анализом эмоциональной окраски* (sentiment analysis)[1]). На рис. 7.1 показана возможная упрощенная архитектура такой службы.

Вашей службе придется посылать запросы внешним API для сбора сообщений из социальных сетей, а затем анализировать их и выяснять эмоциональную окраску каждого сообщения. После их анализа и оценки с некоторой степенью уверенности служба должна будет решить, является ли общее отношение к компании положительным или отрицательным. Например, вы можете определить со степенью достоверности 70 %, что в своем сообщении пользователь положительно отзывается о компании X, в то время как в отношении отрицательной окраски другого сообщения у вас может быть уверенность на 95 %. В таком случае следует взвесить оба сообщения, принимая во внимание степень уверенности (кроме всего прочего).

Скорее всего, вам придется оформлять подписки, платить за подключение к множеству служб разных социальных сетей и писать адаптеры, принимающие данные в разных форматах.

Для опроса каждой внешней службы потребуется отправлять HTTP-запросы за пределы вашей интрасети, и каждый вызов будет иметь некоторую задержку, разную для разных служб. Вы можете отправлять вызовы параллельно, но даже в этом случае, если для принятия решения нужны все данные, общая задержка будет не меньше, чем у самой медленной из этих служб.

Задержка может стать настоящей проблемой; скажем, вы обязались обеспечить такой *уровень обслуживания*, согласно которому ответы на 95 % вызовов в течение месяца будут возвращаться за 750 мс, но, к сожалению, в пиковые часы, когда по сети передается большой объем трафика, пара внешних служб может задерживаться с ответом на время до 3 с.

Более того, у вас еще и тайм-аут для HTTP-ответов установлен равным 3,5 с, то есть необходимо вернуть ответ клиенту в течение 3,5 с, иначе балансировщик нагрузки уничтожит запрос. Возможно, для решения проблемы достаточно перенастроить тайм-аут? Но представьте, что это этого делать нельзя, потому что, изменив тайм-аут, вы не справитесь с входящим трафиком, учитывая доступные ресурсы. Предположим, что обработка данных из одного источника занимает около 250 мс, для получения этих данных требуется 3 с. Учитывая время на обработку входящего запроса, выполнение некоторой заключительной обработки и отправку ответа, есть риск получить множество ошибок 503[2]. И что? А то, что вы не сможете выполнить свои обязательства по *соглашению об уровне обслуживания* (Service Level Agreement, SLA).

---

[1] Анализ эмоциональной окраски основан на таких методах, как обработка естественного языка, анализ текста и компьютерная лингвистика, и позволяет автоматически определять эмоциональное отношение одного или нескольких субъектов к какой-либо теме.

[2] HTTP-код состояния Service unavailable («Услуга недоступна»).

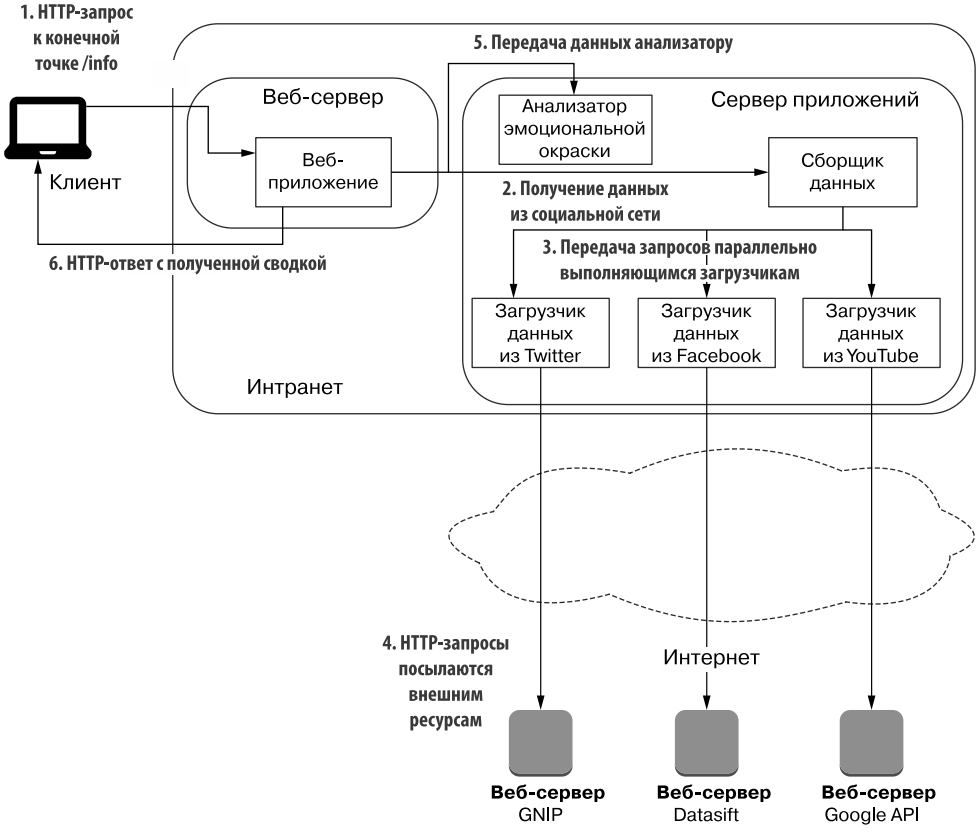


Рис. 7.1. Возможная архитектура приложения, получающего ежедневную сводку из социальных сетей. *1*. Клиент отправляет веб-серверу HTTP-запрос на получение сводки за день (для конкретной компании). *2*. Веб-сервер связывается с сервером приложений, чтобы получить данные из основных социальных сетей (здесь список приводится только для примера). Сервер приложений может физически и логически находиться на том же компьютере, что и веб-сервер (в этом случае запрос превращается в вызов метода), или физически размещаться на том же компьютере, но выполняться в другом процессе, или даже размещаться на другом компьютере, вследствие чего запрос будет являться фактическим HTTP-запросом. *3*. Сборщик данных запускает новый поток для каждого загрузчика/социальной сети. Все они выполняются асинхронно. *4*. Каждый загрузчик отправляет HTTP-запрос внешнему API через Интернет. После получения ответа он возвращает загруженные данные сборщику. По завершении всех загрузчиков сборщик возвращает все собранные данные вызывающей стороне (в данном случае веб-серверу). *5*. Веб-сервер синхронно вызывает анализатор эмоциональной окраски, передавая тому исходные данные, и ожидает получения сводки. Как вариант, веб-сервер может вызвать оркестратор или анализатор эмоциональной окраски непосредственно в точке 2, и этот шаг был бы делегирован. *6*. После этого следуют вычисления оценки эмоциональной окраски и передача ее обратно веб-серверу, тот формирует HTTP-ответ и отправляет его клиенту

Стоит отметить, что если вы оказываете платную услугу и нарушаете условия SLA, то вам, возможно, придется вернуть часть денег своим клиентам, что, конечно же, было бы нежелательно.

Чтобы не усложнять пример, условимся, что анализ эмоциональной окраски выполняется только для одной компании за раз (в одном HTTP-запросе может быть указана только одна компания), только раз в сутки и только на основе данных предыдущего дня. Например, такую информацию, собранную вчера, могли бы использовать инвесторы перед открытием фондового рынка, чтобы решить, стоит ли вкладываться в ту или иную компанию. Предположим также, что вы предоставляете услугу прогнозирования только ведущим компаниям из списка Fortune 100.

7.2. ПЕРВАЯ ПОПЫТКА: ЗАПОМИНАНИЕ ЗНАЧЕНИЙ

Итак, мы не можем позволить себе вычислять промежуточные результаты снова и снова при получении каждого запроса — это слишком дорого. Поэтому первым естественным решением будет сохранить эти результаты после их вычисления и просто возвращать их, когда они снова понадобятся.

Нам повезло, потому что функция, создающая сводку, принимает только один аргумент — имя (или внутренний идентификатор) компании из относительно узкого круга[1]. Поэтому можно легко вычислить нужные промежуточные результаты и просто возвращать их при повторных обращениях. Если приложение работает довольно долго без перезапуска, то после начального периода «разогрева», в течение которого происходит фактическое вычисление значений (и, следовательно, имеет место высокая задержка), мы получим достаточно промежуточных результатов, чтобы быстро отправлять ответы на запросы без дополнительных затрат времени на выполнение HTTP-запросов к социальным сетям.

Взгляните на рис. 7.2, где показано, как эти условия меняют архитектуру приложения, и познакомьтесь с кэшем — нашим рыцарем в сияющих доспехах, который спасает нас от нарушения условий SLA и банкротства.

Чтобы избавиться от мелких деталей, связанных с HTTP и не особенно важных для анализа, абстрагируем их за ширмой волшебной функции, которую мог бы вызывать наш анализатор эмоциональной окраски и получать от нее все нужные данные, собранные из всех социальных сетей. На рис. 7.3 показана упрощенная архитектура такого анализатора эмоциональной окраски.

Теперь она выглядит намного чище, верно? Напомню, что вся сложность скрыта за «генератором эмоциональной окраски», потому что основной интерес для нас сейчас представляет механизм кэширования, а не то, как вычисляется эта окраска. Тем не менее не надо думать, что кэширование может быть только в памяти или использоваться

[1] А именно: из набора допустимых аргументов функции. Как уже говорилось, в этом примере мы ограничиваемся только ведущими компаниями из списка Fortune 100.

лишь в автономных приложениях. Напротив, веб-приложения являются, пожалуй, одним из лучших мест для добавления некоторых слоев кэширования: подробный пример, с которого мы начали, должен был прояснить это.

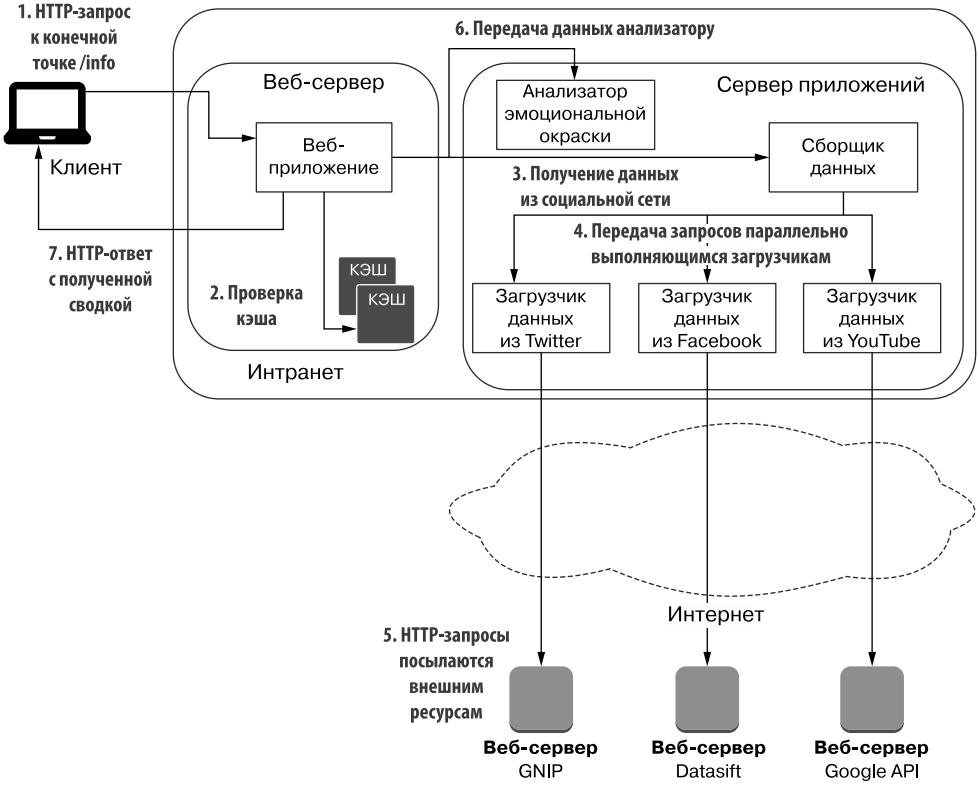


Рис. 7.2. Кэш, внедренный перед вызовом сервера приложений, позволяет веб-серверу проверить наличие уже вычисленных результатов и, если такие результаты есть, пропустить шаги 3–6. Обратите внимание, что кэш можно добавить и перед отдельными загрузчиками данных, если допускается вычисление результатов по неполным данным (например, когда одна из внешних служб недоступна). Теоретически кэш можно добавить в обработчик данных, делегировав ему связь со сборщиком данных, чтобы веб-сервер напрямую обращался только к обработчику. Однако наличие кэша на веб-сервере (физически на том же компьютере) имеет ряд преимуществ

Важно также помнить, что сокрытие сложностей путем абстрагирования низкоуровневых деталей является основной техникой анализа алгоритмов. Например, часто предполагается, что некоторые контейнеры хранят целые числа, чтобы не беспокоиться обо всех возможных типах хранимых данных. Даже сама модель ОЗУ, представленная в приложении Б, упрощена, чтобы скрыть детали бесчисленного множества типов данных, которые могут хранить и обрабатывать реальные компьютеры.

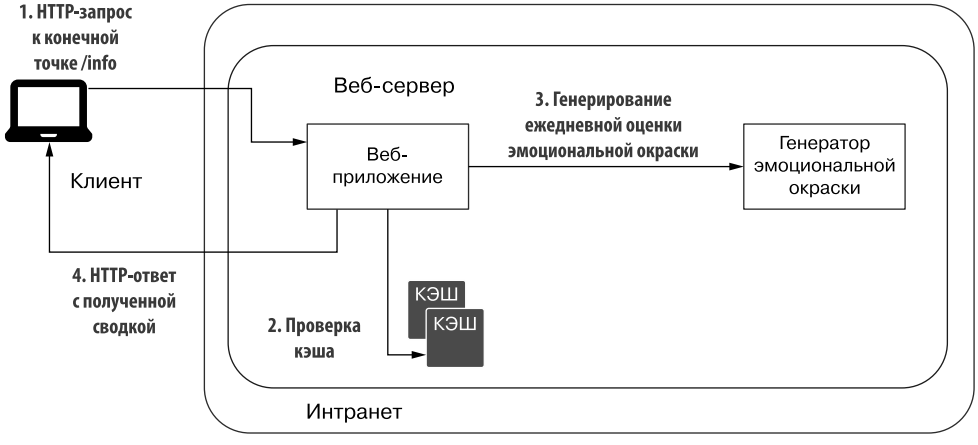


Рис. 7.3. Архитектура примера приложения после абстрагирования всех деталей, связанных с вычислением сводки, в специальном компоненте, генераторе эмоциональной окраски. Она, как нетрудно представить, размещается на веб-сервере и вызывается синхронно из веб-приложения

7.2.1. Описание и API

Как обычно, ниже приводится краткое описание API и контракта, которому должна соответствовать наша структура данных.

Абстрактная структура данных: кэш	

API	`class Cache {`
	`  init(maxSize);`
	`  get(key);`
	`  set(key, value);`
	`  getSize();`
	`}`

Контракт с клиентом	Кэш хранит определенное количество записей (аргумент `maxSize` конструктора в API). Он позволяет создавать новые записи и хранит элементы в соответствии с выбранной политикой вытеснения*. При вызове `set`, если запись с таким ключом уже существует в кэше, новое значение затрет старое

* Политика вытеснения используется в кэшах, чтобы решить, какой элемент следует удалить с целью освободить место для новой записи.

7.2.2. Освежите данные, пожалуйста

Теперь, получив первое решение для нашего конкретного примера, можно задаться вопросом, насколько оно применимо в общем случае. Чтобы ответить, нужно обсудить некоторые моменты, которые до сих пор оставались скрытыми от наших глаз. Прежде всего стоит определить, являются ли вычисления статическими, изолированными и независимыми от времени, как, например, вычисление произведения

матриц или сложного интеграла. Если да, то можно считать, что нам повезло, потому что результаты можно вычислить один раз (для каждого входного аргумента) и они будут действительны всегда.

Однако может случиться так, что результаты некоторых вычислений зависят от времени (например, от дня недели или месяца) от или любого другого внешнего фактора (например, от совокупности ежедневных заказов, которая будет меняться по мере размещения новых заказов и изменения даты). В этом случае нужно быть осторожными с повторным использованием вычисленных значений, потому что они могут *устареть*. То есть при некоторых условиях значение, однажды вычисленное и сохраненное в кэше, может потерять свою актуальность.

Что делать с устаревшим кэшем, зависит от контекста. В некоторых случаях, когда допустима определенная погрешность, даже устаревшие значения могут дать приемлемое приближение. Например, если вычисляются агрегированные данные о ежедневных продажах и отображаются на графике в реальном времени, то может удовлетворить синхронизация данных каждую минуту или каждые 10 мин. Такой выбор позволит избежать повторного вычисления всех агрегатов (возможно, сгруппированных по товарам или магазинам) при каждом отображении диаграммы в инструменте отчетности, который может использоваться десятками сотрудников одновременно.

Вероятно, лучший пример, объясняющий, почему такое «контролируемое устаревание» может быть приемлемым, — это веб-кэши. Стандарты HTTP позволяют серверу (поставляющему контент) добавлять заголовки[1] к ресурсам (веб-страницам, изображениям, файлам и т. д.), чтобы сообщить клиенту (обычно браузеру), когда можно прочитать тот или иной ресурс из кэша, а также чтобы уведомить промежуточные узлы кэша, можно ли[2] кэшировать ответ на HTTP-запрос (обычно GET) и как долго его следует хранить в кэше.

И наоборот, в критичных ко времени и безопасности приложениях, таких как приложение мониторинга атомной электростанции (этот пример хорошо передает представление о критичности...), приблизительные оценки и устаревшие данные (разве что на несколько секунд) просто недопустимы.

Другие открытые вопросы: почему устаревают данные и есть ли способ быстро узнать, что это произошло? Имеется несколько альтернатив. Если данные устаревают просто с течением времени, то при записи в кэш вместе с данными можно сохранить отметку времени и проверять ее при чтении. Если данные окажутся слишком старыми, чтобы считаться актуальными, то можно выполнить повторные вычисления и обновить

---

[1] В идеале все HTTP-ответы должны добавлять заголовки Cache-Control, Expires и Last-Modified, чтобы гарантировать максимально возможное использование ресурсов из кэша.

[2] Не все данные можно распространять без ограничений. Вот почему существуют приватные кэши, в которых данные доступны единственному пользователю/IP, и публичные кэши (используемые, например, для кэширования статического анонимного контента), доступные всем, кто интересуется ресурсом.

кэш. Если есть другие внешние условия, например изменение конфигурации или какие-то события — получение нового заказа или изменение требуемой точности вычислений, то можно просто проверить эти условия. Подобные проверки обходятся обычно намного дешевле, чем повторное вычисление с нуля.

В оставшейся части главы мы упростим задачу, предположив, что данные в кэше не устаревают. Обработку устаревшего содержимого можно рассматривать как ортогональную задачу, улучшающую базовый механизм кэширования независимо от выбранных способов реализации этого механизма.

7.2.3. Обработка асинхронных вызовов

Признание устаревания данных и обсуждение путей решения этой проблемы будет хорошей отправной точкой для обобщения решений кэширования не только в нашем тщательно продуманном примере. Следующим шагом является обобщение модели вычислений. До сих пор предполагалось, что вычисления выполняются в однопоточном окружении, поэтому вызовы обрабатываются синхронно, по очереди, один за другим. Это не только нереально, но и слишком расточительно. Реальные приложения не могут позволить себе задержку, из-за которой клиенты будут ждать, пока обрабатываются предыдущие вызовы. Они могут запустить множество копий конвейера обработки, как показано на рис. 7.4, и каждая будет иметь свой кэш и сможет обрабатывать по одному вызову за раз. Кэширование в этом случае не будет максимально эффективным, потому что если два потока получат один и тот же запрос, то ни один из них не сможет прочитать результат из кэша. Более того, если предположить, что генератор эмоциональной окраски работает синхронно, то обработка запросов по одному замедлит приложение еще больше.

Рассмотрим любой из потоков, изображенных на рис. 7.4, и предположим, что в кэше уже имеются промежуточные результаты для Twitter и Facebook, а генератор эмоциональной окраски получает еще четыре запроса для Twitter, Facebook и Google. Чтобы получить ответ на первый из новых запросов, придется вычислить все с нуля, но для следующих двух все необходимые данные уже будут в кэше и ответы на них можно вернуть сразу же. Однако в синхронной архитектуре пришлось бы ждать окончания обработки первого запроса, прежде чем переходить к следующим двум (и всем остальным, которые могут накопиться за это время). В асинхронной конфигурации веб-приложение может асинхронно вызывать генератор эмоциональной окраски для каждого запроса, «объединять» с уже имеющимися промежуточными результатами и возвращать окончательный результат, как только вся необходимая информация будет собрана. Но что произойдет, если в последовательности примеров на рис. 7.4 второй запрос для Google начнет обрабатываться до того, как первый будет обработан?

Если первый запрос еще не обработан, информация о Google не попадет в кэш. Поэтому, когда поступит второй запрос, веб-приложение обнаружит промах кэша и вызовет генератор эмоциональной окраски, чтобы получить данные из социальной сети и вычислить эмоциональную окраску.

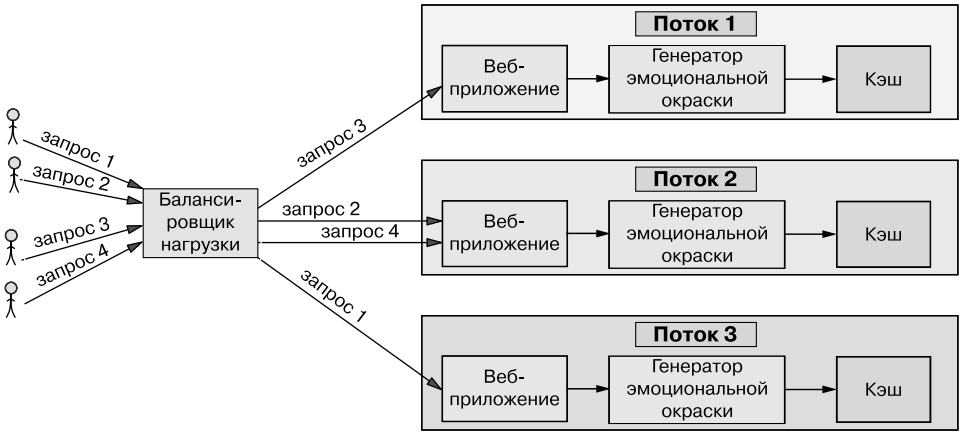


Рис. 7.4. Возможная конфигурация приложения, использующего отдельные кэши в каждом потоке. Каждый поток содержит полный конвейер (веб-сервер, генератор эмоциональной окраски и кэш) и обрабатывает по одному запросу за раз, работая синхронно. Балансировщик нагрузки распределяет запросы между несколькими запущенными потоками, ожидая ответа каждого веб-сервера, прежде чем передать следующий запрос (на рисунке веб-приложение в потоке 2 вернет ответ на вызов 2 до того, как балансировщик нагрузки сможет передать ему запрос 4)

Затем результат запроса, обработанного первым, будет сохранен в кэше. И когда наконец будет выдан результат в ответ на другой запрос, он также попытается сохранить этот результат в кэше. В зависимости от реализации кэш может просто затереть старый результат новым (они могут отличаться по многим причинам), отбросить новый результат или в худшем случае создать дубликат[1].

Но, независимо от особенностей обработки конфликтов, хуже всего то, что результат, который можно было бы получить из кэша, вычисляется дважды. Это, в свою очередь, означает излишне высокую задержку, а в некоторых случаях и дополнительные расходы. Например, если поставщики данных взимают плату за каждую попытку извлечь информацию из социальных сетей.

7.2.4. Маркировка значений в кэше признаком «загружается»

Найти идеальное решение, чтобы предотвратить возникновение условия гонки непросто, а иногда фактически невозможно. В данном случае, например, если рассмотрим детализированную архитектуру на рис. 7.2, заметим несколько HTTP-запросов, которые могут иметь переменную задержку или вообще не выполняться. Поэтому нельзя точно сказать, какой из двух запросов для получения оценки эмоциональной окраски по данным из Google будет обработан первым. Разумно предположить, что первый запрос обработается первым, но если так случится, рискует уменьшиться

[1] Мы еще вернемся к этому примеру в разделе 7.7. Его иллюстрирует рис. 7.12.

эффективность для случая, когда по какой-то причине обработка первого запроса сборщиком данных[1] продлится дольше среднего. Однако еще хуже окажется такой сценарий: если будет задано, что второй запрос к конечной точке `/sentiment` должен ждать и использовать результат, вычисленный для первого запроса, а при этом запрос сборщика данных по какой-то причине завершится неудачей, то в результате оба веб-запроса не будут обработаны.

И все же можно и должно стараться работать лучше, не расходуя ресурсы впустую. Чтобы решить проблему, описанную в предыдущем подразделе, резонно было бы добавить признак «загружается» к значениям в кэше, которые в данный момент вычисляются. Когда первый запрос `/sentiment/Google` проверит кэш и обнаружит промах, он создаст запись для Google и добавит в нее признак «обрабатывается».

Когда второй вызов проверит кэш, он будет знать, что значение уже вычисляется. После этого он может выполнять опрос кэша с некоторой периодичностью, проверяя готовность результатов, либо использовать механизм «издатель — подписчик» и подписаться на события кэша, чтобы сообщить ему о своей заинтересованности в получении результатов из Google. А кэш уведомит всех подписчиков на это событие, как только результаты будут вычислены и сохранены в кэше.

В любом случае невозможно полностью избавиться от проблемы гонки, упомянутой здесь. Но риск небольшой дополнительной задержки во многих случаях оправдан экономией, которую мы получим, избегая повторного вычисления значений, уже рассчитанных.

7.3. НЕХВАТКА ПАМЯТИ (БУКВАЛЬНО)

Пришло время ненадолго приостановиться и подвести некоторые итоги:

- рассматриваемая задача сложна, она генерирует множество промежуточных результатов, которые можно использовать повторно;
- для реализации повторного использования нужен механизм хранения промежуточных результатов, чтобы вычислять их только один (или по крайней мере как можно меньшее количество) раз.

На первый взгляд кажется, что простое решение где-то рядом. Для нашего примера, вероятно, так и есть: задействовано очень небольшое количество записей в кэше (максимум 100) и в каждой сохраняется только вычисленная оценка эмоциональной окраски. Похоже, что для хранения такого кэша потребуется совсем небольшой объем памяти.

Однако представьте, что произойдет, если потребуется создать кэш для сборщика данных на рис. 7.2, где будут храниться в исходном виде все сообщения из основных социальных сетей, в которых упоминаются названия компаний? Учтите также, что для «самых крутых» компаний могут быть миллионы таких сообщений и для

[1] Этот программный компонент показан на рис. 7.1 и 7.2.

каждого в среднем потребуется несколько килобайтов (при условии, что медиафайлы сохраняются в виде ссылок или не сохраняются вообще). Даже для сотни компаний такой кэш может занять несколько гигабайтов.

Аналогично представьте, как будут работать приложения. Когда вы попытаетесь получить доступ к своей стене, алгоритм выявит сообщения с наиболее высокими оценками, чтобы показать вам. Он будет основываться на стенах ваших друзей, ваших предпочтениях, страницах, на которые вы подписаны, и т. д.

Это довольно сложный и ресурсоемкий алгоритм, поэтому желательно максимально кэшировать его результаты, так как, вероятно, нет смысла повторно пересчитывать ленту, если тот же пользователь зайдет на свою стену через одну или пять минут (приложение также может иметь механизм добавочного обновления, отображающий новые сообщения, только когда они публикуются, но это уже другая история и не относится к нашей теме).

Теперь представьте механизм кэширования для миллиарда стен: даже если хранить в кэше только идентификаторы (обычно несколько байтов) для 50 лучших сообщений на стене, то такому кэшу все равно потребуются терабайты.

Эти примеры показывают, как легко достичь размера кэша, который трудно или невозможно уместить в ОЗУ. Можно, конечно, использовать базы данных NoSQL или распределенные кэши, такие как Memcached, Redis или Cassandra, но даже в этом случае чем больше записей хранится в кэше, тем медленнее он будет работать. Это станет очевидно после того, как мы обсудим реализацию типичного кэша.

Можно также представить ситуацию, когда все входные данные различны и для них требуется создавать новые записи в кэше. Очевидно, что нельзя до бесконечности добавлять: в кэше, как в любой конечной системе, рано или поздно наступит момент насыщения.

Поэтому, чтобы добавить новые записи после заполнения кэша, нужно удалить некоторые из существующих.

Возникает вопрос: какие записи удалить? Ответ на этот вопрос выглядит каким-то выходящим за рамки здравого смысла: в идеале желательно удалить записи, которые больше не будут запрашиваться или по крайней мере будут запрашиваться реже.

К сожалению, несмотря на возросшие возможности прогнозирования и наблюдаемый поразительный прогресс в области ИИ, компьютеры не обладают даром предвидения, поэтому невозможно точно предсказать, какие элементы будут востребованы чаще, а какие реже[1].

Конечно, есть некоторые предположения, на основе которых можно попытаться сделать обоснованные догадки. Вот одно из них: если запись не использовалась

[1] Конечно, существует множество методов, включая машинное обучение, способных давать хорошие прогнозы, но всегда с некоторой вероятностью ошибки, особенно высокой в области, подобной этой, где поведение пользователей может быстро и неожиданно меняться.

в течение длительного времени, она имеет меньшую ценность, чем запись, к которой обращались недавно. В зависимости от размера кэша и времени, прошедшего с момента последнего обращения к записи, можно сделать вывод, что самая старая запись, вероятно, устарела и в ближайшее время не понадобится или даже никогда не понадобится.

С другой стороны, в некоторых случаях может иметь место обратное: записи, не использовавшиеся долгое время, очень скоро потребуются. Однако в этом случае удаление более свежих элементов обычно не дает желаемого эффекта и необходимы другие критерии.

Но, принимая во внимание только время последнего обращения, можно не учесть другую полезную информацию. Например, если к записи обращались несколько раз, но не в последние несколько минут, то все более новые записи, даже если к ним обращались только один раз, будут считаться более ценными. Однако подумайте: запись, востребованная только однажды, на самом деле была вычислена, но никогда не извлекалась из кэша, поэтому ее кэширование еще не принесло никаких преимуществ. Если в течение какого-то времени к этой записи не случилось ни одного обращения, то полезность ее хранения становится сомнительной. То есть еще одним решением может быть присваивание записям значения, определяющего частоту обращения к ним.

Подробные обсуждения этих рецептов и реализации соответствующих кэшей — в следующих нескольких разделах.

7.4. УДАЛЕНИЕ УСТАРЕВШИХ ДАННЫХ: КЭШ LRU

Первый тип кэша, или, скорее, первая политика вытеснения, которую рассмотрим, — это кэш LRU (Least Recently Used — «наиболее давно использовавшиеся»), вытесняющий *наиболее давно использовавшиеся* записи.

Что необходимо для реализации этой структуры данных? Вот операции, которые нужно оптимизировать:

- сохранение записи с названием компании (очевидно);
- проверка наличия сохраненной записи для заданной компании;
- получение записи по названию компании;
- получение количества хранимых элементов;
- получение самой старой записи и ее удаление из кэша.

Как объяснялось в разделе 7.3, эта структура данных способна одновременно хранить только определенное количество элементов. Фактический размер кэша можно определить при создании и даже, в зависимости от реализации, изменять динамически. Обозначим через n количество элементов, на которое рассчитан кэш. Когда кэш заполнится, для добавления новой записи нужно будет удалить одну из

существующих, и в данном случае удаляться будет наиболее давно использовавшаяся запись (отсюда и название кэша).

Теперь подумаем, какая из основных структур данных, знакомых нам к данному моменту, способна гарантировать наилучшую производительность.

Следить за размером кэша легко, для этого можно определить переменную и обновлять ее за постоянное время. Далее не будем больше упоминать эту операцию. Проверка наличия записи основана на поиске по названию компании, поэтому анализировать надо только поиск.

Необходимость сохранения и извлечения записей по названиям компаний должны навести вас на мысль. Ведь это явно то, для чего предназначен ассоциативный массив, поэтому очевидным выбором могла бы стать хеш-таблица. Однако хеш-таблицы неблестяще проявляют себя, когда дело доходит до извлечения минимального (или максимального) элемента[1].

На самом деле удаление самого старого элемента может потребовать линейного времени, и если ожидаемое время жизни кэша не настолько короткое, чтобы в среднем он не успевал заполниться, то эта операция может замедлить вставку новых записей.

Использование массивов тоже не выглядит хорошим решением, потому что они не позволяют ускорить ни одну из операций (в лучшем случае только одну), требуя при этом, чтобы все необходимое пространство было выделено с самого начала.

Связанные списки кажутся многообещающими для сохранения порядка вставки, но они плохо подходят для поиска записей.

Если вы прочитали приложение B, то, возможно, вспомните, что существует структура данных, предлагающая компромисс между всеми этими различными операциями: сбалансированные деревья! Используя дерево, можно гарантировать выполнение всех перечисленных операций за логарифмическое время в худшем случае.

Таблица 7.1. Сравнение производительности кэшей на n элементов при разных реализациях

	Массив (несортированный)	**Массив (сортированный)**	**Связанный список**	**Хеш-таблица**	**Сбалансированное дерево**
Сохранение записи	O(1)	O(*n*)	O(*n*)	O(1)*	O(log *n*)
Поиск записи по названию	O(*n*)	O(log *n*)**	O(*n*)	O(1)*	O(log *n*)
Удаление самой старой записи	O(*n*)***	O(*n*)	O(1)	O(*n*)	O(log *n*)

* Амортизированное время.

** При использовании поиска методом дихотомии.

*** На поиск записи требуется время O(1), но, учитывая статический характер массива, после удаления первого элемента остальные необходимо сместить на одну позицию.

[1] Чтобы освежить память, загляните в главу 2 и в приложение B.

Итак, на основе табл. 7.1 можно сделать следующие выводы:

- деревья выглядят лучшим компромиссом в общем случае;
- хеш-таблицы лучше подходят для случаев, когда размер кэша достаточно велик (и вставка новых элементов происходит нечасто) и редко требуется удаление самой старой записи;
- связанный список — допустимый вариант, если удаление старых записей важнее, чем добавление или поиск: но в таких ситуациях кэш практически бесполезен и его применение не принесет никакой пользы;
- во всех случаях для хранения n записей требуемый объем памяти равен $O(n)$.

Теперь вопрос: можно ли добиться большего?

7.4.1. Иногда важно удвоить усилия для решения проблем

Как вы понимаете, если бы ответ был отрицательным, мы вообще не задавали бы этот вопрос.

Но как именно добиться этого большего? Никакой другой структуры данных, позволяющей оптимизировать все три основные операции одновременно, не существует.

И все же... Представьте, что я сказал вам, будто можно реализовать кэш LRU, выполняющий все операции за амортизированное время[1] $O(1)$. Как?

Прежде чем прочитать решение загадки, попробуйте пару минут подумать сами, как это возможно.

Позвольте, я дам две подсказки (внимание, спойлер: можете попробовать разработать свое решение, прежде чем читать их).

1. Вы можете использовать столько дополнительной памяти, сколько хотите (но необходимо оставаться в пределах $O(n)$).
2. Тот факт, что я упомянул связанные списки, должен напомнить вам и заставить задуматься о конкретной структуре данных, которую мы видели в приложении B. Тем не менее уже показано, что этого недостаточно, если брать только это.

Оба моих намека указывают на одну и ту же идею: одной структуры данных может быть недостаточно для наиболее эффективного решения проблемы.

С одной стороны, у нас есть структуры данных, которые особенно хороши для быстрого хранения и извлечения записей. Хеш-таблицы практически чемпионы в этом. С другой стороны, хэш-таблицы ужасны, когда речь идет о поддержании порядка

[1] Технически, как мы уже знаем, правильнее сказать, что амортизированное время для n операций равно $O(n)$, поэтому отдельные операции обычно требуют постоянного времени, но некоторые могут выполняться за большее время, вплоть до $O(n)$.

вещей. Но есть другие структуры, которые очень хорошо с этим справляются. В зависимости от того, какой порядок хотелось бы сохранить, нам могут понадобиться деревья или списки. На самом деле в оставшейся части главы я покажу использование того и другого.

В свете этой новой подсказки сделайте паузу еще на минуту и попытайтесь самостоятельно представить, как заставить это работать. Можете ли вы придумать, как объединить хеш-таблицу и другую структуру данных, чтобы оптимизировать все операции с кэшем?

7.4.2. Упорядочение по времени

Оказывается, добиться этого просто. Прежде чем углубляться в материал раздела, загляните в приложение B, если подзабыли, сколько времени требуется на выполнение операций с массивами и списками, а также какова разница между одно- и двусвязными списками.

Итак, представьте, что нужно лишь поддерживать порядок записей в кэше, чтобы реализовать возможность перехода от наиболее давно к наименее давно использованным записям. Поскольку порядок следования записей зависит только от времени вставки, добавление новых элементов не меняет порядка следования старых; поэтому не нужно ничего необычного: нужна только структура, поддерживающая порядок FIFO (First In First Out — «первым пришел, первым ушел»). В роли такой структуры можно использовать список или очередь. Как вы должны помнить, если количество хранимых элементов не известно заранее или их число может динамически меняться, лучшим выбором является связанный список. Тогда как очереди часто реализуются с использованием массива (и потому часто имеют статичный размер), но оптимизированы для вставки в голову и удаления с хвоста.

Связанные списки тоже могут поддерживать быструю вставку/удаление с конца. Однако нам лучше подойдет двусвязный список (рис. 7.5), позволяющий вставлять элементы в начало и удалять из хвоста. Храня указатель на хвост и ссылки на предшественника в каждом узле, можно реализовать удаление из хвоста за время O(1).

В листинге 7.1 показан псевдокод с реализацией кэша LRU на основе связанных списков.

Когда статические очереди реализуются с использованием массивов, удается сэкономить немного памяти по сравнению со связанными списками. Она могла бы использоваться в списках для хранения ссылок на узлы и объекты узлов. Однако это лишь деталь реализации, зависящая от используемого языка, которая не меняет порядка объема используемой памяти: в обоих случаях объем будет равен O(n).

Итак, что же выбрать: списки или очереди? Еще немного подумаем об организации. До сих пор рассматривались хеш-таблица и связанный список по отдельности, но нужно заставить их работать синхронно.

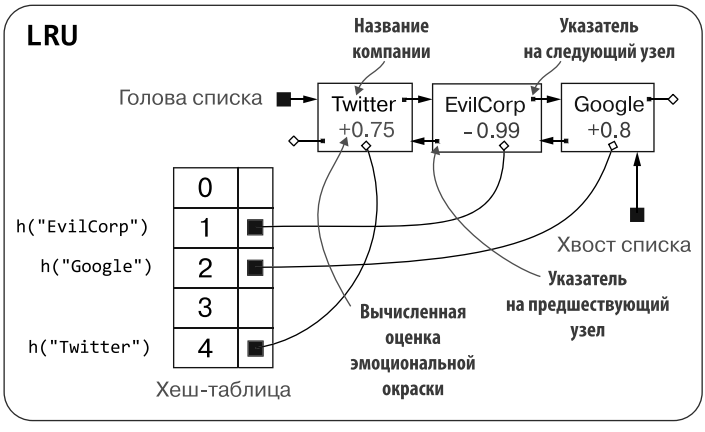


Рис. 7.5. Структура кэша LRU. Как можно видеть, элементы дерева, хранящего кэш, должны обновляться после каждой операции: 1) хеш-таблица; 2) голова двусвязного списка; 3) указатель на последний элемент в списке. Обратите внимание, что каждый элемент в хеш-таблице ссылается на узел в списке, хранящем данные. Чтобы перейти от узла в списке к соответствующей записи в хеш-таблице, нужно хешировать название компании, хранящееся в узле, и получить ключ в таблице. Для простоты разрешение конфликтов на этом и следующих рисунках не рассматривается

Листинг 7.1. Конструкция кэша LRU

```
class LRUCache
  #type integer
  maxSize
  #type HashTable
  hashTable
  #type LinkedList
  elements
  #type LinkedListNode
  this.elementsTail = null

  function LRUCache(maxElements)
    maxSize ← maxElements
    hashTable ← new HashTable(maxElements)
    elements ← new LinkedList()
    elementsTail ← null
```

Сигнатура конструктора для объекта LRUCache. Ему передается максимальное количество элементов, которые должен хранить кэш

Запоминаем, сколько записей может храниться в кэше

Инициализируем хеш-таблицу (передача максимального количества элементов поможет настроить внутренние параметры таблицы)

Инициализируем связанный список для хранения элементов кэша (в первый момент список пуст)

Список пуст, поэтому указатель на последний элемент равен null

В кэше могут храниться очень большие объекты, было бы нежелательно дублировать их в обеих структурах данных. Один из способов избежать дублирования — хранить записи только в одной структуре и ссылаться на них из другой. Можем добавлять записи в хеш-таблицу, а в другой структуре сохранять только ключ, или наоборот.

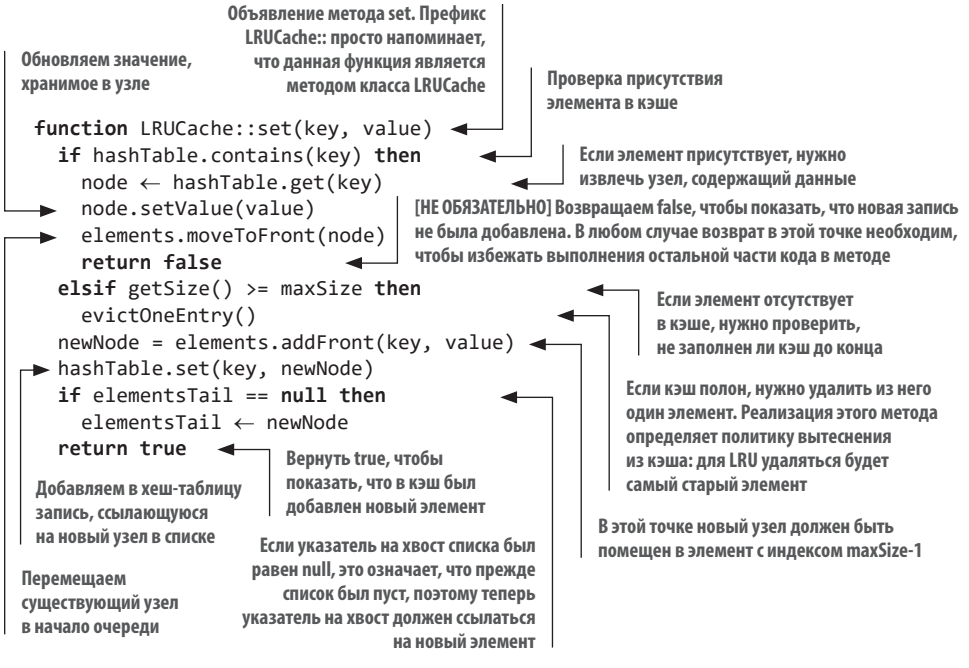
И связанные списки, и очереди поддерживают оба способа. Тем не менее будем использовать список; причина такого решения станет ясна чуть позже.

Также нужно решить, какая структура данных должна хранить значения, а какая — ссылки. Я утверждаю, что наилучший выбор — использовать хеш-таблицу для хранения ссылок на узлы списка, а в списке хранить фактические значения, и основная причина именно этого предпочтения та же, что при выборе между списками и очередями[1]. Она обусловлена ситуацией, которую мы еще не рассмотрели.

Итак, имеем кэш *least recently used* (наиболее давно использовавшиеся). Это не означает наиболее давно *добавленные*. То есть порядок основан на времени не только добавления элемента в кэш, но и последнего обращения к нему (эти времена могут совпадать, например, в записях, которые никогда не использовались повторно, но в большинстве случаев они различаются).

А теперь перейдем к методу `set` (листинг 7.2), добавляющему новую запись. При *промахе кэша*, то есть при неудачной попытке доступа к элементу, отсутствующему в кэше, он просто добавляет новую запись в начало связанного списка, как показано на рис. 7.6.

Листинг 7.2. Метод set кэша LRU



[1] Это еще не все. Если отдать предпочтение противоположному решению, то способ связывания узла списка с записью в хеш-таблице будет зависеть от реализации хеш-таблицы. Им может быть индекс для прямой адресации или указатель, если в хеш-таблице используются цепочки. Такая зависимость от реализации является признаком плохого проектирования и иногда просто нереализуема, ведь мало у кого есть возможность получить доступ к внутренней реализации стандартной библиотеки (по весьма уважительной причине!).

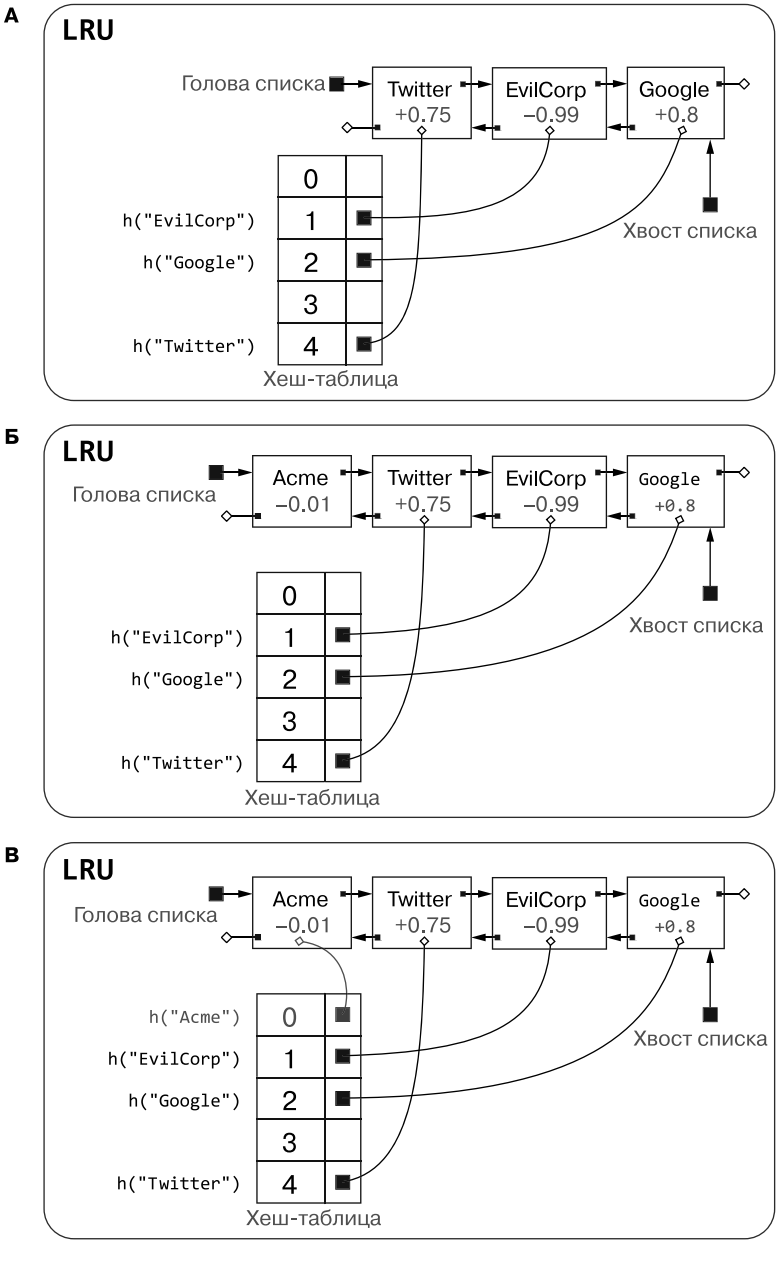


Рис. 7.6. Добавление при промахе кэша. *А*. Кэш перед добавлением нового элемента. В этом примере мы ищем компанию Acme и получаем промах кэша. *Б*. Добавляем в начало списка новый узел для компании Acme. *В*. Создаем новую запись в хеш-таблице и обновляем указатель на голову списка

Но, когда мы сталкиваемся с *попаданием в кэш*, как показано на рис. 7.7, то есть с удачной попыткой доступа к существующему элементу, узел, соответствующий этому элементу, нужно переместить в начало списка. Это можно сделать эффективно, только если есть возможность за постоянное[1] время извлечь указатель на существующий узел связанного списка (который может быть где угодно в списке) и удалить элемент из списка (для этого как раз и нужен двусвязный список, потому что при использовании очереди, реализованной на основе массива, удаление из середины очереди займет линейное время).

Если кэш полон, нужно сначала удалить наиболее давно использовавшуюся запись и только потом добавлять новую. В нашем случае метод удаления самой старой записи, представленный в листинге 7.3 и проиллюстрированный на рис. 7.8, может за постоянное время получить доступ к хвосту связанного списка, где и находится запись, которую нужно удалить. Чтобы найти соответствующую запись в хеш-таблице и удалить ее, нужно получить ее хеш (или идентификатор), затратив дополнительное время (потенциально непостоянное: для строк это время будет зависеть от длины строки).

Листинг 7.3. Метод evictOneEntry (приватный) кэша LRU

```
function LRUCache::evictOneEntry()
  if hashTable.isEmpty() then
    return false
  node ← elementsTail
  elementsTail ← node.previous()
  if elementsTail != null then
    elementsTail.next ← null
  hashTable.delete(node.getKey())
  return true
```

Проверка — не пустой ли кэш

Если кэш пустой, возвращаем признак неудачи

Обновляем указатель на последний элемент в списке. Инвариант: если кэш не пустой, то и указатель на хвост будет не равен null

Если указатель на хвост не равен null, то присваиваем указателю на следующий элемент в хвосте значение null (этот указатель в последнем элементе в списке всегда должен быть равен null)

Удаляем запись из хеш-таблицы

Возвращаем признак успеха

В API осталась еще пара методов, которые мы не обсудили: `get(key)` и `getSize()`. Однако они имеют простую реализацию, потому что фактически являются всего лишь обертками для одноименных методов хеш-таблиц, по крайней мере если значения сохранять непосредственно в хеш-таблице. Если значения должны храниться в связанном списке, а в хеш-таблице — только указатели на узлы списка, то в `get` нужно разрешить эту дополнительную косвенность, как показано в листинге 7.4.

[1] Напомню, что еще нужно включить время вычисления каждого значения хеш-функции для искомой записи. Дополнительную информацию по этой теме вы найдете в приложении B.

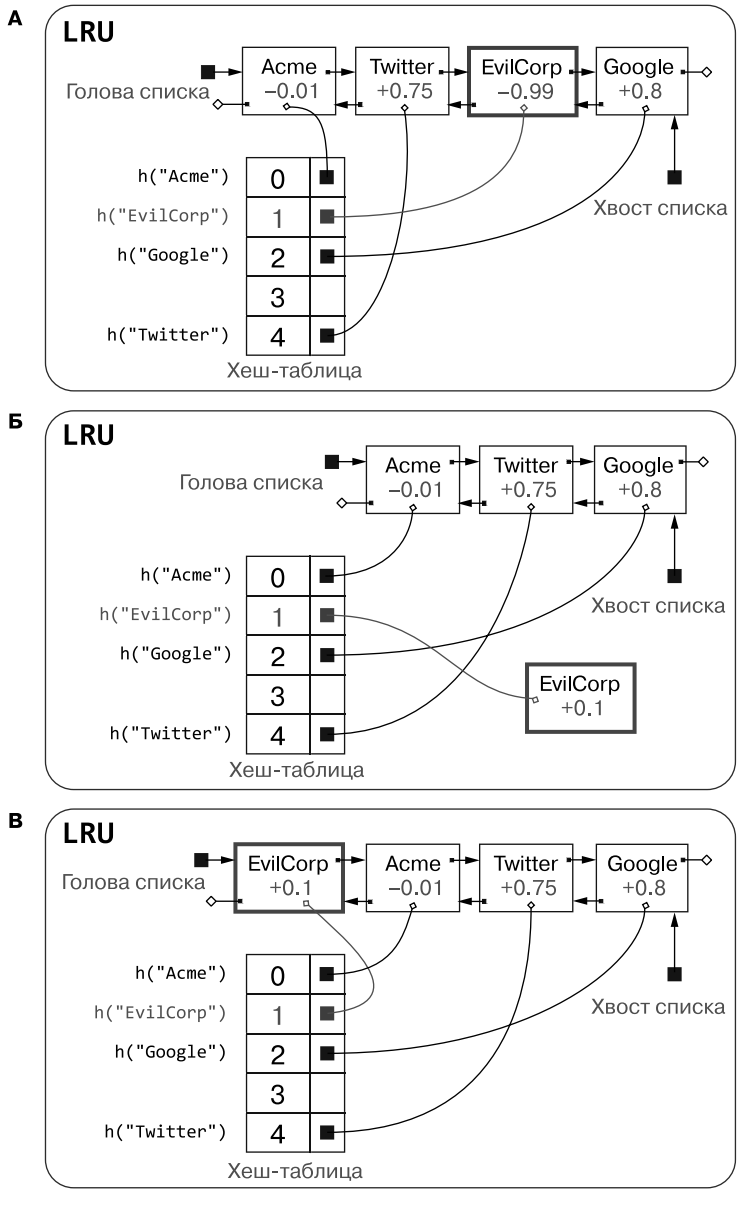


Рис. 7.7. Обновление записи при попадании в кэш. *А*. Исходное состояние кэша. В этом случае запись "EvilCorp" для обновления была найдена и мы получили попадание в кэш. Поиск выполняется в хеш-таблице, поэтому на рисунке выделены сама запись и ссылка на EvilCorp. *Б*. Узел с информацией о EvilCorp в списке обновляется новыми данными (похоже, что EvilCorp потратила немного денег, чтобы улучшить свою репутацию!) и исключает его из списка, обновляя соответствующие ссылки. *В*. Затем узел добавляется в начало списка. Обновлять ссылку в хеш-таблице не требуется

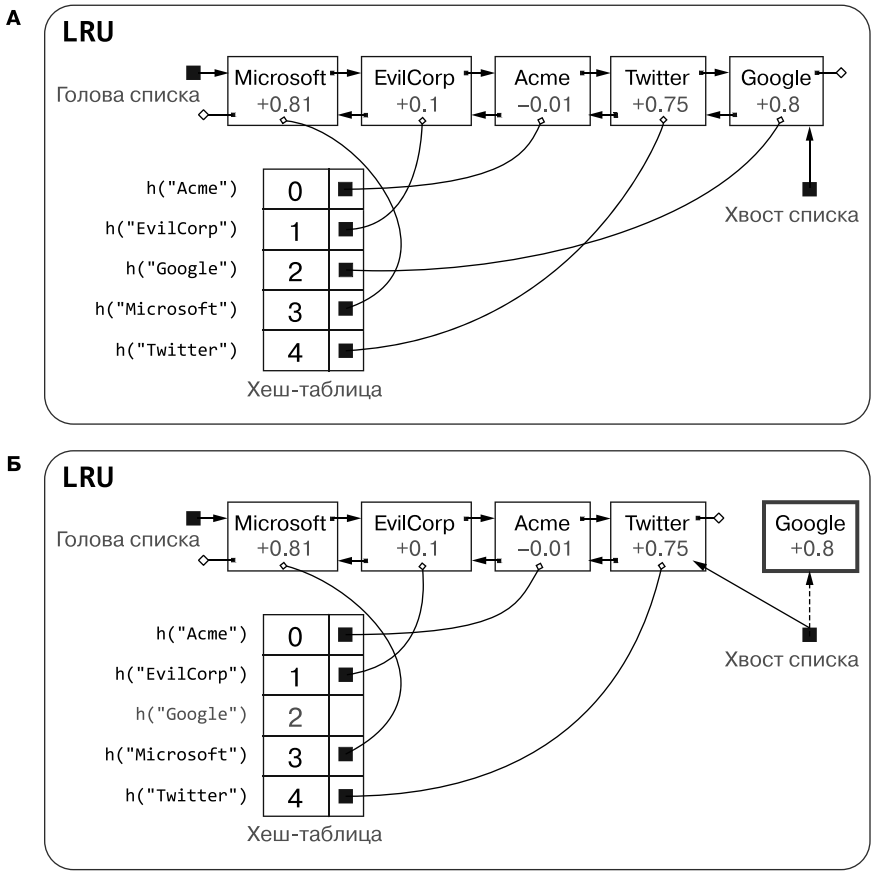


Рис. 7.8. Удаление записи из кэша (за удалением последует добавление при промахе, как показано на рис. 7.6). *А*. Исходное состояние полного кэша. *Б*. Обновляется указатель на конец списка, а также ссылки в предпоследнем узле списка и на хвост списка. Соответствующая запись в хеш-таблице также удаляется, поэтому в кэше не останется ссылок на узел с удаленной записью (в зависимости от языка программирования узел может быть утилизирован сборщиком мусора или его нужно уничтожить вручную, чтобы освободить занятую им память)

Листинг 7.4. Метод get кэша LRU

```
function LRUCache::get(key)
  node ← hashTable.get(key)
  if node == null then
    return null
  else
    return node.getValue()
```

Поиск в хеш-таблице по значению аргумента key. Результатом вызова этого метода get будет указатель на узел в связанном списке или null

Если запись не найдена, значит, искомое значение key отсутствует в хеш-таблице и, соответственно, в кэше

Иначе можно просто вернуть значение, хранящееся в узле в списке

7.4.3. Производительность

Итак, все операции, представленные в листингах выше, имеют постоянное или постоянное амортизированное (в случае хеш-таблицы) время выполнения. Это дает прирост производительности, который и был целью наших поисков!

Таблица 7.2 не что иное, как обновленная версия табл. 7.1, которая теперь включает столбец для кэша LRU.

Таблица 7.2. Сравнение производительности кэшей на n элементов с разными реализациями

	Массив (несортированный)	Связанный список	Хеш-таблица	Сбалансированное дерево	Кэш LRU
Сохранение записи	O(1)	O(*n*)	O(1)*	O(log *n*)	O(1)*
Поиск записи по названию	O(*n*)	O(*n*)	O(1)*	O(log *n*)	O(1)*
Вытеснение	O(*n*)	O(1)	O(*n*)	O(log *n*)	O(1)*

* Амортизированное время.

7.5. КОГДА БОЛЕЕ СВЕЖИЕ ДАННЫЕ ЦЕННЕЕ: LFU

Мы намекали на это в предыдущих разделах: иногда реже используемые записи нечасто запрашиваются снова... но это верно не всегда.

Может так случиться, что данные, очень востребованные в прошлом, в настоящее время временно оказываются неактуальными, но вполне вероятно, что в ближайшем будущем к ним снова будут обращаться.

Представьте интернет-магазин, обслуживающий покупателей по всему миру. Если товар особенно популярен в одной или нескольких странах, на него будет приходиться максимум запросов в часы пик в этой стране, а в другое время в той же стране к нему вообще никто не будет обращаться[1].

Другим примером может служить просто ложный пик запросов на получение определенных товаров, как показано на рис. 7.9.

В любом случае есть риск, что все закончится удалением из кэша данных, в среднем используемых чаще, потому что могут иметься более свежие данные — записи, к которым обращались совсем недавно и которые, возможно, никогда (или в течение длительного времени, дольше среднего времени жизни записи в кэше) не понадобятся снова.

---

[1] Этот пример приводится исключительно для иллюстрации. По всей вероятности, кэш, базы данных, веб-серверы и т. д. такого продавца будут сегментированы географически, как раз чтобы учесть локальные особенности предпочтений.

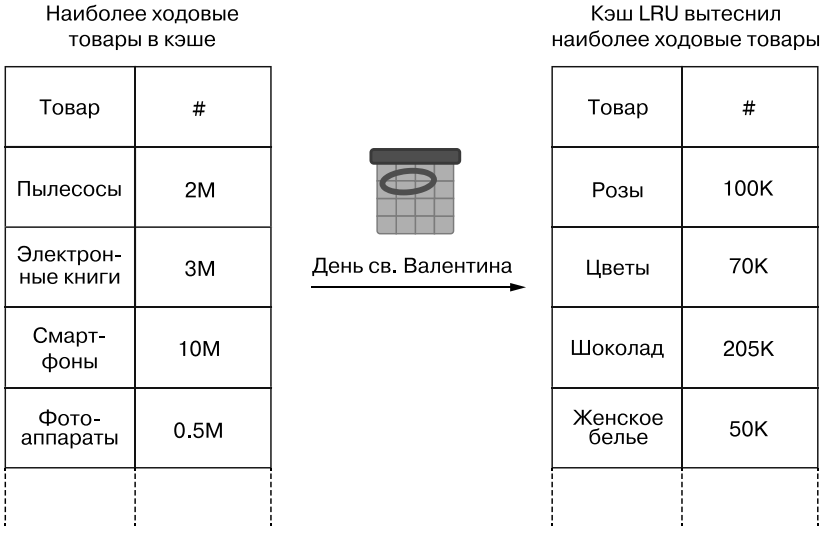


Рис. 7.9. Пример возможной ситуации, когда политика вытеснения LRU очистит кэш от самых «востребованных» записей в пользу недавно просматривавшихся и ставших популярными в часы пик. Товары на рисунке упорядочены по времени последнего просмотра; количество просмотров показано, только чтобы продемонстрировать, как самые последние просмотры в часы пик могут вытеснить из кэша самые продаваемые товары. Политика LRU не гарантирует, что наиболее продаваемые товары останутся в кэше, но более высокая частота их просмотров повышает вероятность, что они останутся в кэше (поскольку они, скорее всего, попадут в начало очереди до того, как попадут под чистку). В часы пик, если количество товаров, которые внезапно стали популярными, станет достаточно большим, они смогут заполнить кэш и вынудить его удалить более ходовые товары. Таков будет временный побочный эффект, который исчезнет после часа пик. В некоторых ситуациях это может быть желательным, потому что товары, популярные в часы пик, должны быть доступны быстрее, чем обычные бестселлеры. Как будет показано далее, кэш LRU имеет более динамичный оборот, чем LFU

В подобных случаях можно использовать альтернативную политику, основанную на подсчете обращений к записи с момента ее добавления в кэш и сохраняющую элементы с наибольшим числом обращений.

Эта стратегия очистки называется *LFU* (*Least Frequently Used* — «наименее часто используемые»). Иногда ее называют *MFU* (*Most Frequently Used* — «наиболее часто используемые»). Преимущество этой политики заключается в том, что элементы, попавшие в кэш, к которым больше никогда не обращаются, достаточно быстро удаляются из кэша[1].

[1] Это верно при правильной реализации. В частности, нужно быть осторожными при разрешении неоднозначностей, когда две записи имеют одинаковое количество обращений — в таких случаях обычно лучше удалять более старые записи.

7.5.1. Как сделать правильный выбор

Это не единственные возможные стратегии вытеснения элементов из кэша. Веб-кэш, например, может также учитывать стоимость извлечения информации в смысле задержки, времени вычислений и даже фактических затрат, когда за извлекаемые данные приходится платить внешней службе.

И это только один из множества примеров. Ключевой момент здесь — необходимость выбора лучшей стратегии в зависимости от конкретного случая и характеристик приложения. Первые прикидки можно сделать уже на стадии проектирования, но для оценки и точной настройки выбора, вероятно, придется провести профилирование и собрать статистику использования кэша (но пусть вас это не пугает, потому что существуют инструменты, которые могут сделать это автоматически).

7.5.2. Отличительные особенности LFU

Знакомство со стратегией LFU начнем не с нуля, потому что в этом нет никакого смысла, так как у нас уже есть хорошая основа для начала — кэш LRU.

Я уже говорил и повторю еще раз, что разница между двумя типами кэшей заключается в политике вытеснения. Это верно, но лишь по большей части.

Как показано на рис. 7.10, кэш LFU можно реализовать путем реорганизации кода из раздела 7.4, просто изменив политику вытеснения и, следовательно, порядок элементов в списке.

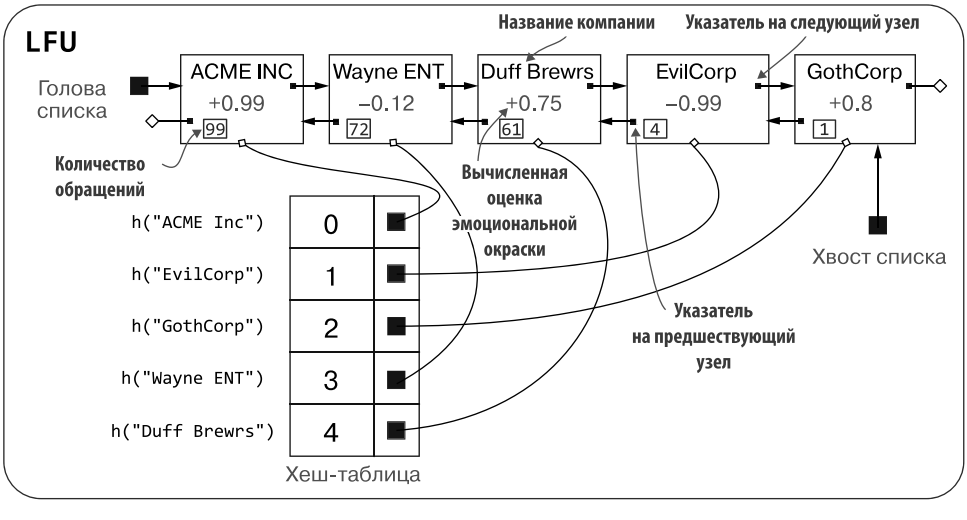


Рис. 7.10. Не самая эффективная реализация кэша LFU. Когда все элементы списка с n записями имеют одинаковое значение счетчика и происходит обращение к элементу в конце списка, кэшу придется выполнить n – 1 перестановку узлов, чтобы переместить обновленную запись в начало списка

Добавляя новую запись, нужно установить ее счетчик равным 1 и включить ее в конец списка, а при попадании в кэш нужно увеличить счетчик узла и начать перемещать его в начало списка, пока не будет найден другой узел с бóльшим счетчиком.

Эта реализация, безусловно, будет работать правильно. Но насколько эффективной она будет? Есть ли шанс улучшить производительность кэша?

Можно забыть о вставке за постоянное время: она работает только для LRU. Но в наших силах обеспечить поиск за постоянное время, и это лучше, чем ничего.

Однако добавление нового или обновление существующего элемента угрожает занять линейное время в пограничных случаях, когда большинство элементов имеют одинаковую частоту обращений к ним.

Можно ли добиться большего? И снова у вас в голове должен прозвенеть колокольчик (если этого не случилось, то загляните в главу 2 и в приложение B прежде, чем двигаться дальше).

При использовании политики LFU нельзя опираться на очереди FIFO («первым пришел, первым вышел»)[1]. Порядок вставки не имеет значения; вместо этого положение в очереди определяется приоритетом, который имеет каждый узел.

Видите, к чему я клоню? Лучший способ упорядочить записи на основе динамически меняющегося приоритета — конечно же, очередь с приоритетом. Это может быть куча — куча Фибоначчи или любая другая ее реализация. Но мы будем придерживаться абстрактной структуры данных, давая пользователям возможность самим решить, какая реализация лучше всего подходит для них в зависимости от используемого языка программирования и, конечно же, контекста использования кэша[2]. Однако для анализа производительности возьмем кучу, потому что она дает достаточно разумные гарантии производительности для всех операций в сочетании с чистой и простой реализацией.

Взгляните на рис. 7.11, чтобы увидеть, как это меняет внутреннее устройство кэша. Нужно внести в нашу реализацию только два изменения, чтобы переключиться с политики LRU на LFU:

- изменить политику вытеснения (всю логику выбора вытесняемого элемента);
- заменить структуру данных, используемую для хранения элементов.

Рассмотрим еще немного кода. Я выделил методы, которые изменились в сравнении с LRU, чтобы вам было проще увидеть различия.

---

[1] Скорее даже FIFO+, потому что элементы могут перемещаться в начало очереди в любое время.

[2] В зависимости от относительной частоты операций чтения/записи может потребоваться тонкая настройка той или иной операции.

Инициализация кэша LFU, показанная в листинге 7.5, практически идентична инициализации LRU. Здесь просто создается пустая очередь с приоритетами вместо связанного списка.

Листинг 7.5. Конструкция кэша LFU

```
function LFUCache(maxElements)
  maxSize ← maxElements
  hashTable ← new HashTable(maxElements)
  elements ← new PriorityQueue()
```

На этот раз нужно создать очередь с приоритетами. Отпала также необходимость в указателе на хвост. Предполагается, что элементы с более низким приоритетом находятся ближе к началу очереди (как в неубывающей куче)

С добавлением новой записи дело обстоит немного сложнее. Помните, выше упоминалось, что нужно быть осторожными, реализуя приоритеты? Необходимо убедиться, что из элементов с одинаковым минимальным приоритетом будет удален самый старый; иначе надо было бы удалять последнюю запись снова и снова, не давая ей возможности увеличить свой счетчик.

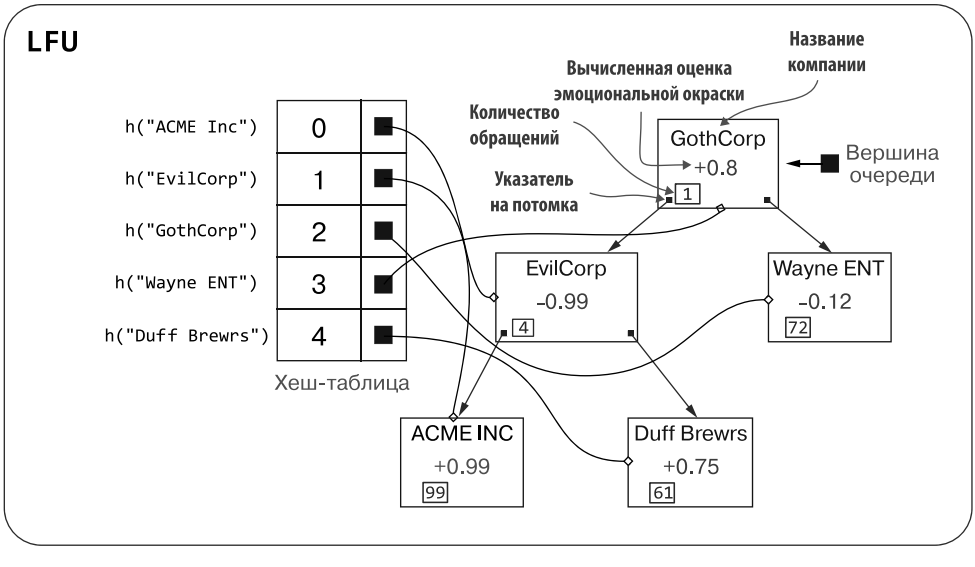


Рис. 7.11. В отличие от примера на рис. 7.10 реализация кэша LFU с использованием очереди с приоритетами для выявления элементов, подлежащих вытеснению, действует эффективнее. Здесь PriorityQueue — это абстрактный тип данных, представляющий двоичную кучу. Обратите внимание, что нам не нужно хранить отдельные указатели на точки вставки и удаления, как при использовании связанного списка

Реализация, представленная в листинге 7.6, использует в роли приоритета кортеж `<counter, timestamp>`, где `timestamp` — это время последнего обращения к записи. Оно

используется для разрешения неоднозначности при совпадении значений счетчиков; чем больше значение счетчика и чем ближе (читай: больше) к настоящему находится отметка времени, тем выше приоритет.

Листинг 7.6. Метод set кэша LFU

```
function LFUCache::set(key, value)
  if hashTable.contains(key) then
    node ← this.hashTable.get(key)
    node.setValue(value)
    node.updatePriority(new Tuple(node.getCounter() + 1, time()))
    return false
  elsif getSize() >= maxSize then
    evictOneEntry()
  newNode ← elements.add(key, value, new Tuple(1, time()))
  hashTable.set(key, newNode)
  return true
```

Проверяем отсутствие записи в кэше (в этом случае возвращается объект другого типа)

На этот раз нельзя просто переместить узел в начало списка; нужно увеличить счетчик на единицу, обновить время последнего обращения и постепенно перемещать запись вниз к концу очереди

Добавляем новую запись в очередь с приоритетами. Здесь счетчик равен 1, что является признаком новой записи

Наконец, нужно обновить реализацию метода вытеснения, как показано в листинге 7.7. Использование очереди с приоритетами делает ее еще проще, потому что все детали удаления элемента и обновления очереди инкапсулированы в самой очереди с приоритетами.

Листинг 7.7. Метод evictOneEntry (приватный) кэша LFU

```
function LFUCache::evictOneEntry()
  if hashTable.isEmpty() then
    return false
  node ← elements.pop()
  hashTable.delete(node.getKey())
  return true
```

Мы просто удаляем верхний элемент из очереди с приоритетами. Инвариант: если хеш-таблица не пустая, то и очередь не должна быть пустой

7.5.3. Производительность

Еще раз обновим таблицу с оценками производительности, добавив в нее колонку для кэша LFU. Как показано в табл. 7.3, поскольку все операции записи в кэше LFU предполагают модификацию кучи, гарантировать постоянное время их выполнения больше невозможно нигде, кроме поиска.

Таблица 7.3. Сравнение производительности кэшей на n элементов с разными реализациями

	Массив	Список	Хеш-таблица	Дерево	Кэш LRU	Кэш LFU
Сохранение записи	O(1)	O(*n*)	O(1)*	O(log *n*)	O(1)*	O(log *n*)*
Поиск записи по названию	O(*n*)	O(*n*)	O(1)*	O(log *n*)	O(1)*	O(1)*
Вытеснение	O(*n*)	O(1)	O(*n*)	O(log *n*)	O(1)*	O(log *n*)*

* Амортизированное время.

Стоит отметить, что в куче Фибоначчи вставка и обновление приоритета будут выполняться за амортизированное постоянное время, но удаление верхнего элемента все еще требует логарифмического времени, поэтому мы не можем улучшить асимптотическое время выполнения операции сохранения записи (потому что она вызывает операцию удаления).

7.5.4. Недостатки политики LFU

Конечно, никакая политика не может быть идеальной во всех случаях, и даже у LFU есть свои минусы. В частности, если кэш работает долго, то время обработки старых записей, ставших непопулярными, может существенно вырасти.

Рассмотрим пример: представьте, что есть кэш с не более чем n записями и самая часто используемая из них, X, запрашивалась ранее m раз, но в какой-то момент к ней перестали обращаться (например, соответствующий товар закончился на складе). В таком случае для вытеснения X потребуется $n \times m$ обращений к n новым записям.

Подставим некоторые числа для большей конкретики. Пусть кэш содержит тысячу элементов и к X была тысяча обращений. В таком случае потребуется не менее одного миллиона обращений к тысяче совершенно новых записей, прежде чем X будет вытеснен после того, как станет бесполезным. Очевидно, что если к другим записям, уже находящимся в кэше, совершается больше обращений, то это число уменьшается, но в среднем оно должно быть того же порядка.

Вот некоторые возможные решения описанной проблемы увеличения временных затрат:

- ограничить максимальное значение счетчика обращений к записи;
- сбрасывать или уменьшать вдвое счетчик обращений к записи с течением времени (например, каждые несколько часов);
- вычислять взвешенную частоту на основе времени последнего обращения (это также помогло бы решить проблему неоднозначности выбора записи для вытеснения из числа записей с равными значениями счетчиков, и можно было бы избежать хранения кортежей в качестве приоритетов).

Кроме LRU и LFU, существуют и другие типы кэшей. Начав с кэша LFU и просто (на этот раз по-настоящему) изменяя политику вытеснения и выбирая другую метрику на роль приоритета записей, можно создавать кэши, адаптированные под конкретные условия.

Например, компаниям можно назначить рейтинги, делающие одни компании важнее других, или взвесить количество обращений возрастом данных, уменьшая вес вдвое каждые 30 мин с момента последнего обращения.

Как бы сложно ни казалось, на этом разнообразие не исчерпывается. Вы можете выбирать не только тип кэша, но и порядок его использования.

7.6. ПОРЯДОК ИСПОЛЬЗОВАНИЯ КЭША НЕ МЕНЕЕ ВАЖЕН

Выбор правильного типа кэша — это еще не все. Чтобы он работал эффективно, необходимо использовать его правильно.

Например, если поместить кэш перед базой данных, то в операциях записи можно записывать значения только в кэш и обновлять БД, только если другой клиент запросит ту же запись; или напротив — всегда обновлять БД.

Более того, данные можно записывать прямо в БД, а кэш обновлять, только когда данные запрашиваются для чтения или при записи.

Все эти политики имеют свои имена, потому что широко используются в разработке программного обеспечения.

Первый пример — *отложенная запись* (или *обратная запись*; *Write-Behind* или *Write-Back*). Согласно этой политике данные записываются в кэш, а в хранилище (память, БД и т. д.) они размещаются только в определенные моменты времени или при выполнении определенных условий (например, при чтении).

В таком случае кэш всегда хранит самые свежие данные, а данные в хранилище могут устаревать. Эта политика помогает поддерживать низкую задержку и способствует уменьшению нагрузки на БД, но может привести к потере данных: например, если данные переносятся в хранилище только при чтении, а запись, попавшая в кэш, больше ни разу не была прочитана. В некоторых приложениях такая потеря данных допустима, и в таких случаях политика обратной записи выглядит предпочтительнее.

Сквозная запись (или *опережающая запись*; *Write-Through* или *Write-Ahead*). При использовании этой политики данные записываются в хранилище и кэш одновременно. Соответственно, база данных будет использоваться приложением преимущественно для записи и (почти) никогда — для чтения. Эта политика медленнее политики отложенной записи, зато снижает риск потери данных, сводя его к нулю, за исключением крайних случаев и сбоев. Вообще говоря, стратегия сквозной записи не улучшает производительность процесса записи (но кэширование все равно улучшит производительность чтения), поэтому она наиболее эффективна в приложениях, интенсивно читающих данные и редко записывающих их. Хороший пример использования этой стратегии — данные сеанса.

Запись в обход (Write-Around) — это политика, согласно которой данные записываются только в хранилище, но не в кэш. Она хорошо подходит для приложений типа «записал и забыл», которые редко или никогда не читают повторно недавно записанные данные. Стоимость чтения недавно записанных данных в этом случае довольно высока, потому что влечет за собой промах кэша.

Сквозное чтение (Read-Through) — это общая стратегия записи данных в кэш только при их чтении. Она наиболее эффективна, когда данные хранятся в медленном хранилище. Для записи в такие хранилища можно использовать политику обратной записи (Write-Back) или записи в обход (Write-Around). Особенность политики сквозного

чтения в том, что приложение взаимодействует с кэшем только для чтения, а к *кэшу хранилища* обращается только при промахе кэша с политикой сквозного чтения.

Стратегия *опережающего обновления (Refresh-Ahead)* используется в кэшах, где элементы могут устареть, считаться просроченными по истечении определенного времени[1]. При таком подходе часто используемые элементы с истекающим сроком действия будут заблаговременно обновляться из основного хранилища, а само приложение не почувствует задержек, вызванных медленным чтением БД.

Политика *кэш на стороне (Cache-Aside)* предполагает, что кэш находится на стороне приложения и используется только приложением. Она отличается от политики сквозного чтения (Read-Through) тем, что в этом случае ответственность за управление кэшем или БД возлагается на приложение, которое сначала должно проверить кэш и в случае промаха проделать дополнительную работу, обратившись к БД и сохранив прочитанное значение в кэше.

Выбор стратегии может быть не менее важным, чем выбор реализации кэша. И в том и в другом случае неправильный выбор может привести к перегрузке и сбою базы данных (или любого другого хранилища).

7.7. ВВЕДЕНИЕ В СИНХРОНИЗАЦИЮ

До сих пор все время предполагалось, что кэши используются в однопоточном окружении.

Как вы думаете, что произойдет, если использовать код из листинга 7.6 в многопоточной среде? Те, кто знаком с законами параллельного выполнения, легко представят, какие проблемы могут возникнуть в состоянии гонки.

Взглянем еще раз на этот код, скопированный для удобства в листинг 7.8 с некоторыми минимальными упрощениями.

Листинг 7.8. Упрощенная версия метода set кэша LFU

```
function LFUCache::set(key, value)
  if (this.hashTable.contains(key)) then              #1
    node ← this.hashTable.get(key)                    #2
    node.setValue(value)                              #3
    return node.updatePriority(node.getCounter())     #4
  elsif getSize() >= this.maxSize then                #5
    evictOneEntry()                                   #6
  newNode ← this.elements.add(key, value, 1)          #7
  hashTable.set(key, newNode)                         #8
  return true
```

[1] Кэши этого типа особенно полезны в системах с согласованностью в конечном счете. Согласованность этого вида ослабляет ограничение согласованности, допуская, например, недолгую рассинхронизацию данных в кэше и в последней версии той же записи в базе данных. Она может с успехом применяться в приложениях, допускающих временную рассинхронизацию данных, например, рассинхронизация в течение 100 мс или даже секунды может быть несущественной для корзины с покупками.

Представим, что одновременно выполняются два вызова для добавления двух новых записей (ни одной из них еще нет в кэше) и кэш уже содержит `maxSize-1` элементов. Другими словами, следующий элемент заполнит кэш до максимальной емкости.

Эти два вызова будут выполняться параллельно. Предположим, что вызов A первым достигает строки 5 в листинге 7.8. Условие ложно, и выполнение передается в строку 7. Но до того, как вызов A выполнит строку 7, вызов B попадет в строку 5, и — угадали? — условие `if` по-прежнему ложно, поэтому выполнение вызова B тоже перейдет к строке 7 и выполнит ее и строку 8.

В результате количество элементов в кэше превысит максимально допустимое. Эта ситуация показана на рис. 7.12. Однако не все так плохо, потому что в строке 5 мы делаем все правильно и проверяем размер кэша, используя операцию сравнения «больше или равно». Если бы применялась операция «равно», то это состояние гонки привело бы к бесконечному росту кэша и, возможно, переполнению области динамической памяти системы.

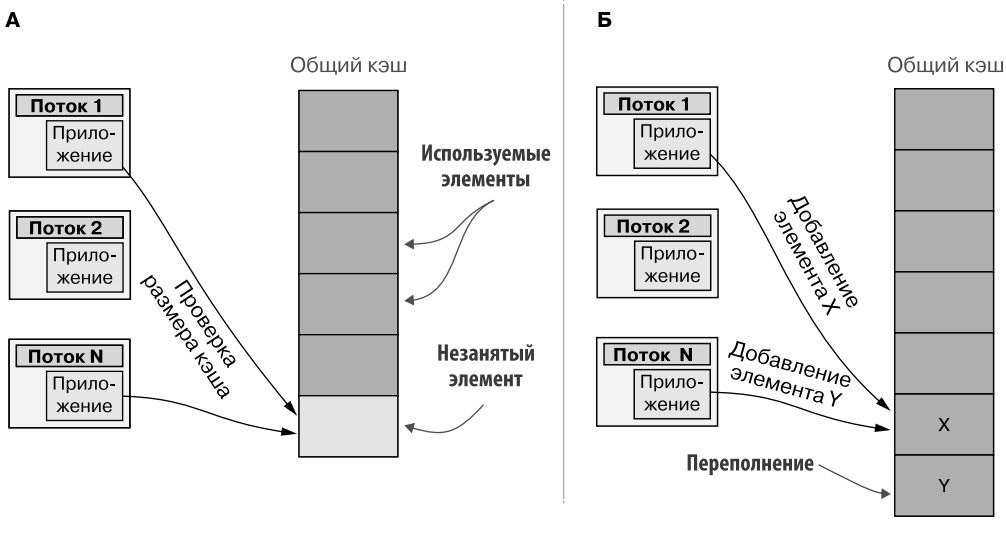


Рис. 7.12. Общий кэш, используемый в многопоточном окружении без реализации синхронизации. *А*. Два потока одновременно пытаются добавить новый элемент в кэш, где имеется только одно свободное место. Когда метод LFUCache::set внутри каждого потока одновременно проверит размер кэша (строка 5 в листинге 7.8), то в обоих случаях обнаружится, что кэш не заполнен до конца. *Б*. Без синхронизации оба потока добавят элемент в кэш (строка 7), не удаляя ничего. В зависимости от реализации кэша (и метода elements.add) это может привести к переполнению кэша или перезаписи одного из двух значений другим. Важно понимать, что даже если кэш выполняет вытеснение, как и должно быть (приложение должно заботиться только о чтении/записи элементов, и только кэш должен решать, когда необходимо вытеснить лишний элемент), то при отсутствии синхронизации внутри него может возникать состояние гонки

Код в строке 1 тоже может привести к состоянию гонки, если одновременно выполнить два вызова для добавления одного и того же ключа.

Налицо еще более тяжелая ситуация. Подумайте, как можно решить эту проблему. У кого-то может возникнуть соблазн добавить еще одну проверку прямо перед строкой 7, но насколько надежно подобное решение? Конечно, вероятность выполнения не по сценарию несколько уменьшится, но проблема не решится.

Дело здесь в том, что операция `set` не *атомарная*: пока выполняется операция добавления новой записи (или обновления старой), может произойти вызов этой же операции из другого потока для другой или точно такой же записи или даже вызов совершенно другой операции, например удаления элемента из кэша.

Пока кэш используется в однопоточном окружении и его методы вызываются синхронно, все идет гладко, потому что вызов каждого метода завершается до того, как кто-то другой попытается изменить кэш, поэтому возникает иллюзия атомарности.

В многопоточном окружении необходимо проявлять особую осторожность и явно регулировать выполнение методов, обращающихся к общим ресурсам, чтобы все методы, изменяющие состояние объекта, выполнялись в полной изоляции (чтобы никто другой не мог изменить ресурс, и, возможно, прочитать данные из него). Диаграмма на рис. 7.13 поможет вам получить представление о том, как можно исправить ситуацию, чтобы предотвратить состояние гонки.

При работе с составными структурами данных, такими как кэши, нужно быть вдвойне осторожными и использовать только структуры данных, обеспечивающие *безопасность в многопоточном окружении*[1].

В случае с кэшем LFU необходимо убедиться, что очередь с приоритетами и хеш-таблица поддерживают параллельное выполнение.

Современные языки программирования предоставляют множество механизмов синхронизации, от блокировок до семафоров и защелок.

Подробное освещение этой темы потребовало бы отдельной книги, поэтому, к сожалению, нам придется ограничиться простым примером решения этой проблемы: полный код потокобезопасной реализации LRU/LFU на Java вы найдете в репозитории на GitHub[2].

---

[1] Иначе говоря, они должны надежно работать в многопоточном окружении, не приводя к состоянию гонки.

[2] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#cache.

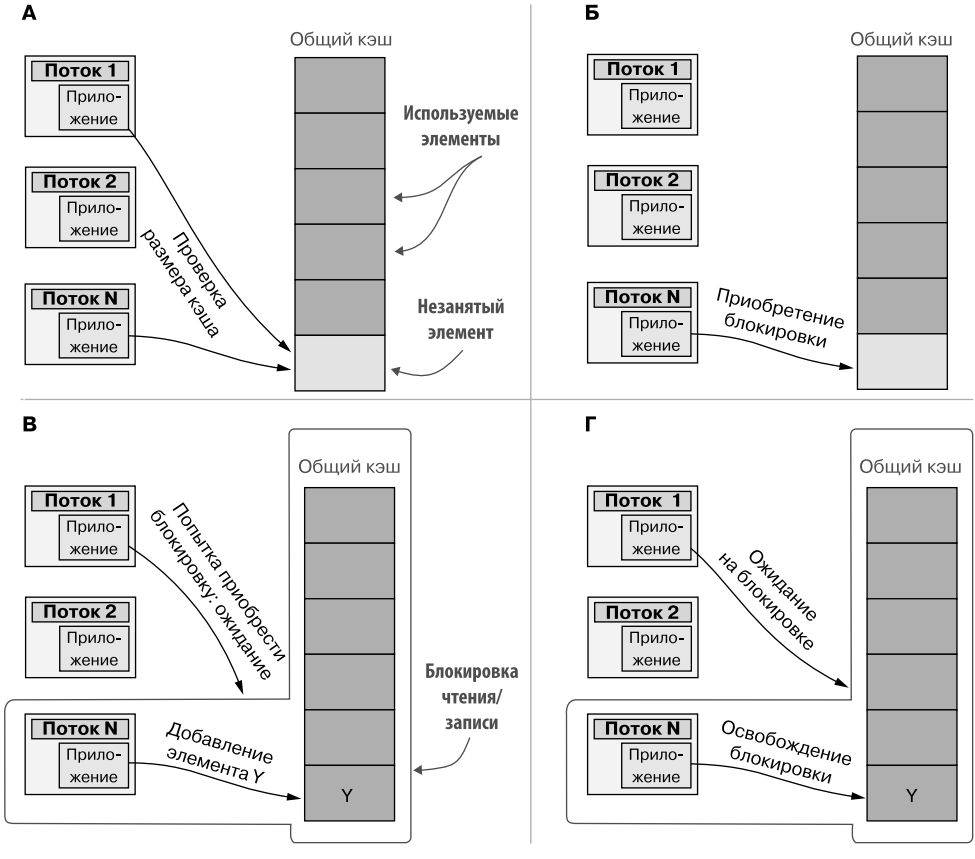


Рис. 7.13. Синхронизированная реализация кэша помогает избежать состояния гонки. В примере на рис. 7.12 синхронная структура данных требует, чтобы любой поток блокировал доступ к ней *(Б)* перед записью значения *(В)*. Если другой поток попытается приобрести блокировку, пока она используется потоком N, то система приостановит его до момента снятия блокировки *(Г)*. После этого (не показано на рисунке) ожидающий поток сможет получить блокировку и, возможно, будет состязаться за ее приобретение с другими потоками, тоже пытающимися ее получить

7.7.1. Решение проблемы параллельного выполнения (на Java)

Если сделать еще одно исключение в этой главе и показать некоторый код на конкретном языке программирования вместо псевдокода, то сложится впечатление, что эти концепции можно эффективно реализовать только в контексте конкретного языка.

Однако многие языки программирования поддерживают одни и те же концепции, и представленную логику несложно перенести на любой язык.

Начать следует с конструктора класса, показанного в листинге 7.9, потому что там появятся новые элементы.

Листинг 7.9. Конструктор кэша LFU, реализация на Java

```
LFUCache(int maxSize) {
  this.maxSize = maxSize;                                                    #1
  this.hashTable = new ConcurrentHashMap<Key, PriorityQueueNode >(maxSize); #2
  this.elements = new ConcurrentHeap<Pair<Key, Value>, Integer>();           #3
  ReentrantReadWriteLock lock = new ReentrantReadWriteLock();                #4
  this.readLock = lock.readLock();                                           #5
  this.writeLock = lock.writeLock();
}
```

Здесь можно видеть несколько заметных изменений по сравнению с листингом 7.5, кроме, конечно, самого перехода на синтаксис Java.

Обратите внимание, что в строках 2 и 3 вместо простого класса `HashMap` используется `ConcurrentHashMap`, что вполне ожидаемо, и то же касается класса `ConcurrentHeap`[1]. Внутренние структуры данных тоже должны иметь поддержку синхронизации, чтобы гарантировать атомарность операций, выполняемых параллельно. Технически если правильно организовать синхронизацию методов класса `LRUCache`, то параллельный доступ к его внутренним полям вообще будет невозможен, поэтому с успехом можно использовать обычные однопоточные структуры данных. При использовании же их параллельных версий следует быть очень осмотрительными, чтобы избежать ошибок и взаимоблокировок[2].

7.7.2. Блокировки

Однако в строке 4 нас поджидает кое-что новенькое, совершенно новый атрибут нашего класса: объект типа `ReentrantReadWriteLock`. Прежде чем объяснять, что такое реентерабельная блокировка, отмечу: это механизм, который мы будем использовать для организации параллельного доступа к методам `LFUCache`. Java предоставляет несколько способов для его реализации, от объявления методов как синхронизированных до использования семафоров, защелок и других подобных механизмов. Какой из них лучше, зависит от контекста, и иногда с успехом можно использовать любой из нескольких методов. В данном случае я предпочитаю реентерабельную блокировку для *чтения/записи* вместо объявления методов синхронизированными, потому что это дает больше гибкости в принятии решений, когда использовать блокировку для чтения, а когда для записи. Как будет показано далее, неправильный выбор может сильно повлиять на производительность.

---

[1] Стандартная библиотека Java не предоставляет реализацию PriorityQueue, поэтому вы не найдете этот класс в JRE. Код в нашем репозитории использует версию простой кучи с поддержкой синхронизации.

[2] Ситуаций, когда все потоки ожидают на блокировках, попытавшись обратиться к общим ресурсам, и ни один из них не может двинуться дальше. В результате взаимоблокировки приложение перестает отвечать на запросы, застревает и в конечном счете завершается с ошибкой.

Но обо всем по порядку: давайте определимся, что такое блокировка, и поговорим о реентерабельности.

Блокировка — это механизм параллелизма, название которого говорит само за себя. Вы, наверное, уже слышали о блокировках в базах данных. Здесь блокировки действуют по тому же принципу. Если объект (экземпляр класса) имеет некоторый код, выполняющийся под защитой блокировки, это означает, что фрагмент кода этого экземпляра, выполняющийся под защитой блокировки, всегда будет работать в одиночку, сколько бы одновременных вызовов ни было сделано, а все остальные фрагменты будут ждать своей очереди.

В нашем примере это означает, что из двух вызовов `set` второй будет простаивать в ожидании, пока выполняется первый.

Как видите, это мощный механизм, но он тоже таит в себе опасности. Если не снять блокировку, то все остальные вызовы остановятся в ожидании навсегда *(взаимоблокировка)*, а чрезмерное использование блокировок может заметно снизить производительность, вызывая ненужные задержки.

Блокировки, по сути, преобразуют параллельное выполнение в синхронное (отсюда и термин *«синхронизация»*), используя очереди ожидания для управления доступом к общему ресурсу.

Подробное описание особенностей работы блокировок, причин появления взаимоблокировок и способов их предотвращения обычно занимает пару глав в книгах по операционным системам, поэтому углубленное обсуждение этого материала выходит за рамки нашей задачи. Я настоятельно рекомендую продолжить знакомство с этой темой[1], потому что она становится все более и более важной в современном программировании веб-приложений, а также приложений, выполняющих интенсивные параллельные вычисления на графических процессорах.

Теперь разберемся, что значит «реентерабельный». Как следует из самого слова[2], используя реентерабельную блокировку, поток может войти в критический фрагмент кода (или заблокировать ресурс) более одного раза, и так каждый раз. Когда поток приобретает блокировку в очередной раз, ее счетчик увеличивается на единицу, а когда освобождает — уменьшается на единицу. Когда счетчик обнуляется, блокировка освобождается фактически. Почему важно использовать реентерабельную

---

[1] Для знакомых с языком C++ или желающих узнать о нем больше отличным введением может стать книга C++ Concurrency in Action Энтони Уильямса (Anthony Williams), Manning Publications, 2019 (*Уильямс Э.* Параллельное программирование на C++ в действии. Практика разработки многопоточных программ). Для приверженцев функционального программирования я рекомендую книгу Concurrency in .NET Риккардо Террелла (Riccardo Terrell), Manning Publications, 2018 (*Террелл Р.* Конкурентность и параллелизм на платформе .NET. Паттерны эффективного проектирования. — СПб.: Питер, 2019).

[2] Реентерабельным (от англ. reentrant — «повторно входимым») называется модуль программы, который может вызываться рекурсивно или одновременно несколькими процессами. — *Примеч. пер.*

блокировку? Потому что без нее, если один и тот же поток попытается заблокировать один и тот же ресурс более одного раза, можно попасть в ситуацию взаимоблокировки.

Почему это так важно для нас? Потерпите немного, несколькими абзацами ниже я покажу вам это на практическом примере.

7.7.3. Получение блокировки

Итак, теперь, получив представление о том, что такое блокировка, можно взглянуть на модифицированную версию метода `set` (листинг 7.10).

Первым своим действием метод блокирует ресурс (весь кэш) для записи. Что это значит? Когда ресурс блокируется для записи, то только поток, удерживающий блокировку, сможет писать и читать данные из этого ресурса, а все остальные потоки должны будут ждать, пока текущий поток освободит блокировку.

Вторая инструкция в методе — `try` с блоком `finally` в конце метода (строки, отмеченные цифрами 4 и 5), где производится освобождение блокировки. Это необходимо для предотвращения взаимоблокировок: если какая-либо из операций внутри `try` потерпит неудачу, метод все равно освободит блокировку перед выходом, чтобы другие потоки могли попытаться получить ее без риска оказаться заблокированными навсегда.

Листинг 7.10. Параллельная версия метода set кэша LFU на Java

```
public boolean set(Key key, Value value) {
  writeLock.lock();                                                    #1
  try {                                                                #2
    if (this.hashTable.contains(key)) {
      PriorityQueueNode node = this.hashTable.get(key);
      node.setValue(value);
      return node.updatePriority(node.getCounter());
    } else if (this.getSize() >= this.maxSize) {
      this.evictOneEntry();                                            #3
    }
    PriorityQueueNode newNode = this.elements.add(key, value, 1);
    this.hashTable.put(key, newNode);
    return true;
  } finally {                                                          #4
    writeLock.unlock();                                                #5
  }
}
```

В этих примерах мы по-прежнему используем хеш-таблицу для хранения ссылок на узлы очереди с приоритетами, подобно тому как в реализации LRU хранили ссылки на узлы связанного списка. Для LRU в этом есть определенный смысл — такой подход обеспечивает постоянное время выполнения всех операций, как описано в подразделе 7.4.2. Однако в реализации LFU достичь постоянного времени выполнения невозможно, поэтому в хеш-таблице можно хранить сами значения. Это упростит реализацию, как можно видеть в коде из нашего репозитория, хотя и за счет некоторого изменения времени выполнения операций.

Приостановитесь на минутку и поразмышляйте над этим, прежде чем читать объяснение. Причина, как вы уже поняли, проста: в обоих методах, `set` и `get`, вызываем `updatePriority` для обновления приоритетов в очереди. При наличии ссылки на обновляемый узел методы `pushdown` и `bubbleUp` потребуют логарифмического времени, как рассказывалось в главе 2, в подразделах 2.6.1 и 2.6.2.

Однако поиск элемента в куче обычно требует линейного времени. В подразделе 2.6.8 было показано альтернативное решение, основанное на использовании вспомогательной хеш-таблицы внутри очереди с приоритетами. Так становится возможным выполнять поиск за амортизированное постоянное время, благодаря чему метод `updatePriority` будет выполняться за амортизированное логарифмическое время.

Выше отмечалось, что при работе с блокировками следует проявлять особую осторожность, поэтому возникает вопрос: можно ли перенести получение блокировки ниже в функции?

Это зависит от конкретной ситуации. Нам нужны гарантии, что в процессе поиска ключа в хеш-таблице никакой другой поток не сможет выполнить запись в нее. Если бы в роли хеш-таблицы использовалась структура данных с поддержкой параллельного выполнения, то мы могли бы переместить получение блокировки для записи в первый `if`. Но тогда также пришлось бы изменить остальную часть функции (потому что вытеснение старого элемента и добавление нового должны происходить атомарно) и проявить особую осторожность при работе с хеш-таблицей, потому что появляется второй блокируемый ресурс, а это может привести к состоянию взаимоблокировки[1].

Проще говоря, не стоит усложнять (к тому же это усложнение все равно не даст большого преимущества).

7.7.4. Реентерабельные блокировки

Теперь наконец можно объяснить, как реентерабельная блокировка предотвращает состояние взаимоблокировки в нашем случае. В строке, отмеченной номером 3 в листинге 7.10, текущий поток вызывает `evictOneEntry` для удаления одного элемента из кэша. Мы решили оставить этот метод приватным по уважительной причине, но представим, что понадобилось объявить его общедоступным, чтобы позволить клиентам освобождать место в кэше (например, чтобы избежать проблем с нехваткой памяти или динамически сжимать кэш в процессе сборки мусора, потому что крайне нежелательно перезапускать его и терять все его содержимое).

В таком случае также придется предусмотреть синхронизацию в `evictOneEntry` и внутри него устанавливать блокировку для записи. При использовании нереентерабельной блокировки метод `evictOneEntry` попытается получить блокировку для записи, но `set` уже установил ее и не сможет освободить, пока не завершится `evictOneEntry`.

---

[1] Если поток A блокирует хеш-таблицу, а поток B получает блокировку для записи на весь кэш, а затем каждый вынужден ждать, пока другой освободит свою блокировку, получится эквивалент мексиканского противостояния для потоков. В фильмах это всегда заканчивается плохо.

Проще говоря, поток зависнет в ожидании, когда освободится ресурс, чего никогда не произойдет, и все остальные потоки тоже зависнут очень скоро. По сути, это путь к тупику, и попасть в такую ситуацию очень просто. Вот почему необходимо быть осторожными с механизмами синхронизации.

При использовании реентерабельной блокировки, поскольку `evictOneEntry` и `set` принадлежат одному потоку, `evictOneEntry` сможет в строке 1 получить блокировку, удерживаемую методом `set`, и продолжить выполнение. В этом случае взаимоблокировка не наступает!

7.7.5. Блокировки для чтения

Пока ни слова не сказано о блокировках для чтения. Выше упоминалось, что они важны для производительности, но почему? Вернемся к методу `get` в листинге 7.11.

Перед чтением данных из кэша он получает блокировку для чтения и освобождает ее в самом конце перед выходом из функции.

Очень похоже на то, что делается в методе `set`, верно? Единственное отличие в том, что на этот раз используется блокировка для чтения, так что разница должна быть, и она действительно есть.

Листинг 7.11. Метод get кэша LFU на Java

```
public Value get(Key key) {
  readLock.lock();                                    #1
  try {
    PriorityQueueNode node = this.hashTable.get(key);
    if (node == null) {
      return null;
    } else {
      node.updatePriority(node.getCounter() + 1);     #2
      return node.getValue();
    }
  } finally {
    readLock.unlock();                                #3
  }
}
```

В каждый конкретный момент времени только один поток может удерживать блокировку для записи, но блокировка для чтения может удерживаться несколькими потоками при условии, что ни один поток не удерживает блокировку для записи.

Выполняя чтение из структуры данных (или из базы данных), мы не модифицируем ее, поэтому, если 1, 2, 10 или миллион потоков одновременно будут читать содержимое из одного и того же ресурса, они всегда будут получать согласованные результаты (если инвариант реализован правильно и нет побочных эффектов, влияющих на содержимое ресурса).

Выполняя запись в ресурс, мы его изменяем, и пока эта операция не завершена, все другие операции чтения и записи не будут иметь смысла, потому что они могут

основываться на противоречивых данных. Это обстоятельство наводит на мысль о четырех возможных комбинациях блокировок (чтение-чтение, чтение-запись, запись-чтение и запись-запись), как показано в табл. 7.4.

Таблица 7.4. Комбинации удерживаемых и запрашиваемых блокировок

Запрашиваемая/удерживаемая блокировка	Чтение	Запись
Чтение	Допускается	Необходимо ждать
Запись	Необходимо ждать	Необходимо ждать

Если какой-то (другой) поток удерживает блокировку для чтения, ничто не препятствует получить другую блокировку для чтения, но при попытке получить блокировку для записи нужно дождаться, пока все блокировки для чтения будут освобождены.

Если какой-то поток удерживает блокировку для записи, все остальные потоки должны ждать, чтобы получить блокировку для записи или чтения.

То есть преимущество различий между блокировками для чтения и для записи с точки зрения кэша заключается в том, что все вызовы `get` могут выполняться параллельно, а блокироваться будут только вызовы `set`. Если не учитывать этот аспект, то доступ к кэшу будет полностью синхронным и только один поток в каждый момент времени сможет выполнять поиск в кэше. Это приведет к напрасным задержкам, уменьшит преимущество использования кэша и, возможно, даже сделает его контрпродуктивным.

7.7.6. Другие подходы к решению проблемы параллелизма

Блокировки не единственный способ борьбы с проблемами параллелизма. Блокировка защищает доступ к сегменту кода, который изменяет содержимое ресурса, поэтому можно быть уверенными, что эти инструкции будут выполняться одним и только одним потоком в каждый момент времени.

На другом конце спектра находится кардинально противоположное решение, широко используемое в функциональном программировании: полностью исключить возможность изменения ресурсов.

Фактически один из принципов функционального программирования требует, чтобы все объекты были неизменяемыми. Это дает ряд преимуществ; например, упрощает анализ кода: мы можем проанализировать функцию, убедиться, что никакие другие методы не повлияют на ее работу, и полностью устранить состояние гонки. Но этот подход имеет также некоторые недостатки. Например, усложнение некоторых операций, таких как сохранение текущего состояния, и удорожание обновления состояния, включающего большие объекты.

Когда речь идет о параллелизме, то наличие неизменяемых объектов означает возможность читать ресурс, не беспокоясь о том, что кто-то другой изменит его в этот момент, поэтому отпадает необходимость в блокировках.

Писать такой код немного сложнее[1]. В функциональном мире метод, подобный `set`, вернул бы не логическое значение, а новый экземпляр самого кэша с обновленным содержимым.

7.8. ПРИМЕРЫ ПРИМЕНЕНИЯ КЭША

Этот раздел определенно был бы короче, если бы в нем перечислялись примеры систем, не использующих кэш! Кэш можно встретить повсюду — от процессоров до приложений самого высокого уровня, которые только можно себе представить.

Однако низкоуровневые аппаратные кэши работают немного иначе, поэтому сосредоточимся на программных кэшах.

Как уже упоминалось, многие одно- и многопоточные приложения используют специальные кэши в памяти, чтобы избежать повторения дорогостоящих вычислений.

Эти кэши обычно реализованы в виде библиотечных объектов (в ОО-языках[2]) и выделяют динамическую память в куче того же приложения, которое их использует. В многопоточных приложениях кэш может работать в собственном потоке и совместно использоваться несколькими потоками (как было показано в предыдущем разделе, в этом случае требуется проявлять особую осторожность для предотвращения состояния гонки).

Следующий шаг — выделение кэша из приложения в отдельный процесс и взаимодействие с ним по определенному протоколу; это может быть HTTP, Thrift или даже RPC.

Типичным примером может служить организация нескольких уровней кэша в веб-приложении:

- CDN (Content Delivery Network — сеть доставки содержимого) для кэширования и доставки статического (и некоторого динамического) контента;
- веб-кэш, кэширующий контент веб-сервера (обычно целые страницы и, может быть, самой CDN);
- кэш приложений, хранящий результат дорогостоящих вычислений на сервере;
- кэш БД, хранящий наиболее часто используемые записи, таблицы или страницы из базы данных.

Наконец, в больших приложениях, получающих тысячи запросов в секунду (и даже больше), кэши необходимо масштабировать вместе с остальной частью веб-стека.

---

[1] Состояние гонки все еще возможно, когда два потока пытаются одновременно обновить общий кэш, но эту проблему можно решить с помощью оптимистичных блокировок и версионирования ресурса. Для более детального знакомства с оптимистичными блокировками взгляните на алгоритм «сравнение с заменой» (Compare-And-Swap, CAS). Используя неизменяемость и оптимистичные блокировки, можно получить ощутимый прирост производительности во многих распространенных контекстах/языках.

[2] Объектно-ориентированные языки, такие как Java, где основными строительными блоками являются классы и объекты.

Это означает, что одного процесса может быть недостаточно для хранения всех записей, которые приложение должно кэшировать, обеспечивая бесперебойную работу. Например, если у нас есть база данных, получающая миллион обращений в секунду, то кэш перед ней с емкостью один миллион элементов, вероятно, будет практически бесполезен, поэтому процент промахов будет слишком велик и почти все запросы будут выполняться в БД, приводя через несколько минут к ее остановке.

По этой причине были созданы распределенные кэши. Эти кэши, например Cassandra, используют несколько процессов, называемых узлами, каждый из которых является автономным кэшем, и еще один процесс (иногда специальный узел), называемый оркестратором, который отвечает за передачу запросов нужному узлу. Распределенные кэши используют специальную функцию хеширования, *согласованное хеширование* (consistent hashing)[1], чтобы решить, какой узел должен хранить запись. Такое хеширование показано на рис. 7.14.

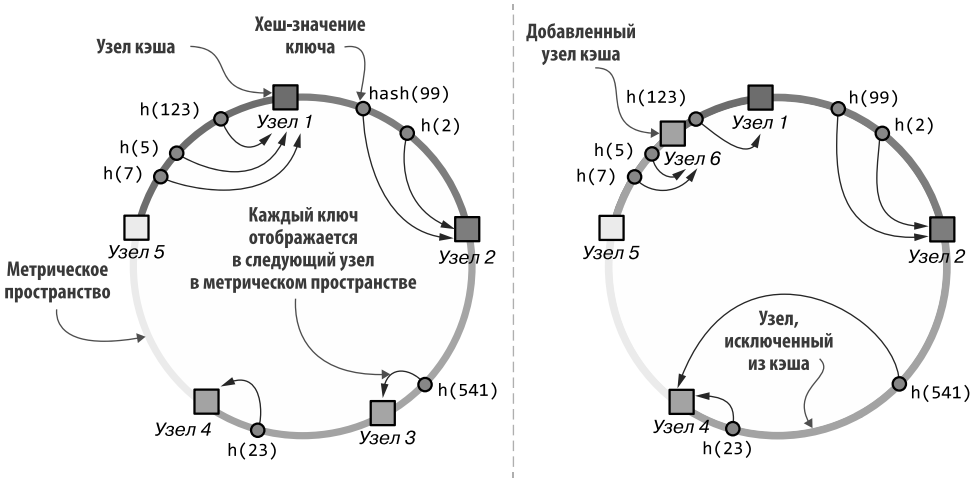


Рис. 7.14. Массив узлов кэша, использующих согласованное хеширование для распределения ключей. *Слева* — исходный массив узлов. Узлы и ключи отображаются в одно и то же метрическое пространство (пространство идентификаторов), обычно представляемое в виде круга (это пространство можно получить даже с помощью простой операции деления хеш-значений по модулю). Ключи назначаются следующему узлу в круге, то есть узлу, значение которого равно наименьшему идентификатору узла, большему, чем хеш ключа. *Справа* показано, как при удалении или добавлении узла кэша требуется повторно отобразить только часть ключей

[1] Согласованное хеширование — особый вид хеширования, который гарантирует, что при удалении одного из узлов необходимо будет отобразить только ключи, размещенные на этом узле. Порядок работы этого механизма показан на рис. 7.14. Дело не только в кэше: применительно к хеш-таблице согласованное хеширование гарантирует, что при изменении размера таблицы с n слотами потребуется повторно отобразить только один из n хранимых ключей. Это особенно важно для распределенных кэшей, где иначе могло бы потребоваться повторно отобразить миллиарды элементов в случае сбоя.

Веб-клиенты тоже широко используют кэширование: браузеры имеют свои кэши (хотя они работают немного иначе, храня контент на локальном диске), но они также имеют кэш DNS для хранения IP-адресов, соответствующих доменам, которые вы просматриваете. В последнее время появились кэши для локального хранилища и хранилища сеансов, но, пожалуй, наиболее многообещающим из всех является кэширование с использованием механизма Service Worker.

РЕЗЮМЕ

- Кэши используются повсеместно, на всех уровнях вычислительного стека, от аппаратных кэшей в процессоре до распределенных веб-кэшей в Интернете.
- Выполняете ли вы сложнейшие вычисления на графическом процессоре или подключаетесь к внешней службе, разумно предположить, что результат, который вы только что получили, может быть снова использован в ближайшем будущем, поэтому просто сохраните его в кэше.
- Кэши приносят выгоды на всех уровнях. Вероятно, их можно было бы игнорировать в автономных приложениях, работающих локально на одном компьютере, но для распределенных приложений они имеют первостепенное значение, позволяя обрабатывать миллиарды запросов в день.
- Существуют разные типы кэшей, и основное различие между ними — политика вытеснения. Кэш обычно имеет фиксированный максимальный размер, по достижении которого политика вытеснения определяет, какие записи должны быть удалены, чтобы освободить место для более новых.
- Существуют также различные способы подключения кэша к системе. Свои особенности работы демонстрируют упреждающие записи, упреждающее чтение, сквозная запись, отложенная запись и упреждающее обновление.
- Совместно используя ресурс (в данном случае кэш) в нескольких потоках, необходимо проявлять осторожность при его проектировании, чтобы не столкнуться с состоянием гонки, несогласованностью данных или, что еще хуже, взаимоблокировками.
- Функциональное программирование позволяет упростить решение проблем параллелизма.

Часть II

Многомерные запросы

Общей темой всех глав центральной части книги является *поиск ближайшего соседа*. Сначала он будет представлен как еще один частный случай поиска, а затем мы используем его как строительный блок для более сложных алгоритмов.

Эта часть начинается с описания сложностей, характерных для обработки многомерных данных: их индексации и выполнения пространственных запросов. Я еще раз покажу, как специализированные структуры данных могут обеспечить значительные улучшения по сравнению с базовыми алгоритмами поиска.

Далее будут показаны две дополнительные структуры данных, которые можно использовать для поиска в многомерных данных.

Во второй половине этой части нам предстоит рассмотреть применение поиска ближайшего соседа. Начнем с нескольких практических примеров, а затем сосредоточимся на кластеризации, широко использующей пространственные запросы. Разговор о кластеризации позволит представить распределенные вычисления, в частности модель программирования *MapReduce*, которую можно использовать для обработки объемов данных, слишком больших для одной машины.

Структура части II имеет важное отличие от структуры первых семи глав. Как вы увидите сами, обсуждаемые темы чрезвычайно широкие и ни их, ни хотя бы наиболее важные их части невозможно охватить в одной главе. Поэтому, в отличие от глав в части I, следовавших разным схемам объяснения, здесь нам придется придерживаться единой схемы, согласно которой каждая глава будет охватывать только одну часть общего обсуждения.

В главе 8 вы познакомитесь с проблемой поиска ближайшего соседа. Обсудим несколько упрощенных подходов к многомерным запросам и определим задачу, которая будет использоваться в качестве примера на протяжении почти всей части II.

В главе 9 описываются *k-мерные деревья*, применяемые для эффективного поиска в многомерных наборах данных, причем основной упор делается на двумерный случай (исключительно ради простоты визуализации).

Материал главы 10 представляет более совершенные версии этих деревьев, *R-деревья*, рассматриваемые очень кратко, и *SS-деревья*, при обсуждении которых, напротив, мы углубимся в реализацию каждого метода. В заключительных разделах этой главы мы с вами обсудим производительность SS-деревьев и способы ее дальнейшего увеличения, а затем сравним их с *k*-мерными деревьями.

В главе 11 основное внимание уделяется применению поиска ближайшего соседа и подробно описывается пример его практического использования (поиск ближайшего склада, с которого товары должны отправляться покупателям), а также упоминаются несколько проблем, которые могут быть решены применением *k*-мерных или SS-деревьев.

В главе 12 представлен еще один пример использования эффективных алгоритмов поиска ближайшего соседа. Здесь будет дано введение в сферу машинного обучения и описаны три алгоритма кластеризации: метод *k-средних*, *DBSCAN* и *OPTICS*.

Глава 13 завершает эту часть введением в *MapReduce*, мощную модель распределенных вычислений, и описанием ее применения к трем алгоритмам кластеризации: методу *k-средних*, *DBSCAN* (они обсуждаются в главе 12) и алгоритму *кластеризации купола* (canopy clustering), представленному в этой главе.

8 Поиск ближайших соседей

В этой главе

- ✓ Поиск ближайших точек в многомерном наборе данных.
- ✓ Структуры данных для индексации многомерного пространства.
- ✓ Проблемы, характерные для индексации многомерных пространств.
- ✓ Эффективный алгоритм поиска ближайшего соседа.

До сих пор в этой книге мы работали с контейнерами, содержавшими одномерные данные: записи, которые хранились в очередях, деревьях и хеш-таблицах, всегда представлялись как числа (или некоторое их подобие) — элементарные значения, которые можно сравнивать друг с другом в математическом смысле.

В этой главе предстоит увидеть, что такое упрощение не всегда применимо в реальной жизни, и рассмотреть проблемы, возникающие при обработке сложных многомерных данных. Но не отчаивайтесь, потому что в следующих главах вы познакомитесь со структурами данных, способными помочь в обработке упомянутых многомерных данных, и увидите примеры практического их применения для реализации эффективного поиска ближайшего соседа, например в процессе кластеризации.

Увидите сами, обсуждаемые темы слишком широкие, чтобы их или хотя бы наиболее важные их части можно было охватить в одной главе. Поэтому, в отличие от глав части I, материал которых подавался по разным схемам объяснения, здесь, как я уже сказал, придется придерживаться единой схемы, согласно которой каждая глава будет охватывать только одну часть общего обсуждения:

- глава 8 знакомит с рассматриваемой задачей и примером ее применения в реальной жизни;
- главы 9–10 описывают три структуры данных, позволяющие реализовать эффективный поиск ближайшего соседа;
- глава 11 демонстрирует применение этих структур для решения задачи, описанной здесь же, и показывает дополнительные примеры их применения;
- главы 12–13 представляют конкретный пример применения поиска ближайшего соседа: кластеризацию.

8.1. ЗАДАЧА ПОИСКА БЛИЖАЙШЕГО СОСЕДА

Начнем наше путешествие с рис. 8.1, на котором изображена карта (из альтернативной вселенной) с несколькими городами и расположенными в их районе складами.

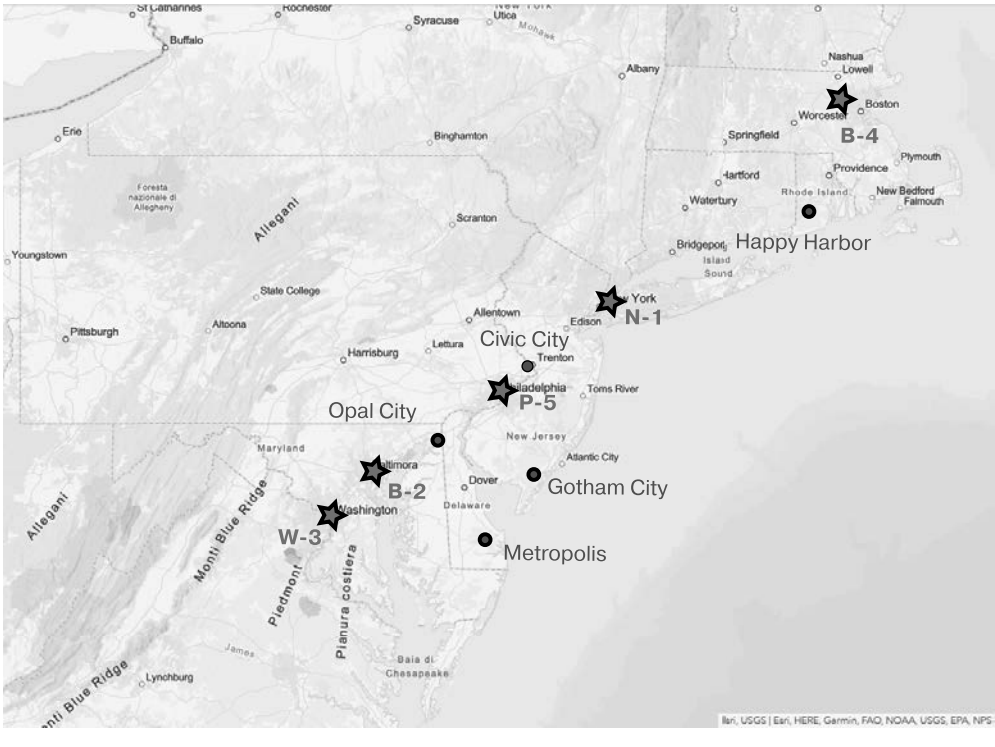


Рис. 8.1. Вымышленная карта складов (отмечены звездочками) рядом с крупными городами на побережье (в альтернативной вселенной)

Представьте, что вы живете в 1990-х, на заре эры Интернета, когда электронная коммерция еще только делает свои первые шаги.

Вы создали интернет-магазин, через который предлагаете на рынок товары местного производства, сотрудничая с несколькими розничными торговцами. Они продают товары в реальных магазинах, а вы предоставляете им инфраструктуру для реализации тех же товаров через Интернет с небольшой скидкой.

Каждый склад берет на себя хлопоты по доставке заказанных товаров, но, чтобы набрать обороты и привлечь больше розничных продавцов, вы делаете им специальное предложение: за каждую доставку на расстояние более 10 км вы будете сокращать свою комиссию пропорционально расстоянию.

Вернемся к рис. 8.1. Вы — главный архитектор этой компании, и ваша главная цель — отыскать склад, ближайший к клиенту, где имеется заказанный товар, и по возможности не далее чем в 10 км от клиента.

Проще говоря, чтобы ваша компания не прогорела, заказы всегда должны пересылаться на склад, ближайший к клиенту.

Представьте, что кто-то из Gotham City заказал французский сыр. Вы просматриваете список складов, вычисляете расстояния между ними и адресом клиента и выбираете ближайший, P-5. Сразу после этого кто-то из Metropolis заказывает два круга одного и того же сыра; к сожалению, вы не можете использовать ни одно из ранее вычисленных расстояний, потому что заказчик находится в совершенно другом месте. Итак, вы просто снова просматриваете список складов, вычисляете все расстояния и выбираете склад B-2. Если следующий запрос придет, скажем, из Civic City, все повторится вновь! Вам снова нужно вычислить все N расстояний до всех N складов.

8.2. РЕШЕНИЯ

Я знаю, что на рис. 8.1 показаны только пять складов, поэтому выбор подходящего склада для каждого покупателя кажется простой и быстрой операцией. Заказы можно даже обрабатывать вручную, выбирая нужный склад в каждом конкретном случае, исходя из своего внутреннего чутья и опыта.

Но представьте, что через год ваш бизнес развернулся, появилось большое количество магазинов, решивших вести торговлю через ваш сайт, и у вас их теперь около сотни. Задача усложнилась, и ваш отдел обслуживания клиентов уже не в состоянии справиться с тысячей заказов в день: выбирать вручную ближайший склад для каждого заказа больше просто невозможно.

Итак, вы пишете небольшой фрагмент кода, автоматически выполняющий описанные выше шаги для каждого заказа и проверяющий все расстояния.

Однако, спустя еще один год бизнес разросся настолько, что ваш генеральный директор решает выйти на национальный рынок, заключив сделки, в результате которых сотни или тысячи средних и крупных магазинов (по всей стране) присоединятся к вашей платформе.

Вычисление миллионов расстояний для каждого клиента начинает казаться непосильной задачей, а прежние решения — неэффективными. Кроме того, поскольку вы живете еще только в конце 1990-х, серверы не так быстры, фермы серверов — редкость,

а центры обработки данных — это то, чем только-только начинают заниматься крупные производители аппаратного обеспечения, такие как IBM. Эти центры еще не настолько дешевы, чтобы думать об их применении в электронной коммерции.

8.2.1. Первые попытки

Ваше первое решение: заранее вычислить расстояние до ближайших складов для всех клиентов и для всех товаров. Но оно обречено на провал, потому что клиенты могут и будут перемещаться, к тому же иногда они могут пожелать, чтобы заказ был доставлен в офис или на почту, а не домой. Кроме того, наличие товаров на складе будет меняться со временем, то есть не всегда можно будет выбрать ближайший склад. Вам также придется вести список складов, упорядоченных по расстоянию, для каждого клиента (или по крайней мере для каждого города). Я уже упоминал, что центров обработки данных еще не существует?

8.2.2. Иногда кэширование не является решением

Это один из тех случаев, когда кэширование не особо помогает. Как упоминалось в главе 7, таких ситуаций немного, но иногда они встречаются.

Учитывая, что мы работаем в двумерном пространстве, можно было бы попробовать другой подход, основанный на приемах работы с реальными картами: разделить карту на квадраты, используя обычную сетку. Так можно быстро найти квадрат, содержащий точку, по ее координатам (просто разделив значение каждой координаты на размеры квадрата, рис. 8.2) и затем локализовать ближайшие точки в том же или соседних квадратах. Судя по всему, это может помочь уменьшить количество точек, которые нужно сравнить. Однако есть загвоздка. Этот подход работает, когда точки распределены равномерно, что редко встречается в реальности, как показано на рис. 8.2.

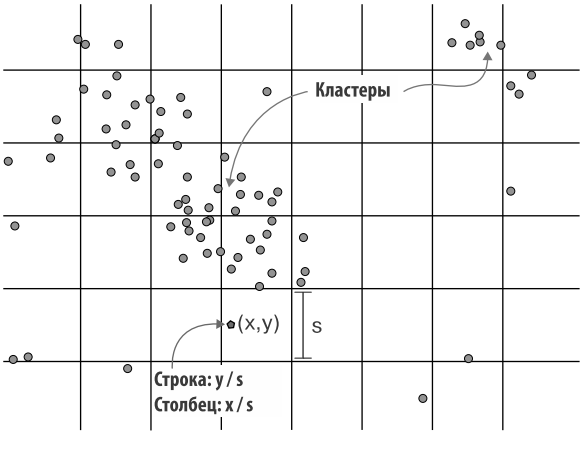


Рис. 8.2. Индексирование двумерного пространства с помощью обычных квадратов одинакового размера. Найти квадрат, в котором находится точка, несложно, но неравномерность распределения точек приводит к тому, что в среднем многие квадраты остаются пустыми, а некоторые имеют высокую плотность точек

Реальные данные образуют кластеры — плотные конгломераты точек, расположенных близко друг к другу, перемежающиеся разреженными областями с небольшим количеством точек. В равномерно распределенной сетке есть риск появления большого количества пустых квадратов и нескольких квадратов, включающих сотни и тысячи точек, что противоречит идее этого подхода. Нужно какое-то другое решение, более гибкое.

8.2.3. Упрощение, дающее подсказку

Эта задача кажется неразрешимой. В таких случаях иногда помогает формулирование упрощенной версии задачи и последующий поиск более общего решения, подходящего для исходной задачи.

Предположим, например, что поиск ограничен одномерным пространством. Скажем, нам нужно обслуживать только клиентов на одной дороге и все склады тоже расположены вдоль этой же дороги.

Чтобы упростить ситуацию еще больше, допустим, что дорога совершенно прямая, а общее преодолеваемое расстояние достаточно мало, чтобы нам не пришлось беспокоиться об искривлении земной поверхности, широте, долготе и т. д. По сути, мы предполагаем, что аппроксимация одномерным отрезком достаточно близка к рассматриваемому пространству и для представления реального расстояния между городами можно использовать евклидово расстояние в одномерном пространстве.

Схема на рис. 8.3 может помочь представить этот сценарий. У нас есть отрезок с начальной точкой, отмеченной как 0, и города и склады, которые были на рис. 8.1, но теперь все они находятся на одной линии.

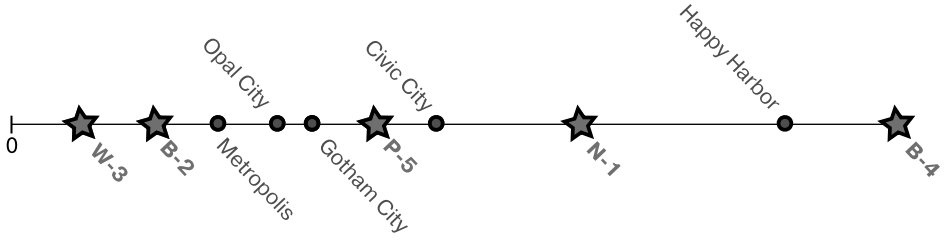


Рис. 8.3. Города и склады, теперь спроецированные в одномерное пространство — на отрезок прямой

Вот такая аппроксимация исходного сценария, в котором точки принадлежат двумерной плоскости, в свою очередь являющейся аппроксимацией реальности, где те же точки находятся на трехмерной криволинейной поверхности. В зависимости от варианта использования такой аппроксимации может оказаться вполне достаточно, но иногда может потребоваться более точная модель, учитывающая кривизну земной поверхности.

Для случайно выбранной точки на отрезке хотелось бы знать, какая из опорных точек находится ближе. В одномерном случае это напоминает поиск методом дихотомии,

верно? Посмотрите на рис. 8.4, чтобы увидеть, как можно использовать такой поиск, чтобы найти ближайшую одномерную точку.

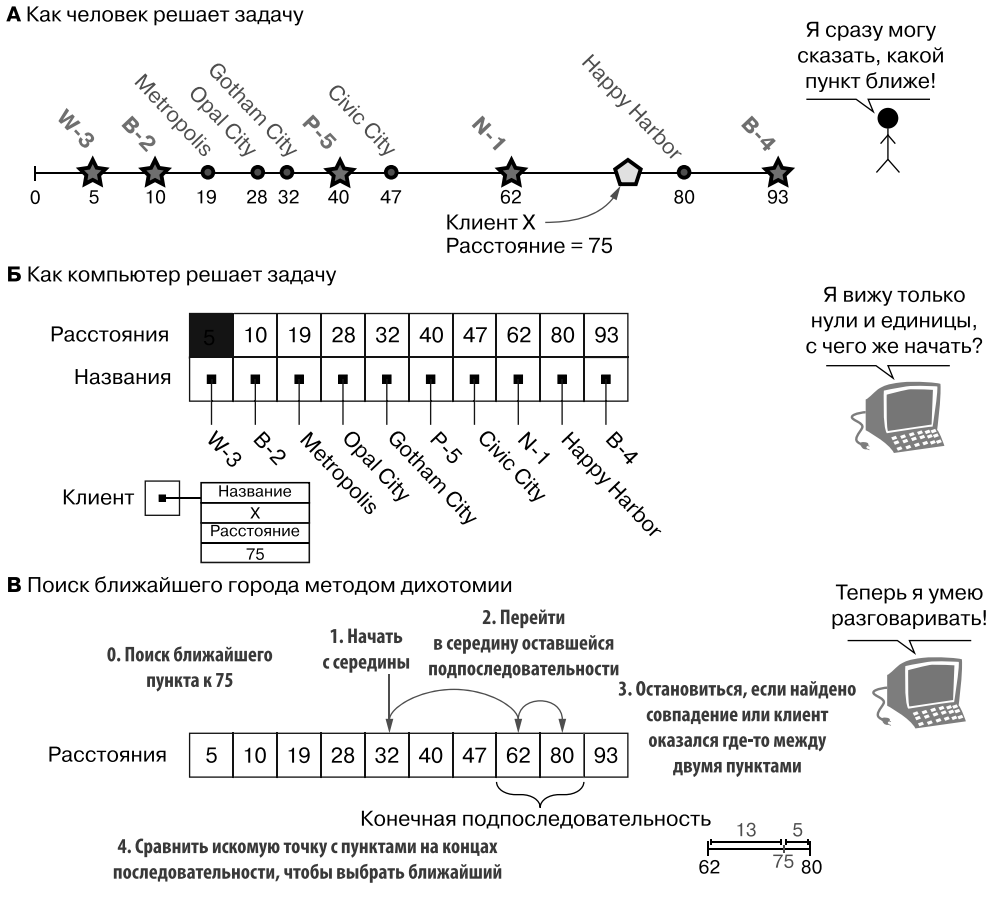


Рис. 8.4. Как люди и компьютеры решают одномерную версию задачи поиска. Каждый город, склад и клиент определяются с точки зрения расстояния до них от фиксированной (отправной) точки, расстояние до нее самой равно 0. *А*. Человек, взглянув на одномерную карту, с легкостью сможет сказать, какой пункт ближе к местоположению клиента. *Б*. Компьютеру, напротив, нужны инструкции, руководствуясь которыми он сможет найти отношение между точкой и тем, что он видит как упорядоченную последовательность чисел. Можно, конечно, просмотреть всю последовательность по порядку, но *(В)* поиск методом дихотомии подходит для этого лучше. В этом случае поиск начинается с середины последовательности: проверяется, равно ли расстояние до клиента (75) расстоянию до пункта в середине (32). Расстояния не равны, поэтому делается шаг вправо от 32, в середину подпоследовательности между 40 и 93. Так продолжается до тех пор, пока не будет найдено идеальное совпадение (оно же будет и ближайшим пунктом) или пока не получится подпоследовательность с двумя пунктами на концах. В последнем случае расстояния от нулевой точки до пунктов сравниваются с местоположением клиента и выясняется, что 80 ближе, чем 62. То есть Happy Harbor — искомый пункт

8.2.4. Тщательно выбирайте структуру данных

Поиск методом дихотомии в массиве — это круто, но массивы на самом деле не отличаются гибкостью (как отмечается в приложении B). Например, если понадобится добавить еще одну точку между W-3 и B-2, то придется переместить все точки в массиве от B-2 до B-4 и, возможно, перераспределить массив, если он статический.

К счастью, нам известна структура данных, более гибкая, чем массивы, и поддерживающая возможность эффективного поиска методом дихотомии. Как следует из названия этой структуры — двоичное дерево поиска (Binary Search Tree, BST), — это именно то, что нужно. На рис. 8.5 показано сбалансированное двоичное дерево поиска. Напомню, что гарантировать логарифмическое время выполнения типичных операций может только сбалансированное дерево.

В этом примере возьмем дерево, содержащее города и склады. Для простоты можно представить, что в каждом городе есть большой склад или распределительный центр, поэтому поиск может просто вернуть ближайший к клиенту (находящемуся вне какого-либо города) пункт (город или склад).

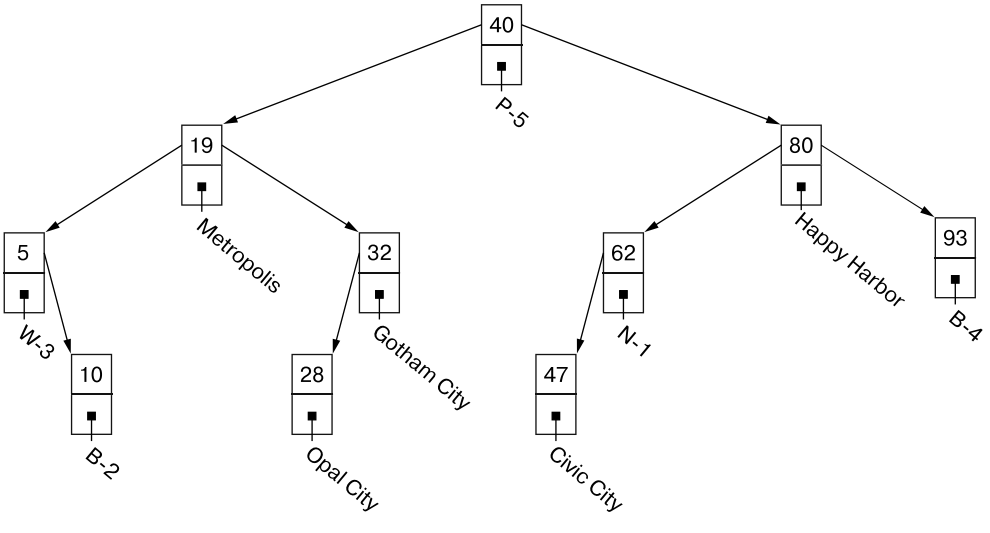


Рис. 8.5. Сбалансированное двоичное дерево поиска, содержащее города и склады. Обратите внимание, что оно не единственно возможное из сбалансированных деревьев поиска для этого набора значений. Для него существует как минимум 32 допустимых и сбалансированных двоичных дерева поиска. В качестве упражнения можете попробовать перечислить их все (подсказка: какие внутренние узлы можно повернуть[1] без изменения высоты дерева?)

[1] Здесь под поворотом понимается операция уравновешивания, выполняемая в том числе на красно-черных деревьях.

И действительно, вставка, удаление и поиск в нашем сбалансированном двоичном дереве поиска гарантированно будут выполняться за логарифмическое время, что намного лучше по сравнению с первоначальным линейным поиском. Логарифмическое время выполнения растет удивительно медленно; просто подумайте, что при наличии миллиона пунктов вы могли бы найти желаемый всего за 20 шагов!

На рис. 8.6 показано, как выполняется поиск ближайшего соседа в двоичном дереве поиска, изображенном на рис. 8.5 и содержащем расстояния от начала координат (координата *x*), для точки с координатой 75. Если есть точное совпадение, то ближайшим соседом станет результат поиска методом дихотомии. Если нет точного совпадения, что является наиболее распространенным случаем, то ближайшим соседом всегда будет либо узел, в котором поиск потерпит неудачу, либо его родитель.

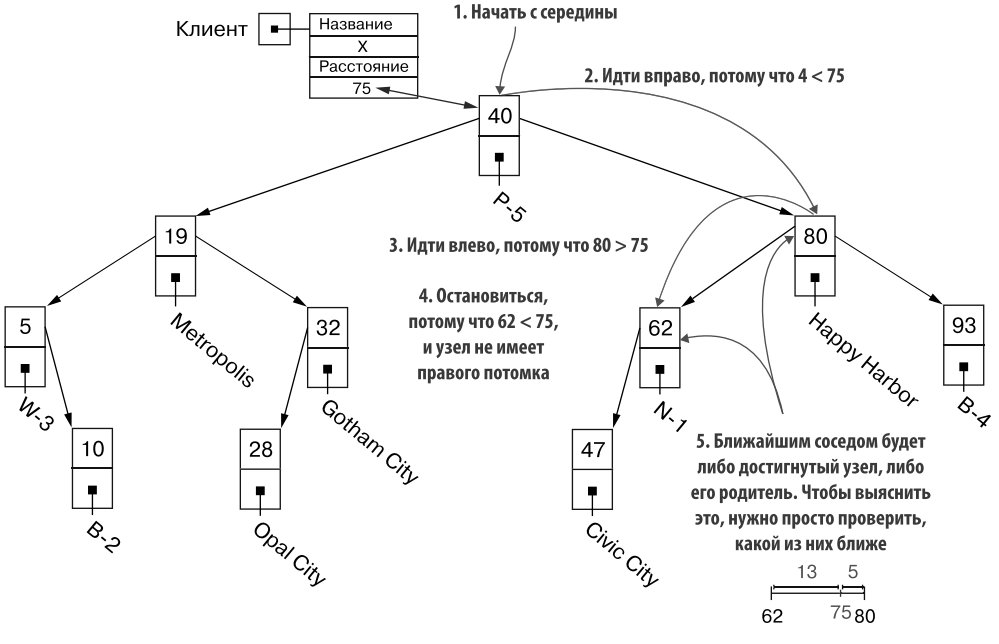


Рис. 8.6. Использование двоичного дерева поиска для определения ближайшего соседа (в одномерном пространстве) целевой точки

Как же все-таки выглядит алгоритм поиска ближайшего соседа одномерной точки, когда набор данных хранится в двоичном дереве поиска?

1. Выполнить поиск в двоичном дереве поиска.
2. Если есть точное совпадение, найденный узел явится ближайшим соседом (с расстоянием до целевой точки равным 0).
3. Иначе проверить, какой узел ближе к целевой точке — последний посещенный (на котором остановился поиск) или его родитель.

Теперь, блестяще решив задачу в одномерном пространстве, зададим себе вопрос: можно ли использовать подобную структуру данных для решения задачи в двумерном пространстве?

8.3. ОПИСАНИЕ И API

Да, можно. Но, скорее всего, сам факт постановки этого вопроса уже навел вас на определенные подозрения. И небезосновательно, потому что переход от одномерного пространства к двумерному — это большой скачок. Трудно, очень трудно представить дерево, работающее в двух измерениях. Тем не менее не волнуйтесь, мы с вами подробно исследуем двумерные деревья в следующем разделе. Совершив упомянутый скачок, затем уже легко сможем перейти к трехмерному пространству и вообще к гиперпространству с произвольным числом измерений.

Не будем ограничиваться наборами данных, лежащими в двух- или трехмерном геометрическом пространстве. Размерности могут быть любыми, если есть аппарат, позволяющий определить меру расстояния, при условии, что мера расстояния соответствует некоторым требованиям. А именно, это должно быть евклидово расстояние[1]. Например, у нас могут быть двумерные точки, первая координата которых — цена, а вторая — рейтинг, и в множестве таких точек мы сможем отыскать точку, ближайшую к целевому кортежу, например (цена = 100 $, рейтинг = 4,5). Более того, сможем отыскать *N* точек, ближайших к целевой.

В этой и следующих главах рассмотрим три структуры данных, три контейнера, позволяющие не только эффективно отыскивать ближайших соседей, но также предоставляющие несколько специальных операций:

- получение *N* ближайших к целевой точек (не обязательно в контейнере);
- получение всех точек из контейнера, находящихся не далее определенного расстояния от целевой точки (геометрически их можно интерпретировать как все точки, находящиеся внутри гиперсферы);
- получение всех точек из контейнера в пределах диапазона (все точки, лежащие внутри гиперпрямоугольника или полугиперпространства).

Теперь кратко познакомимся с этими структурами:

- *k-мерное дерево* — это особое двоичное дерево, в нем каждый нелистовой узел представляет разделяющую гиперплоскость, которая делит *k*-мерное пространство на два полупространства. Точки по одну сторону этой гиперплоскости хранятся в левом поддереве узла, а точки по другую сторону — в правом поддереве. Подробнее о *k*-мерных деревьях *k–d* рассказывается главе 9;
- *R-дерево* — буква R здесь означает rectangle (прямоугольник). R-дерево группирует близлежащие точки и определяет минимальную ограничивающую рамку

[1] Евклидово расстояние — это обычное расстояние по прямой между двумя точками в евклидовом пространстве, таком как евклидова плоскость или трехмерное евклидово пространство и их обобщения на *k* измерений.

(то есть гиперпрямоугольник), объединяющий их. Множество точек в контейнере разбито на части в виде иерархической последовательности минимальных ограничивающих рамок, по одной для каждого промежуточного узла. Причем рамка для корня объединяет все точки в дереве, а рамка каждого узла — все ограничивающие рамки его дочерних узлов. Подробнее о работе R-деревьев рассказывается в главе 10;

- *SS-дерево* — похоже на R-дерево, но в отличие от него *дерево поиска сходств* (Similarity Search Tree) использует в роли ограничивающих областей гиперсферы. Гиперсферы имеют рекурсивную структуру: листья содержат только точки, а потомки внутренних сфер — другие гиперсферы. В любом случае гиперсфера может собрать до определенного числа *n* точек (или сфер) на определенном расстоянии от центра сферы. Когда количество потомков становится больше *n* или некоторые из них находятся слишком далеко от других, дерево перебалансируется. Как это делается, я подробно расскажу в главе 10, посвященной SS-деревьям.

И наконец, определим универсальный интерфейс, общий для всех конкретных реализаций.

Абстрактная структура данных: кэш	

API	`class NearestNeighborContainer {`
	`  size(),`
	`  isEmpty(),`
	`  insert(point),`
	`  remove(point),`
	`  search(point),`
	`  nearestNeighbor(point),`
	`  pointsInRegion(targetRegion)`
	`}`

Контракт с клиентом	Контейнер позволяет вставлять и удалять точки, а также выполнять следующие запросы: • проверять существование: находится ли точка в контейнере; • определять ближайшего соседа: получать ближайшую точку (или *N* ближайших точек для любого числа *N*) к целевой. Целевая точка не обязательно должна находиться в контейнере; • определять регион: все точки в контейнере, находящиеся в пределах определенного региона — гиперсферы или гиперпрямоугольника

8.4. ПЕРЕХОД К K-МЕРНЫМ ПРОСТРАНСТВАМ

В разделе 8.2 было показано, что можно эффективно решить задачу поиска ближайших соседей в одномерном пространстве, используя двоичное дерево поиска. Если вы прочитали главы и приложения, посвященные основным структурам данных, то уже должны быть знакомы с двоичными деревьями. (Если вы пропустили приложение B, то вернитесь к нему и прочитайте раздел о двоичных деревьях!)

Однако при переходе от одномерного пространства к двумерному ситуация немного усложняется, потому что в каждом узле нет четкой развилки между двумя путями, а именно левым и правым потомками. То же самое наблюдалось в тернарных деревьях, где в каждом узле была развилка с тремя путями (или больше, в *n*-ичных деревьях) и направление дальнейшего следования зависело от результата сравнения, который может принимать не только значения `true`/`false`. Но поможет ли нам в этой ситуации *n*-ичное дерево? Проанализируем происходящее в одномерном пространстве и посмотрим, можно ли обобщить эту ситуацию.

8.4.1. Одномерный двоичный поиск

Судя по тому, что описано в разделе 8.2, легко можно выполнить двоичный поиск (поиск методом дихотомии), когда точки лежат в одномерном пространстве. На рис. 8.7 показано, как наши исходные точки можно преобразовать в точки на прямой (по сути, в точки в одномерном пространстве), чтобы каждая точка на этой прямой неявно определяла левую и правую стороны.

Итак, каждый узел имеет значение, соответствующее точке на этой линии, и каждая точка на линии определяет стороны слева и справа от нее. Но подождите — в двоичном дереве поиска каждый узел уже имеет левый и правый пути: вот почему так легко понять, что делать при поиске в двоичных деревьях!

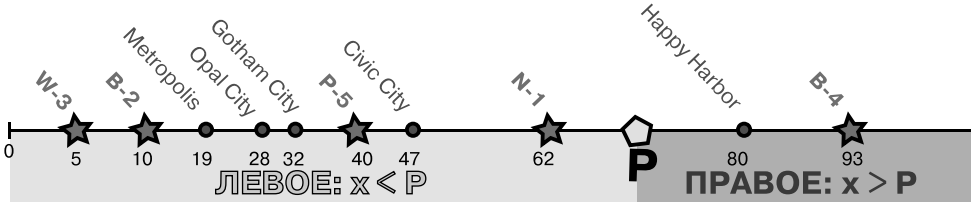


Рис. 8.7. Вещественные числа в $\mathbb{R}$. Для точки P непрерывная линия естественным образом разбивается на левое и правое подмножества

8.4.2. Переход к более высоким измерениям

Теперь мы понимаем механизм действительных чисел, но что можно сказать об $\mathbb{R}^2$? А как насчет точек в евклидовом двумерном пространстве? А как насчет $\mathbb{C}$ (множества комплексных чисел)?

ПРИМЕЧАНИЕ

Обратите внимание, что $\mathbb{R}^2$ и $\mathbb{C}$ являются двумерными евклидовыми пространствами и точки в этих пространствах можно представить парами действительных чисел.

После перехода к пространствам с бóльшим числом измерений становится не совсем понятно, как выполнить обычный поиск методом дихотомии.

Поиск методом дихотомии основан на рекурсивном делении пространства поиска пополам, но если рассматривать точку *P* на декартовой плоскости, то как разделить эту плоскость на две области, расположенные слева и справа от *P*? На рис. 8.8 показано несколько возможных вариантов.

Интуитивное решение состоит в том, чтобы разбить плоскость вдоль вертикальной линии, проходящей через *P*, чтобы в нашем представлении декартовой плоскости образовались два полупространства, фактически расположенные слева и справа от *P* (см. рис. 8.8).

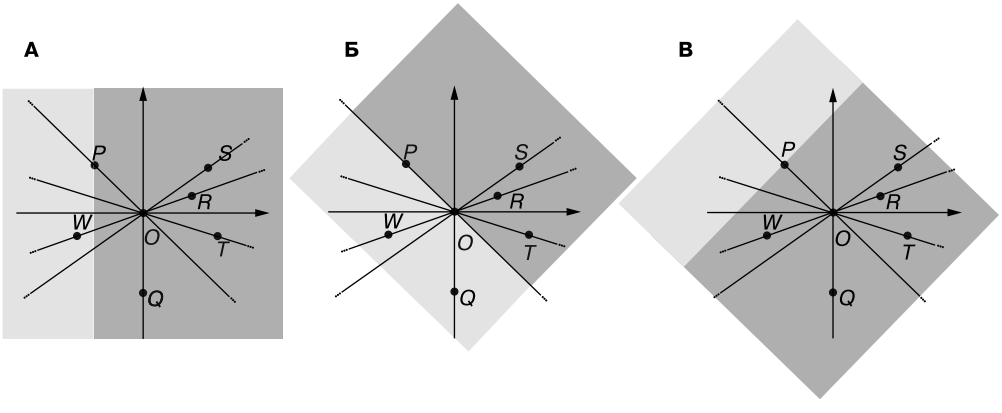


Рис. 8.8. Точки на декартовой плоскости. Обозначены линии, проходящие через начало координат и каждую точку. Показано также несколько возможных разбиений на подпространства слева и справа от P. *А*. Плоскость делится вертикальной линией, проходящей через точку P. *Б*. Линия, проведенная через точку P и начало координат, определяет два полупространства по обе стороны от нее. *В*. Используется та же прямая, что и на диаграмме B, но точки делятся на левое и правое подпространства по их проекциям на эту прямую

Это решение хорошо выглядит, если считать *P* опорной точкой, но если взять другие точки, то выявится пара недостатков:

- если взять точку *R* на рис. 8.9, провести вертикальную линию, параллельную оси *Y*, и использовать координату *x* для разделения точек, то слева окажутся точки *W*, *P*, *O*, *Q* и *U* в левом разделении, а справа — *S* и *T*. То есть, несмотря на то что *U* и *S* намного ближе друг к другу, чем *U* и *O* или *S* и *T*, они оказываются в разных разделах (в то время как две другие пары точек находятся в одном разделе);
- если взять точку *O*, то в каком разделе окажется *Q*? А как насчет любой другой точки на оси *Y*? Мы можем произвольно относить точки с одной и той же координатой *x*, совпадающей с координатой *x* опорной точки, к левому или правому разделу и должны будем сделать это для них всех, независимо от того, как далеко они находятся от *O* вдоль оси *Y*.

Оба примера показывают, какие проблемы возникают из-за одной и той же ошибки: полного игнорирования координаты *y* точек. Используя только часть доступной

информации, нельзя найти идеальное решение; всякий раз, отбрасывая некоторую информацию о наборе данных, мы упускаем возможность более эффективно организовать данные.

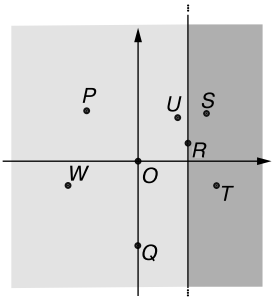


Рис. 8.9. Другой пример разделения пространства $\mathbb{R}^2$ с помощью линии, параллельной оси Y и проходящей через заданную точку (в данном случае R)

8.4.3. Моделирование двумерных разделов с помощью структуры данных

Использование одного и того же подхода с единственным направлением для всех точек не работает, поэтому плоскость лучше разделить на четыре квадранта.

И действительно, как показано на рис. 8.10, этот подход выглядит состоятельнее предыдущих попыток. Конечно, поскольку теперь у нас четыре квадранта, разбиение на левое и правое подпространства больше неприменимо.

Можно использовать дерево, каждый узел которого имеет четыре потомка вместо двух, по одному потомку на каждый возможный квадрант. При этом точки на осях или на прямых, проходящих через точку, произвольно отнести к одному из квадрантов (при условии, что это делается единообразно).

Похоже, это решение работает для $\mathbb{R}^2$, позволяя преодолеть основное ограничение, которое определилось в предыдущей попытке. Координаты `x` и `y` учитываются при группировке точек (ситуация будет еще более очевидной, если добавить больше точек на диаграмму).

Теперь возникает вопрос, можно ли распространить это решение на $\mathbb{R}^3$. Чтобы перейти в трехмерное пространство, нужно определиться: на сколько подгиперплоскостей нужно разбить плоскость для каждой точки? В одномерном пространстве мы выделяли два сегмента для каждой точки, в двумерном — четыре квадранта, соответственно, в трехмерном пространстве потребуется выделить восемь октантов[1].

---

[1] Согласно определению, октант — это один из восьми разделов евклидовой трехмерной системы координат. Обычно октантами называют восемь гиперкубов, полученных в результате разбиения $\mathbb{R}^3$ по трем декартовым осям, поэтому каждый октант определяется знаком координат. Например, (+++) — это октант, в котором все координаты положительны, (+−+) — это октант, в котором координаты x и z положительны, а координата y отрицательна.

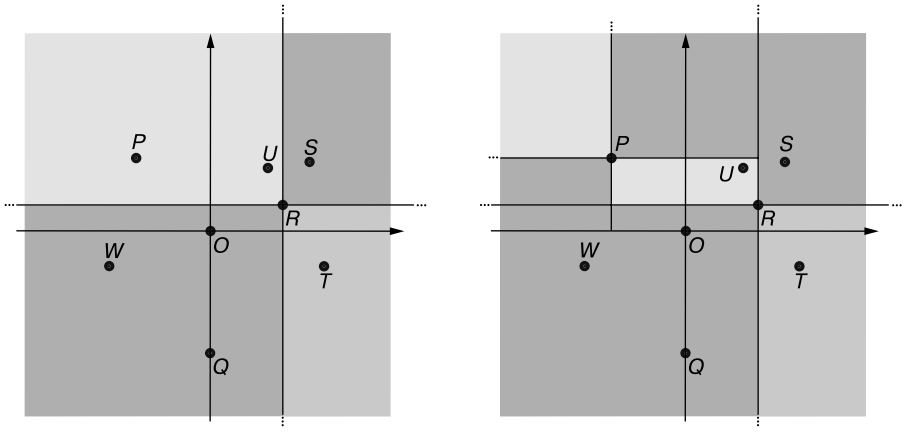


Рис. 8.10. Четырехстороннее разделение плоскости. Для каждой точки область, которой она принадлежит, разбивается на четыре квадранта с использованием линий, проходящих через точку и параллельных осям координат. Для первой из них, R в этом примере, получаем четыре бесконечных квадранта *(слева)*. Выбирая следующую точку P *(справа)*, необходимо дополнительно разделить левый верхний квадрант на четыре части, в результате чего получим конечный прямоугольник, которому принадлежит U, и три бесконечных сечения плоскости. Для каждой следующей точки, такой как U, дополнительно разделим область, которой она принадлежит, еще на четыре меньшие области

Поэтому для каждой точки в наборе данных нужно добавить в дерево узел с восемью дочерними элементами, по одному для каждого октанта, полученного в результате деления трехмерного пространства вдоль линий, параллельных декартовым осям и проходящих через точку.

В общем случае для представления k-мерного пространства потребуется дерево с узлами, содержащими 2^k потомков, потому что каждая точка будет делить гиперпространство на 2^k частей.

В реальной жизни, занимаясь обработкой больших данных, придется иметь дело с многомерными пространствами, а это означает, что k легко может иметь величину от 10 до 30 или даже 100. Наборы данных нередко содержат сотни признаков и миллионы точек. Иногда действительно необходимо иметь возможность поиска ближайших соседей в этих наборах данных, чтобы их интерпретировать (кластеризация — один из таких случаев, как будет показано в главе 12).

Даже при меньшем количестве признаков, порядка 10, каждый узел уже будет иметь около тысячи дочерних элементов. Как было показано в главе 2 при обсуждении d-ичных куч, когда коэффициент ветвления дерева возрастает слишком сильно, дерево уплощается и становится больше похожим на список.

В 100-мерных наборах данных количество дочерних элементов на узел будет близко к 10^{30}. Это число настолько велико, что делает почти невозможным хранение даже одного узла такого дерева. Нет, определенно нужно что-то другое. Но что?

Как будет показано в следующих двух главах, специалисты в области информатики нашли несколько решений этой задачи. В частности, в следующей главе вы познакомитесь с k-мерными деревьями, структурой данных, в которой используется разновидность подхода из подраздела 8.4.3, позволяющая избежать экспоненциального роста дерева.

РЕЗЮМЕ

- Для обработки многомерных данных нельзя использовать обычный контейнер.
- Когда количество измерений в наборе данных растет, индексирование традиционными методами становится невозможным из-за экспоненциального роста количества ветвей на каждом шаге.
- Общий API класса контейнеров может содержать многомерные данные и предоставлять метод, возвращающий точки, ближайшие к произвольной цели.

9 *k-мерные деревья: индексирование многомерных данных*

В этой главе

- ✓ Эффективное индексирование двумерных (и в общем случае *k*-мерных) наборов данных.
- ✓ Реализация поиска ближайших соседей с помощью *k*-мерных деревьев.
- ✓ Обсуждение сильных и слабых сторон *k*-мерных деревьев.

Эта глава структурирована несколько иначе, чем другие главы, просто потому, что она продолжает обсуждение, начатое в главе 8. Итак, задача описана: поиск в многомерных данных ближайших соседей некоторой обобщенной точки (возможно, не входящей в этот набор).

Здесь продолжим начатое обсуждение, поэтому не будем определять новую задачу, а возьмем пример поиска «ближайшего склада» из главы 8 и рассмотрим другой вариант решения с использованием *k*-мерных деревьев.

9.1. ПРОДОЛЖАЕМ С ТОГО МЕСТА, НА КОТОРОМ ОСТАНОВИЛИСЬ

Вспомним, на чем остановились в предыдущей главе. Вы разрабатываете программное обеспечение для компании электронной коммерции. Одной из функций этого ПО является поиск для любой точки на очень большой карте расположенного

поблизости склада, продающего конкретный товар (рис. 9.1). Напомню также, что нам нужно обслуживать миллионы клиентов в день по всей стране, забирая товары с тысяч складов, также разбросанных на карте.

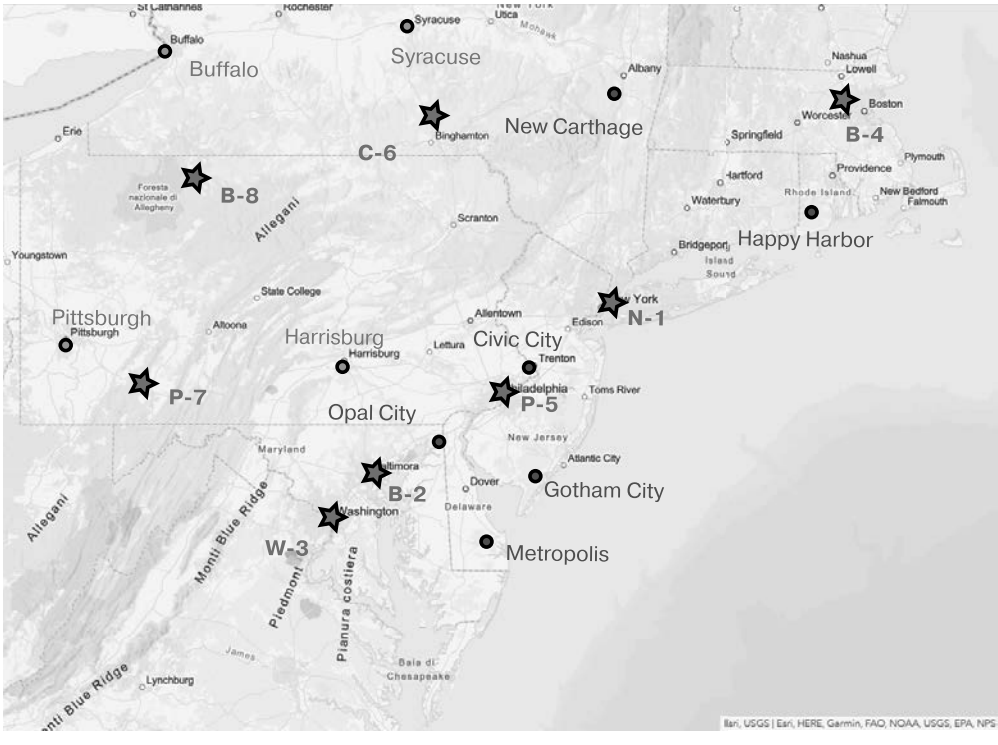


Рис. 9.1. Пример карты, на которой показаны города (настоящие и воображаемые) и склады (все воображаемые!) на восточном побережье. Хотелось бы, используя k-мерные деревья, отыскать для заданной точки на карте ближайший склад или ближайший город. Источник карты: ArcGIS

В разделе 8.2 мы с вами выяснили, что решение методом полного перебора, просматривающее весь список точек и сравнивающее каждую из них с целевой, не годится для практического применения.

Также было показано, что многомерная структура данных не позволяет использовать базовые решения, описанные в первой части книги, от куч до хеш-массивов.

Однако жизнеспособные решения существуют. В этой главе мы сначала обсудим проблемы, возникающие при переходе к многомерным пространствам; затем в ней и следующей главах рассмотрим несколько вариантов эффективных решений упомянутых проблем.

9.2. ПЕРЕХОД К K-МЕРНЫМ ПРОСТРАНСТВАМ: ЦИКЛИЧЕСКИЙ ПЕРЕБОР ИЗМЕРЕНИЙ

У вас может сложиться впечатление, что мы зашли в тупик, и неудивительно, потому что это впечатление долгое время бытовало даже в научном сообществе.

Ответ пришел в виде эвристики, созданной руками (и мозгом) Джона Луи Бентли (Jon Louis Bentley)[1].

Идея решения проста и гениальна, она проистекает из тех же соображений, которые завели нас так далеко в главе 8. Если ограничиться двумерными пространствами, то вместо деления каждой области на четыре подобласти для каждой точки можно выполнить деление пополам, чередуя деление по вертикали с делением по горизонтали.

Выполняя каждое деление, мы разделяем все множество точек в области на две группы. Затем для следующей опорной точки в каждой из двух подобластей выбирается перпендикулярное направление для следующего разделения.

На рис. 9.2 показаны несколько шагов этого алгоритма. Здесь для первой выбранной опорной точки проводится вертикальная линия, для второй опорной точки — горизонтальная полупрямая (на этот раз делится полуплоскость, а не вся плоскость!) и затем снова вертикальная (полу)линия.

Обратите внимание, что вертикальная линия, проходящая через точку $P = (P_x, P_y)$, имеет специфическую особенность: она параллельна оси Y и все точки на прямой имеют одинаковое значение координаты x, P_x. Точно так же горизонтальная линия, проходящая через P, состоит из всех точек плоскости, для которых выполняется условие $y = P_y$.

Итак, разделяя плоскость по вертикальной линии, проходящей через $P = (P_x, P_y)$, на самом деле создаем два раздела: один для точек L на плоскости, для которых $L_x < P_x$, и один для тех точек R, для которых $R_x > P_x$. То же для горизонтальных линий, только в отношении координаты y точек.

Такое двоичное разделение позволяет использовать двоичные деревья для индексации точек. Каждый узел в дереве — это опорная точка, выбранная для разделения оставшейся области пространства, а ее левое и правое поддеревья содержат все точки в двух разделах и представляют собой две подобласти, полученные в результате разделения, выполненного вдоль опорной точки (см. рис. 9.2 и 9.5).

Можно ли обобщить этот алгоритм на большее число измерений? Да, он допускает такое обобщение, потому что, хотя мы делим пространство по одной координате в каждой точке, можем циклически перебирать все координаты k-мерного пространства, и i-й уровень построенного таким образом двоичного дерева будет соответствовать разбиению по (i mod k)-му измерению.

---

[1] Multidimensional binary search trees used for associative searching. Communications of the ACM, 1975. — Vol. 18, Issue 9. — P. 509–517.

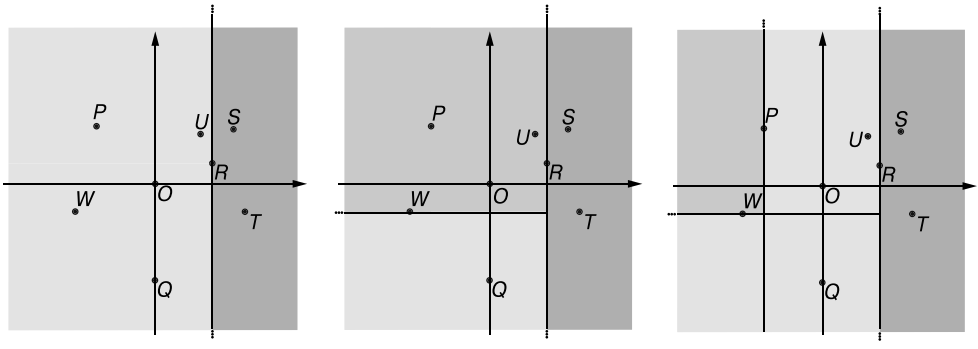


Рис. 9.2. Разделение точек в двумерном декартовом пространстве путем циклического перебора направлений разделения. Для первого разделения *(слева)* выбрана точка R, и через нее проводится вертикальная линия (параллельная оси Y, координата x постоянна). В результате получаются два полупространства, более светлое слева и более темное справа, группирующие точки W, P, O, Q, U с одной стороны и S, T — с другой. Точка R является центром этого разбиения. Затем выбирается точка W в более светлом разделе. На этот раз через нее проводится горизонтальная линия (параллельная оси X, координата y постоянна). Она делит светлый раздел на два новых раздела: один вверху слева и содержит P, O и U, второй внизу слева и содержит только Q. Для дальнейшего деления верхней левой области в точке P снова необходимо провести вертикальную линию, как показано на правой диаграмме

Это означает, что в двумерном пространстве корень разделит плоскость вдоль оси X, его потомки разделят каждую из двух полуплоскостей вдоль оси Y, а их потомки разделят свои области снова вдоль оси X и т. д. В k-мерном пространстве, где $k > 2$, разделение на уровне 0 (корень) начинается с первой координаты, на уровне 1 выполняется разделение по второй координате, на уровне 2 по третьей и т. д.

В результате плоскость будет разделена на прямоугольные области. В отличие от первоначальной идеи разделения точек только вертикальными линиями, здесь меньше областей, уходящих в бесконечность (если бы всегда использовалась одна и та же координата, все области были бы бесконечными!), и исключается возможность попадания отдаленных друг от друга точек в один раздел.

В то же время у каждого узла есть только два потомка, и можно сохранить все преимущества и гарантии двоичного разбиения и двоичного поиска.

9.2.1. Конструирование двоичного дерева поиска

До сих пор мы только намекали на конструкцию двоичного дерева поиска, подразумевая прямую трансляцию разделов, а точнее, выбранных нами опорных точек, в BST. Также было принято, что двоичное дерево поиска, которое предполагается построить, является важной частью k-мерного дерева. Дадим более формальное определение, чтобы прояснить связь между этими двумя структурами данных.

ОПРЕДЕЛЕНИЕ

k-мерным деревом называется двоичное дерево поиска, его элементами являются точки из k-мерного пространства, то есть кортежи из k элементов, которые можно сравнивать. (Для простоты предположим, что значения каждой координаты можно преобразовать в действительные числа.) Кроме того, в k-мерном дереве на каждом уровне i при выборе ветви для дальнейшего прохождения сравниваются только i-е (по модулю k) координаты точек.

Проще говоря, *k*-мерное дерево можно описать в терминах двоичного дерева поиска с необычным методом сравнения его ключей. Дополнительным преимуществом являются алгоритмы поиска, которые могут выполняться на таком дереве намного эффективнее, чем на других более простых структурах данных.

На рис. 9.3 показано, как происходит конструирование дерева для одномерного случая. Это крайний случай, потому что одноэлементные кортежи всегда приводят к использованию координаты x в точках.

Обратите внимание, что «охват» областей каждым узлом дерева носит иерархический характер: корень охватывает весь набор данных, узлы 1-го уровня — левый и правый разделы, созданные разделением по опорной точке в корне, а узлы 2-го уровня охватывают еще меньшие разделы, созданные разделением по опорным точкам в узлах уровня 1.

Под термином «охват» здесь подразумевается, что для заданного элемента X в пространстве (в одномерном пространстве — действительное число X), которое представляет дерево (как двоичное дерево поиска), все узлы, пересекаемые при поиске X (образующие путь от корня до узла N), охватывают X. В частности, узел, где останавливается поиск, — это узел, охватывающий наименьшую область, содержащую X.

Другими словами, каждый узел в дереве связан с диапазоном значений, которые приведут к нему при поиске в дереве. И этот диапазон определяет область, охватываемую узлом.

Обязательно рассмотрите пример на рис. 9.3 и постарайтесь понять каждый шаг. Можете даже попробовать выполнить приведенные в нем действия самостоятельно (например, просто изменить опорные точки или их порядок и проверить, как меняется дерево). Понимание одномерного случая необходимо для понимания конструирования двумерного дерева.

Для демонстрации этапов конструирования двумерного дерева и чтобы подчеркнуть преимущества циклического перемещения по измерениям при разбиении, на карту, изображенную на рис. 9.1, добавлены несколько городов. Так получилось более однородное двумерное пространственное распределение. С этой же целью на рис. 9.4 добавили систему координат: и начало координат, и масштаб произвольны и никак не влияют на результаты алгоритма. Всегда можно применить любую операцию переноса и масштабирования, потому что они сохраняют евклидовы расстояния.

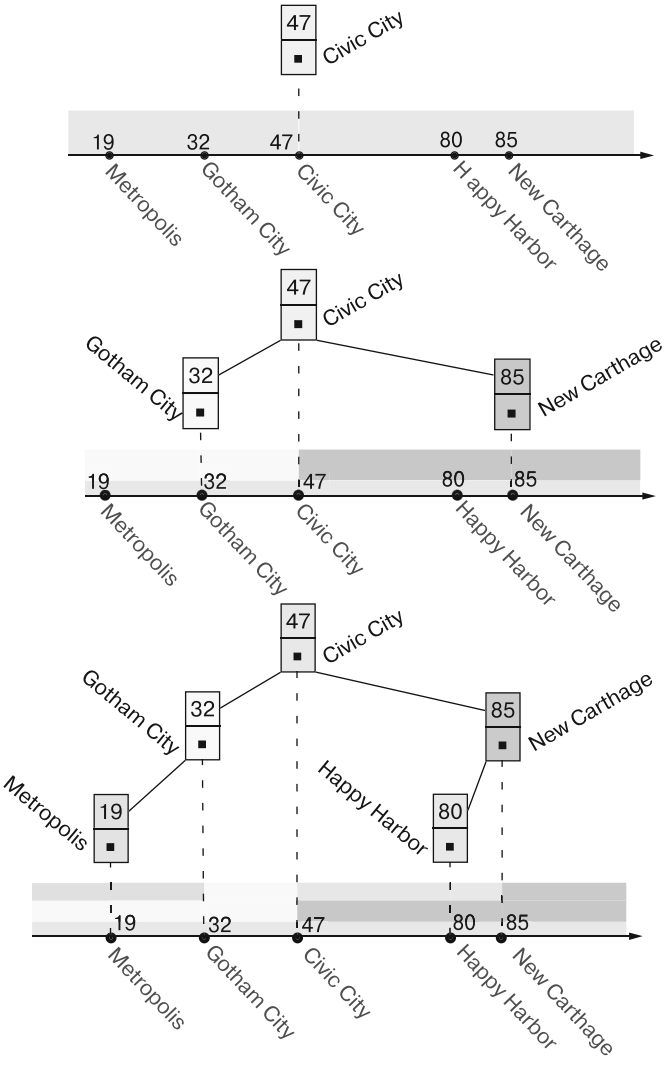


Рис. 9.3. Конструирование двоичного дерева поиска из одномерных опорных точек. *Слева:* добавляется первая опорная точка, которая будет корнем дерева. Поворот создает неявное деление по оси X, разделяя оставшиеся точки на левое и правое подмножества. Область, охватываемая узлом, есть объединение разделов, создаваемых его опорной точкой, соответственно, корень охватывает все подмножество. *По центру:* каждая из двух подобластей, образованных разделением по корню, дополнительно разбивается путем выбора опорной точки в каждой области. Как показано выделением над горизонтальной осью, каждый из узлов на уровне 1 охватывает половину пространства (в то время как корень по-прежнему охватывает все пространство). *Справа:* добавляются узлы уровня 2, дополнительно делящие пространство. Обратите внимание, что некоторые области по-прежнему охватываются только узлами уровня 1, потому что у этих промежуточных узлов есть только один дочерний узел

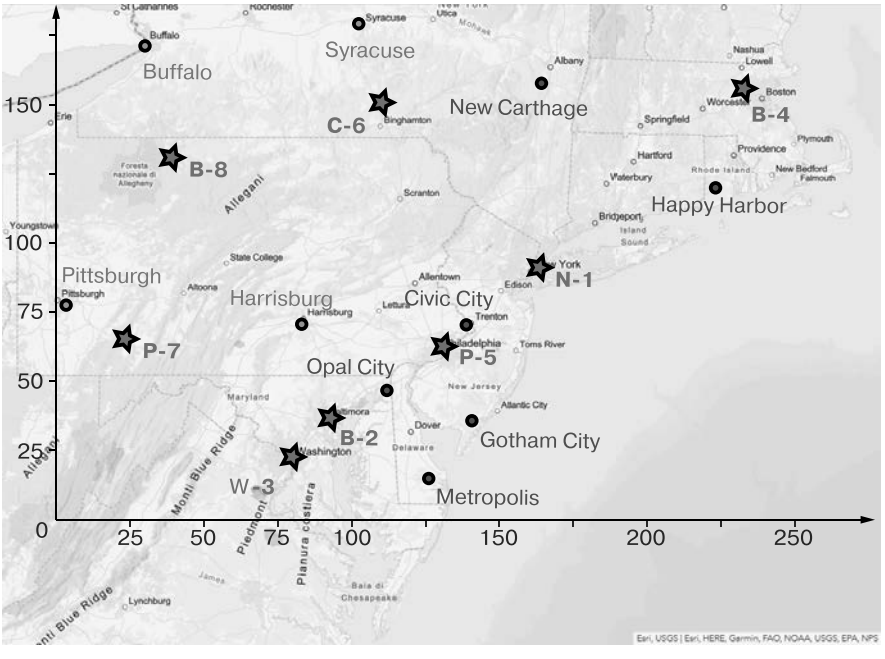


Рис. 9.4. Дополненная версия карты на рис. 9.1 с системой координат. Источник карты: ArcGIS

На рис. 9.5 показаны результаты двух первых шагов алгоритма конструирования дерева. Здесь использован другой масштаб, отличный от масштаба на рис. 9.4. Тот рисунок более реалистичный в отношении расстояний между точками, но он может вызвать некоторую путаницу; кроме того, как уже упоминалось, система координат неважна для алгоритма, главное, чтобы она была непротиворечивой.

На роль первой опорной точки выбрана Opal City[1]. Так как сначала разделение производится по координате x, все города слева от этой точки попадут в один раздел, а все города справа — в другой раздел. Далее, на втором шаге, сосредоточимся на правом разделе. Выберем, например, на роль опорной точки Civic City. На этот раз разделение производится по координате y, соответственно, все точки в верхней правой области попадают в подраздел, относящийся к правой ветви, а все точки в нижней правой области — в подраздел, соответствующий левой ветви. В данный момент у нас есть три раздела, и можно дополнительно разделить любой из них.

На рис. 9.6 показано получившееся дерево после вставки всех городов (без складов). Ребра дают одинаковое вертикальное или горизонтальное разделение на каждом уровне, а вертикальные и горизонтальные ребра чередуются в любом пути.

[1] Здесь точки выбираются произвольно, только чтобы получить более четкую картинку. В подразделе 9.3.3 будет объяснено, как реализовать выбор программно, чтобы получить сбалансированное дерево.

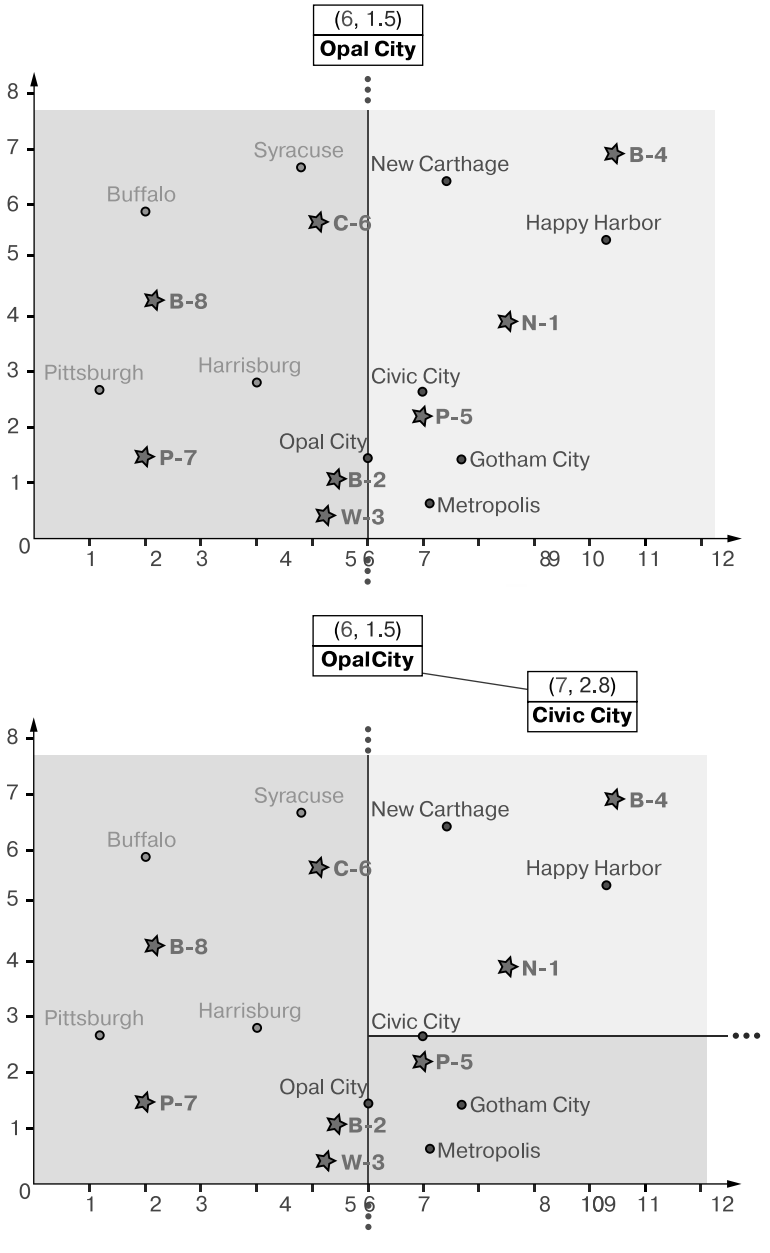


Рис. 9.5. Первые два шага конструирования k-мерного дерева для нашей карты городов в округе Колумбия. *Вверху:* сначала выполняется разделение по вертикали, по опорной точке Opal City, которая становится корнем k-мерного дерева. *Внизу:* правый раздел (созданный корнем) разделяется горизонтальной линией, проходящей через вторую опорную точку Civic City и к корню двоичного дерева поиска добавляется правый потомок. Этот узел соответствует разделению на верхнюю и нижнюю подобласти

Теперь у нас есть десять четко разделенных регионов, и если посмотреть, как распределены склады, то можно увидеть, к какому городу они могут быть близки, просто взглянув на области, где они находятся.

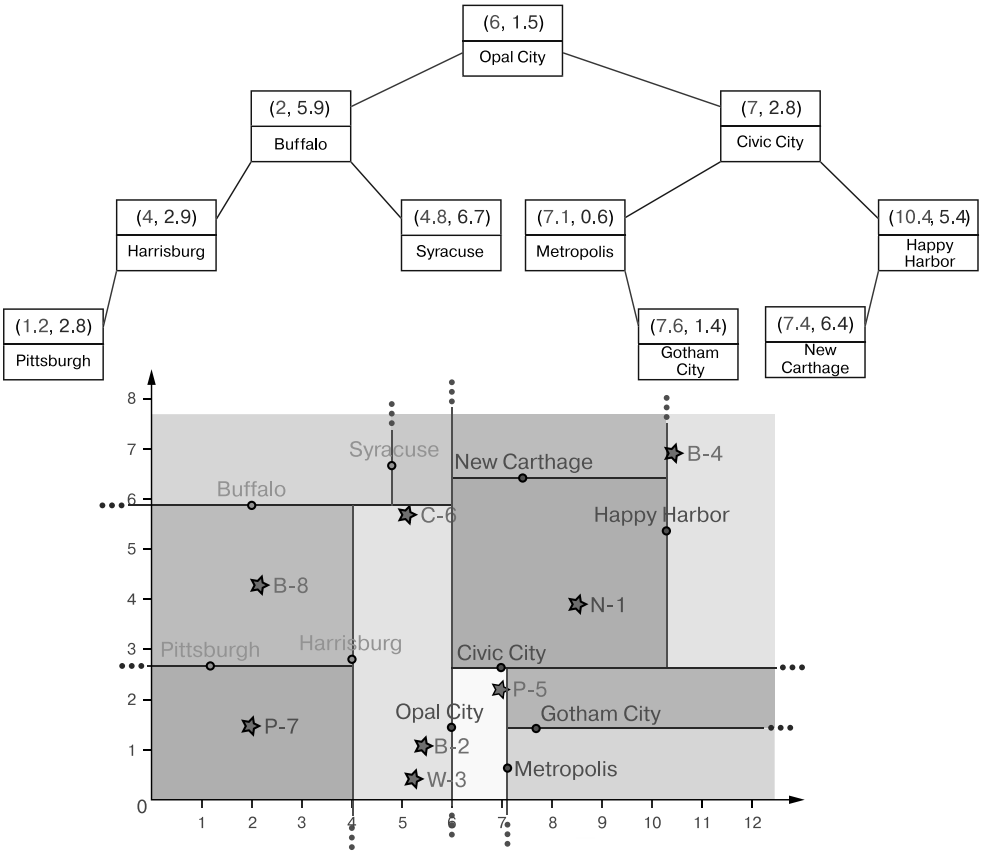


Рис. 9.6. k-мерное дерево, полученное после добавления всех городов, изображенных на рис. 9.4. (В дерево не добавлены склады не только для ясности, но и потому, что имеет смысл создать дерево только с одним типом записей, будь то города или склады, и затем искать в нем другой тип)

Однако прямого совпадения складов с городами нет, и недостаточно просмотреть области, чтобы определить ближайший склад. Например, посмотрите на склад B-8: трудно определить, какой из городов является ближайшим к этому складу — Buffalo, Pittsburgh или Harrisburg. Склад C-6 кажется ближе к Syracuse, чем к Harrisburg, несмотря на то что «отнесен» к последнему, а не к первому.

Определение ближайшей точки (точек) в таком дереве требует выполнить на несколько шагов больше, чем в обычных двоичных деревьях поиска (одномерный случай), поэтому отложим описание полного алгоритма до следующего раздела.

Как уже упоминалось, этот метод конструирования k-мерных деревьев легко обобщается на большее количество измерений, однако визуализировать деревья и сами пространства становится труднее.

Для $k = 3$ мы еще можем представить $\mathbb{R}^3$, разделенное на параллелепипеды, как показано на рис. 9.7, но для случаев с 4 и бóльшим количеством измерений обычная геометрическая интерпретация неприменима. Тем не менее, рассматривая k-мерные точки как кортежи, можно выполнить те же шаги для конструирования k-мерного дерева, которые описаны на примере двумерного дерева.

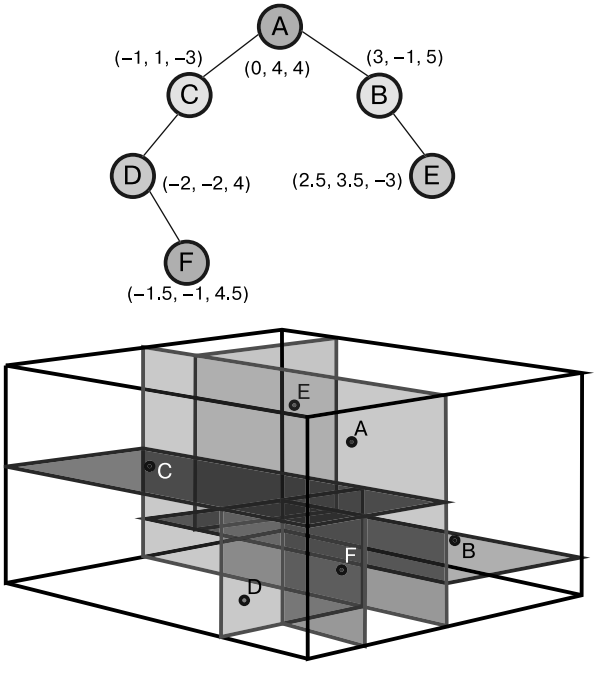


Рис. 9.7. Пример трехмерного дерева (оно же k-мерное дерево с размерностью 3). Для наглядности области не выделены, а узлы залиты тем же цветом, что и плоскости разделения, проходящие через них

9.2.2. Инварианты

Можно было бы обобщить определение k-мерных деревьев в нескольких инвариантах. k-мерное дерево определяется как двоичное дерево поиска с k-мерными точками в роли элементов, которое соблюдает следующие инварианты:

- все точки дерева имеют размерность k;
- каждый уровень имеет индекс j координаты, по которой производится разделение, такой что $0 \leq j < k$;

- если индекс координаты разделения в узле N равен j, то дочерние узлы N имеют координату разделения, равную $(j + 1) \bmod k$;
- для каждого узла N с индексом j координаты разделения все узлы L в его левом поддереве имеют меньшее значение координаты разделения, чем в узле N, $L[j] < N[j]$, а все узлы R в правом поддереве N имеют значение координаты разделения, большее или равное значению той же координаты в узле N, $R[j] \geq N[j]$.

9.2.3. Важность сбалансированности

До сих пор в этом разделе мы сознательно игнорировали одну важную деталь — особую характеристику двоичных деревьев поиска: сбалансированность. Что касается обычного двоичного дерева поиска, то вы уже знаете, что порядок вставки элементов в k-мерное дерево определяет его форму. Даже наличие конечных областей не равносильно наличию небольших областей или хорошему разделению. Если посмотреть на пример с рис. 9.6, то можно заметить, что точки довольно тщательно выбраны «вручную», чтобы в результате получилось сбалансированное дерево. Но точно так же легко получить пример ужасной последовательности вставки, создающей несбалансированное дерево (просто добавьте точки, например начиная с левого верхнего угла и двигаясь к правому нижнему углу).

Кроме того, двоичное разделение считается «хорошим», только если оно разбивает весь набор на два подмножества примерно одинакового размера.

Если удастся получить хорошее разделение на каждом шаге, то в результате получится сбалансированное дерево с логарифмической высотой. Однако можно выбрать такой порядок вставки точек, который создаст перекошенное дерево с линейной высотой.

Несколькими страницами ниже я покажу, как можно предотвратить возникновение несбалансированности, соблюдая определенные условия. В отличие от обычных двоичных деревьев поиска перебалансировка при вставке, к сожалению, не является жизнеспособным вариантом для k-мерных деревьев.

Но прежде, чем позаботиться о балансировке деревьев, подробно рассмотрим работу методов k-мерных деревьев, таких как `insert`, `remove`, и все методы обработки запросов.

9.3. МЕТОДЫ

К настоящему моменту рассмотрено несколько примеров k-мерных деревьев и приемы их конструирования. Теперь должно быть понятно, как выглядит k-мерное дерево, какие основные идеи лежат в основе этой структуры данных и как можно ее использовать.

Пришло время углубиться в основные методы k-мерных деревьев и посмотреть, как они работают и как их можно реализовать. В этом разделе будет представлен

псевдокод с реализацией этих методов, а фактическую реализацию можно найти в репозитории на GitHub[1].

На рис. 9.8 показано предварительно сконструированное *k*-мерное дерево, которое мы будем использовать в качестве отправной точки для этого раздела. Чтобы не отвлекаться на ненужные мелочи и сосредоточиться на самом алгоритме, применим упрощенное представление, показывающее точки на декартовой плоскости без осей координат. Точки обозначены заглавными буквами (пока не будем использовать названия городов), чтобы абстрагироваться от контекста и сосредоточиться на алгоритмах. Вертикальное и горизонтальное разделения тоже отображаются на диаграмме, но области не выделены, как это было сделано на рис. 9.6. Благодаря этому можно заполнить узлы дерева (вместо ребер) штриховкой, чтобы показать, что на этом уровне используется вертикальное или горизонтальное разделение.

На рис. 9.8 координаты показаны рядом и с узлами, и с точками, но иногда мы будем их опускать, чтобы они не отвлекали внимание.

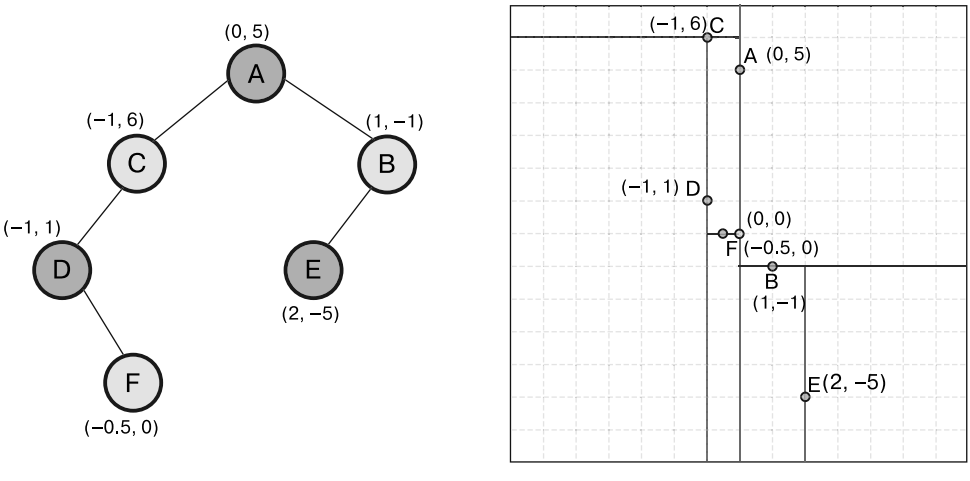


Рис. 9.8. Пример k-мерного дерева для двумерного набора данных. *Слева* изображено само дерево. *Справа* показана (часть) двумерная декартова плоскость, где начало координат находится в центре. Вертикальные линии проходят через узлы A, D и E и соответствуют разбиениям по координате x. Горизонтальные линии проведены через узлы B, C и F и соответствуют разбиениям по координате y

Начнем с «простых» методов: `search` и `insert` работают почти так же, как в обычных двоичных деревьях поиска. Сейчас последует их описание и псевдокод, но если вы уже знакомы с двоичными деревьями поиска, то можете пропустить следующие несколько подразделов.

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#k-d-tree.

Сначала определим модель k-мерного дерева и его узлов (листинг 9.1).

Листинг 9.1. Класс KdTree

```
class KdNode
  #type tuple(k)
  point
  #type KdNode
  left
  #type KdNode
  right
  #type integer
  level
  function KdNode(point, left, right, level)

class KdTree
  #type KdNode
  root
  #type integer
  k
  function KdTree(points=[])
```

`KdTree` просто содержит корневой узел и метод конструктора, принимающий необязательный массив точек. Реализацию конструирования k-мерного дерева рассмотрим позже, после знакомства с методом `insert`. А пока достаточно отметить, что он инициализирует корень пустой ссылкой на элемент (это может быть `null`, специальный экземпляр `KdNode` или что-то более подходящее для конкретного используемого языка программирования).

Для удобства предположим, что дерево хранит также значение k — размерность пространства, в котором оно определено.

Корень, как уже было сказано, является экземпляром `KdNode`. Эта структура данных моделирует узел двоичного дерева поиска и имеет два потомка, левый и правый, а его значением является точка в k-мерном пространстве. Будем использовать специальное значение `null` для представления пустого узла (и, следовательно, пустого дерева).

9.3.1. Метод search

В разделе 9.2 неявно описано, как работает поиск в k-мерных деревьях. В этом алгоритме нет ничего необычного — ординарный поиск в двоичном дереве поиска, хранящем кортежи, но с оговоркой, что вместо сравнения всего кортежа на каждом шаге сравнивается только одна координата. На уровне i сравнивается i-я координата (или $i \bmod k$, если $i \geq k$).

В листинге 9.2 показано несколько вспомогательных функций, которые помогут сохранить код чистым и ясным. В этих функциях инкапсулирована логика циклического обхода координат разделения при обходе дерева, чтобы не дублировать ее во всех методах. Благодаря им код других методов будет выглядеть проще, и если когда-нибудь понадобится изменить реализацию сравнения (например, если обнаружится

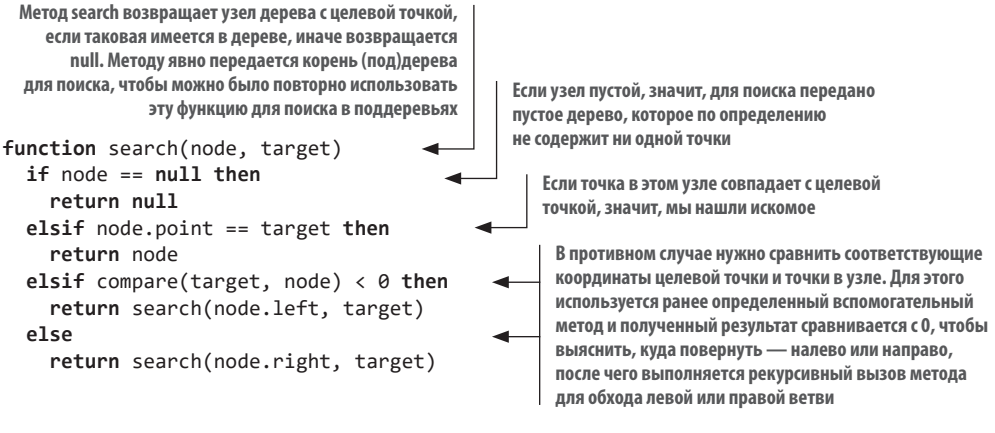
ошибка или понадобится реализовать более сложный алгоритм), достаточно будет внести изменение только в одно место.

Листинг 9.2. Вспомогательные функции


```
function getNodeKey(node)
  return getPointKey(node.point, node.level)

function getPointKey(point, level)
  j ← level % k
  return point[j]

function compare(point, node)
  return sign(getPointKey(point, node.level) - getNodeKey(node))

function splitDistance(point, node)
  return abs(getPointKey(point, node.level) - getNodeKey(node))
```

В листинге 9.3 показан псевдокод метода `search`. Во всех методах предполагаем, что имеется внутренняя версия, которая принимает `KdNode` в аргументе. Общедоступный API для `KdTree` определяет методы-обертки, вызывающие внутренние версии; например, `KdTree::search(target)` будет вызывать `search(root, target)`.

Листинг 9.3. Метод search




```
function search(node, target)
  if node == null then
    return null
  elsif node.point == target then
    return node
  elsif compare(target, node) < 0 then
    return search(node.left, target)
  else
    return search(node.right, target)
```

При использовании такого подхода получаем больше гибкости при повторном использовании методов (например, появляется возможность выполнять поиск только по поддереву, а не по всему дереву).

Обратите внимание, что к этой рекурсивной реализации применима оптимизация хвостовой рекурсии в тех языках и компиляторах, которые ее поддерживают (объяснение хвостовой рекурсии вы найдете в приложении Д).

Исследуем пример на рис. 9.9 шаг за шагом.

Поиск начинается с вызова `search(A, (-1.5, -2))`, где узел `A` является корнем *k*-мерного дерева, как показано на рис. 9.9. Поскольку `A` — не пустой узел, условие в строке 2 не выполняется, и в строке 4 производится сравнение целевой точки с `A.point` — кортежем `(0, 5)`. Они явно не совпадают, поэтому производится переход к строке 6, где вызывается вспомогательная функция `compare`, чтобы проверить, в каком направлении двигаться дальше. `A.level` имеет значение `0`, поэтому сравниваются первые элементы кортежей: `-1.5 < 0`, и это приводит к выполнению перехода в левое поддерево и вызову `search(C, (-1,5, -2))`.

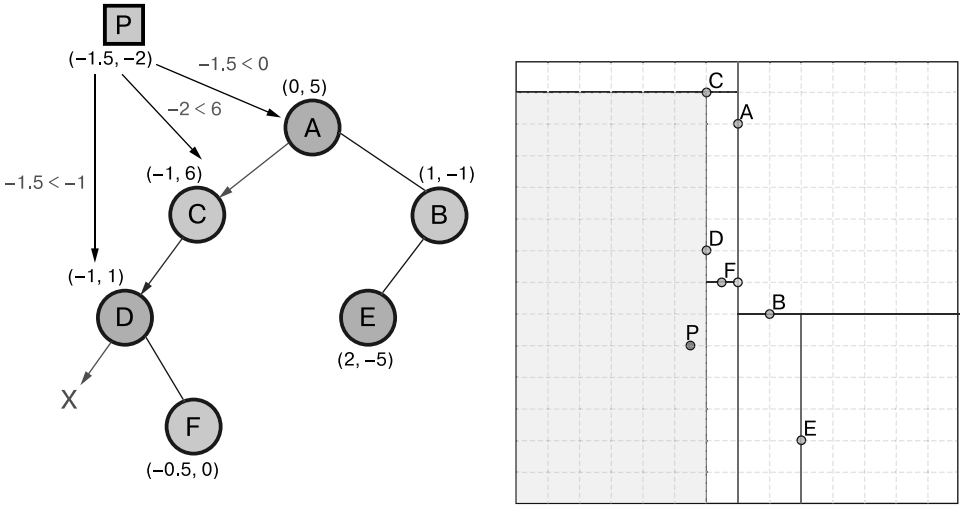


Рис. 9.9. Пример неудачного поиска в k-мерном (двумерном) дереве. Искомая точка P в идеале должна лежать в области, выделенной в левом нижнем углу, которая соответствует левому поддереву узла D

Этот вызов повторяет практически те же шаги, за исключением того, что на этот раз `C.level` имеет значение `1`, поэтому сравниваются вторые элементы кортежей. `-2 < 6`, поэтому выполняется переход в левое поддерево и вызывается `search(D, (-1.5, -2))`.

Еще раз выполняются строки 2, 4 и 6, и снова производится поворот влево; только на этот раз `D.left == null`, поэтому далее производится вызов `search(null, (-1.5, -2))`, который вернет `null` в строке 2. Выполнение возвращается назад по

стеку вызовов, и первоначальный вызов `search` также возвращает `null`, сообщая, что целевая точка не найдена.

На рис. 9.10 показан другой пример, начинающийся с вызова `search(A, (2, -5))`. В первом вызове условия в строках 2 и 4 не выполняются, как и условие в строке 6, потому что `2 > 0`. В этой ситуации в узле `A` выбирается правое поддерево и рекурсивно вызывается `search( B, (2, -5))` и затем `search(E, (2, -5))`, для которого условие в строке 4 выполняется (`E.point` соответствует целевой точке), и, как следствие, первоначальный вызов `search` возвращает узел `E`.

Насколько быстро выполняется поиск по *k*-мерному дереву? Как и в случае с обычными двоичными деревьями поиска, время выполнения пропорционально высоте дерева. Следовательно, если сохранить дерево сбалансированным, время выполнения будет равно O(log(*n*)) для дерева, содержащего *n* точек.

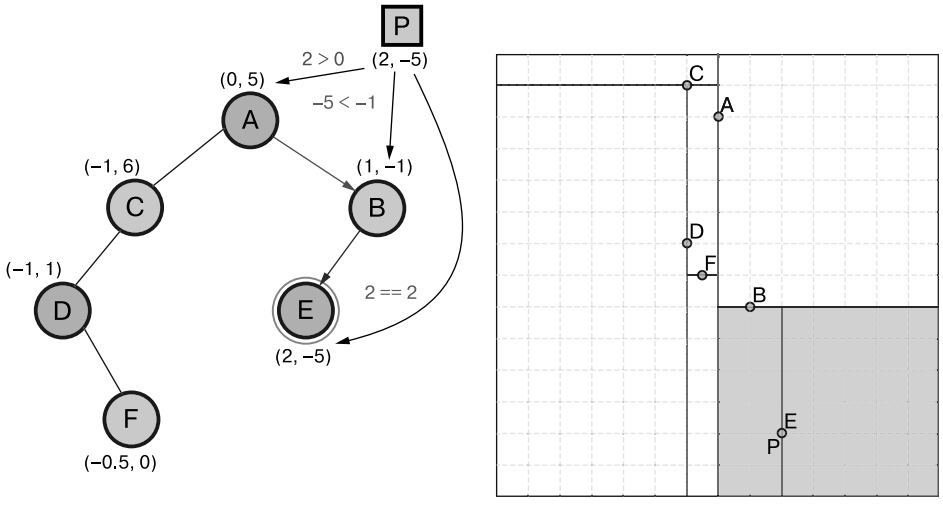


Рис. 9.10. Пример успешного поиска в k-мерном (двумерном) дереве. Здесь точки P и E совпадают, а выделенная область соответствует поддереву с корнем в узле E

9.3.2. Метод insert

Как было показано на примере двоичного дерева поиска, операцию вставки можно разбить на два этапа. Первый этап — поиск добавляемой точки. Он либо обнаружит, что точка уже имеется в дереве, либо остановится на будущем родителе — узле, к которому новая точка должна быть добавлена как дочерняя. Если точка уже находится в дереве, то дальнейшие действия зависят от принятой политики в отношении дубликатов. При условии, что дубликаты запрещены, новую точку можно проигнорировать или сгенерировать ошибку. В противном случае возможен широкий спектр решений: от использования счетчика в узлах, подсчитывающего, сколько раз точка была добавлена, до последовательного добавления дубликатов в любую

ветвь узла — например, всегда в левую, хотя, как рассказывается в приложении B, это в среднем приводит к некоторой разбалансировке деревьев.

Если точка отсутствует в дереве, поиск остановится на узле, в который новая точка должна быть добавлена в роли потомка, и после этого второй этап будет состоять в создании нового узла для точки и добавлении его в правильную ветвь родителя.

В листинге 9.4 показана реализация, не использующая метод `search`. Этот подход несколько нарушает принцип DRY[1], но позволяет упростить оба метода. Чтобы `search` можно было повторно использовать в `insert`, он должен возвращать родителя найденного узла (и, что особенно важно, последний пройденный узел, если поиск завершился неудачей). Однако такая реализация `search` малополезна и к тому же станет излишне сложной. Кроме того, предлагаемый подход позволяет написать `insert` более элегантным способом, используя шаблон, поддерживающий неизменяемость[2].

Листинг 9.4. Метод insert

Вставляет заданную точку в дерево. Возвращает указатель на корень (под)дерева, содержащего новую точку. Уровень, на который добавляется узел, по умолчанию принимается равным 0

Если узел пустой, значит, передано пустое дерево и продолжать поиск бессмысленно. Поэтому можно сразу создать новый узел и вернуть его, чтобы вызывающая сторона могла сохранить ссылку на него. Обратите внимание, что этот прием работает, даже когда дерево пустое — в этом случае создается и возвращается новый корень

```
function insert(node, newPoint, level=0)
  if node == null then
    return new KdNode(newPoint, null, null, level)
  elsif node.point == newPoint then
    return node
  elsif compare(newPoint, node) < 0 then
    node.left ← insert(node.left, newPoint, node.level + 1)
    return node
  else
    node.right ← insert(node.right, newPoint, node.level + 1)
    return node
```

Если новая точка совпадает с точкой в этом узле, значит, найден дубликат. В этом примере мы просто игнорируем дубликаты, но вы можете применить другую политику, изменив эти строки

В противном случае нужно сравнить координаты целевой точки и точки в узле. Для этого используется ранее определенный вспомогательный метод и возвращаемое им значение сравнивается с 0, чтобы выяснить, куда повернуть — налево или направо, после чего производится рекурсивный вызов метода для вставки целевой точки в выбранное поддерево

На рис. 9.11 и 9.12 показаны два примера вставки новых точек в *k*-мерное дерево, изображенное на рис. 9.10.

---

[1] Don't repeat yourself («не повторяйся»). В этом случае имеет место некоторое дублирование кода, что делает код менее удобным для сопровождения.

[2] Неизменяемость структур данных является ключевым аспектом функционального программирования. Он дает несколько важных преимуществ — от присущей безопасности в многопоточной среде до простоты отладки. Хотя этот код не реализует неизменяемую структуру данных, его легко изменить так, чтобы он соответствовал упомянутому шаблону.

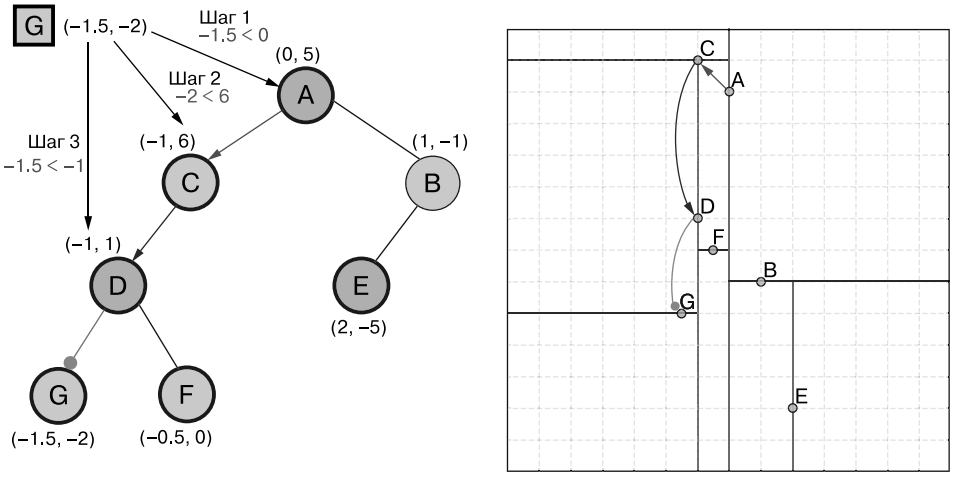


Рис. 9.11. Вставка новой точки в k-мерное (двумерное) дерево. В k-мерное дерево на рис. 9.10 добавляется новая точка G. Операция вставки, как и для двоичного дерева поиска, сначала выполняет поиск (завершающийся неудачей), а затем определяет узел, который будет родителем добавляемого узла. Если новая точка уже существует в дереве, то в зависимости от выбранной политики разрешения конфликтов можно просто проигнорировать дубликат или обработать его должным образом

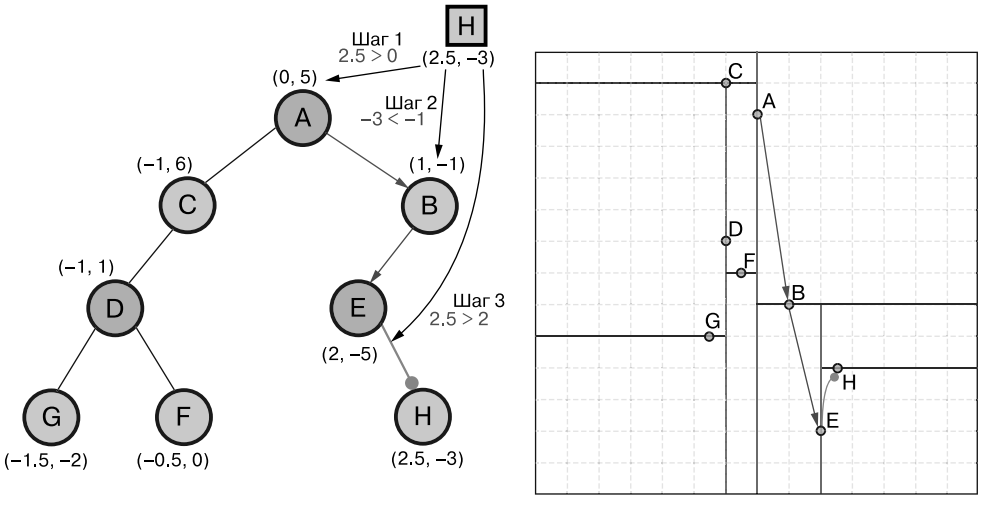


Рис. 9.12. Другой пример вставки новой точки в k-мерное (двумерное) дерево. Отправной точкой служит k-мерное дерево, полученное после вставки на рис. 9.11

Разберем первый пример шаг за шагом. Вставка начинается с вызова `insert(A, (-1.5, 2))`. В этот вызов не передается явно значение для параметра `level`, поэтому он получает значение по умолчанию, соответствующее корню (как определено в сигнатуре функции, это значение равно `0`).

`A <> null`, поэтому условие в строке 2 не выполняется; `A.point <> (-1.5, 2)`, и, соответственно, условие в строке 4 тоже не выполняется. В строке 6 имеем `-1.5 < 0`, поэтому `compare` вернет `-1`, произойдет переход в левое поддерево и выполнится вызов `insert(C, (-1,5, -2), 1)`.

Следующие несколько вызовов будут выполняться аналогично (как мы видели в примере для `search`), и по очереди будут выполнены вызовы `insert(D, (-1.5, -2), 2)`, `insert(null, (-1.5, -2), 3)`. В последнем условие в строке 2 выполнится, и будет создан новый узел `KdNode((-1.5, -2), null, null, 3)`, который и вернется предыдущему вызову. Предыдущий вызов в строке 7 сохранит в свойстве `.left` вновь созданный `KdNode`, а затем вернет `D`.

Обратите внимание, как в строке 6 разрешается неоднозначность при совпадении значения разделяющей координаты в текущем узле и в узле справа от него. Это решение может показаться не особенно важным, но, как мы увидим, оно будет иметь большое значение в операции удаления узлов.

Если посмотреть на трассировку стека для примера, изображенного на рис. 9.12, то можно заметить, что она полностью совпадает с трассировкой для предыдущего случая:

```
insert(A, (2.5, -3))
  insert(B, (-1.5, 2), 1)
    insert(E, (-1.5, 2), 2)
      insert(null, (-1.5, 2), 3)
      return new KdNode((-1.5, 2), null, null, 3)
    return E
  return B
return A
```

9.3.3. Сбалансированное дерево

Прежде чем перейти к более продвинутым методам, разработанным для k-мерных деревьев, сделаем шаг назад. В подразделе 9.2.3 отмечен один ключевой аспект k-мерных деревьев: нам нужно, чтобы дерево было сбалансированным. В предыдущих разделах уже было показано, что время выполнения поиска и вставки $O(h)$ пропорционально высоте дерева, поэтому для сбалансированного дерева $h = \log(n)$, и все эти методы будут выполняться за логарифмическое время.

К сожалению, k-мерное дерево не является самобалансирующимся, как, например, красно-черные деревья или деревья 2–3. То есть если произвести большое количество вставок и удалений в дереве, то в среднем дерево станет разбалансированным, но и это нельзя гарантировать. Более того, если разрешать неоднозначности при сравнении координат, всегда выбирая одно и то же направление, то доказано, что при выполнении большого количества операций баланс будет ухудшаться.

Для решения этой проблемы можно немного изменить метод `compare`, который определен в листинге 9.2, чтобы он никогда не возвращал `0`. В половине случаев, когда есть совпадение, он будет возвращать `-1`, а в другой половине — `+1`, тем

самым обеспечивая лучшую балансировку дерева. Но выбор должен быть определен точно, то есть нельзя делать случайный выбор и иметь идеальный баланс. Одно из возможных решений для реализации этой коррекции, как показано в листинге 9.5, состоит в том, чтобы возвращать `-1`, когда уровень узла четный, и `+1`, когда нечетный (или наоборот, это не имеет большого значения, главное — его предсказуемость).

Листинг 9.5. Измененный метод compare

Сигнатура метода осталась прежней

Сохраняется значение знака разности компонент, то есть то, что было результатом прежней версии метода

```
function compare(point, node)
  s ← sign(getPointKey(point, node.level) - getNodeKey(node))
  if s == 0 then
    return node.level % 2 == 0 ? -1 : +1
  else
    return s
```

Если это значение равно 0, то есть присутствует равенство сравниваемых координат, то в половине случаев выбирается левая ветвь, а в другой половине — правая. Иначе знак, 1 или –1, просто возвращается

Это решение помогает получить в среднем более сбалансированное дерево, но не дает никаких гарантий в каждом конкретном случае. На сегодняшний день не существует решения, позволяющего легко поддерживать баланс k-мерного дерева при сохранении времени выполнения `O(h)` для вставки.

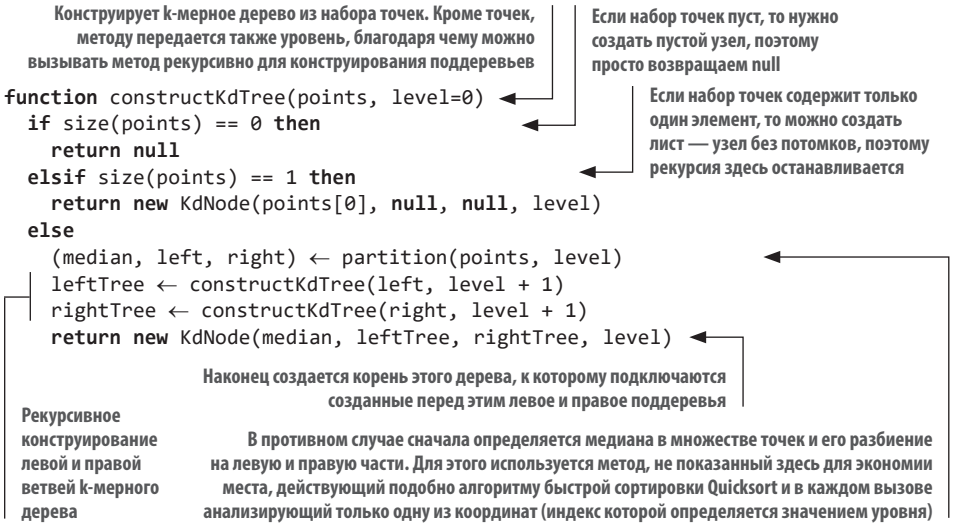
И все же если набор точек для вставки известен заранее (на момент создания k-мерного дерева), то можно пойти другим путем и найти оптимальный порядок вставки, позволяющий получить сбалансированное дерево. В том случае, когда дерево не будет изменяться после конструирования или количество вставленных/удаленных элементов пренебрежимо мало по сравнению с размером дерева, можно получить наихудшее сбалансированное k-мерное дерево, в котором `insert` и `search` будут выполняться за логарифмическое время в худшем случае.

В листинге 9.6 показан алгоритм создания сбалансированного k-мерного дерева из набора точек.

Принцип прост: дерево должно содержать все точки из заданного множества, и в идеале хотелось бы, чтобы левое и правое поддеревья имели одинаковое количество элементов. Для этого можно найти медиану в наборе точек относительно первой координаты и использовать ее в качестве опорной точки, или корня. В таком случае половина точек окажется в левой ветви корня и вторая половина — в правой. Но каждая из ветвей корня сама по себе является k-мерным деревом, поэтому указанный шаг можно повторить для левого и правого поддеревьев, с той лишь оговоркой, что на этот раз искать следует медианы по второй координате точек, и так далее для каждого уровня дерева. Достаточно просто отслеживать глубину рекурсии, которая будет определять текущий уровень дерева.

Ключевой момент в листинге 9.6 — вызов метода `partition` в строке 7. Ему следует передать текущий уровень, чтобы он знал, какие координаты использовать для сравнения точек. Результатом этого вызова будет кортеж, содержащий медиану массива точек и два новых массива с $(n - 1) / 2$ элементами в каждом, если `size(points) == n`.

Листинг 9.6. Конструирование сбалансированного дерева



Каждая точка в `left` будет «меньше» (в смысле координаты с индексом `level % k`) медианы, а каждая точка в `right` будет «больше» медианы; следовательно, используя эти два набора, можно рекурсивно построить оба (сбалансированных) поддерева.

Обратите внимание, что нельзя просто отсортировать массив один раз и рекурсивно разбить его на части, потому что на каждом уровне критерии сортировки меняются!

Чтобы понять, как это работает, рассмотрим вызов `convertKdTree([(0,5),(1,-1),(-1,6),(-0,5,0),(2,5),(2,5, 3),(-1, 1),(-1,5,-2)])`.

Медиана для этого множества точек (в смысле первой координаты, то есть медиана среди первых значений кортежей) равна либо `-0.5`, либо `0`: у нас четное количество элементов, поэтому технически имеем две медианы. Вы можете перепроверить значения, отсортировав массив.

Скажем, выбираем на роль медианы `-0.5`; в таком случае имеем:

```
(median, left, right) ← (-0.5,0), [(-1, 1),(-1.5,-2),(-1,6)], [(1,-1),
➥ (2.5,3),(2,5)(0,5)]
```

Соответственно, в строке 8 выполняется вызов `constructKdTree([(-1, 1),(-1.5,-2),(-1,6)], 1)` для создания левого поддерева. Он, в свою очередь, разделит полученный подмассив, сравнивая уже вторые координаты кортежей, по медиане с координатой `y = 1`, и получится:

```
(median, left, right) ← (-1, 1), [(-1.5,-2)], [(-1,6)]
```

Далее метод аналогичным образом обработает другие разделы, созданные в исходном массиве.

Каково время выполнения метода `constructKdTree`? Обозначим время создания k-мерного дерева с k размерностями из массива с n элементами как $T_k(n)$. Проверим

работу метода шаг за шагом: строки 2–5 выполняются за постоянное время, как и строка 10, которая просто создает новый узел. Строки 8 и 9 являются рекурсивными вызовами и будут вызываться для наборов точек, содержащих не более $n/2$ элементов, поэтому каждая из них выполнит $T_k(n/2)$ шагов.

Наконец, строка 7, где вызывается `partition`: медиану можно найти за линейное время и разбить массив из `n` элементов по опорной точке за $O(n)$ перестановок (или создать два новых массива, также за $O(n)$ присваиваний).

Подводя итоги, получаем вот такую формулу времени выполнения:

$$T_k(n) = 2 \times T_k(n/2) + O(n).$$

Решить это уравнение можно несколькими способами, например методом подстановки или телескопирования, но, вероятно, самый простой из них — использование основной теоремы[1]. Обсуждение всех этих методов выходит за рамки книги, поэтому я просто представлю готовое решение, предложив любознательным читателям разобраться во всем самим:

$$T_k(n) = O(n \times \log(n)).$$

Другими словами, метод конструирования сбалансированного дерева требует линейно-логарифмического[2] времени.

В завершение анализа оценим, какой объем дополнительной памяти необходим рассматриваемому методу. Для самого дерева потребуется $O(n)$ памяти. Однако это еще не все. Если массив разделять не на месте, а создавать копии для левого и правого подмассивов, то каждый вызов `partition` потребует $O(n)$ дополнительной памяти. Выводя формулу, аналогичную формуле времени выполнения, можно обнаружить, что она тоже имеет вид $M(n) = O(n \times \log(n))$.

Напротив, если массив разбивать на месте, то результат получится лучше:

$$M_k(n) = O(n).$$

Это объясняется тем, что для выполнения внутренних функций нужен лишь постоянный объем памяти плюс $O(n)$ памяти для самого дерева.

9.3.4. Метод remove

После `search` и `insert` можно перейти к третьей основной операции с контейнером: `remove`. И это несмотря на то, что в k-мерном дереве операция удаления используется довольно редко, а некоторые реализации вообще не предлагают этот метод. Как отмечалось в предыдущем разделе, k-мерные деревья не являются

---

1 https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms).

2 Время $n \log(n)$ часто называют линейно-логарифмическим, как комбинацию линейного и логарифмического времени.

самобалансирующимися, поэтому они работают лучше всего, когда создаются на основе статического набора точек и операции вставки и удаления выполняются очень редко.

И все же в любом реальном приложении вполне может понадобиться обновлять набор данных, поэтому опишем порядок удаления элементов. Рисунки 9.13 и 9.14 демонстрируют алгоритм работы метода `remove` и результат удаления из него точки *D* на примере нашего *k*-мерного дерева.

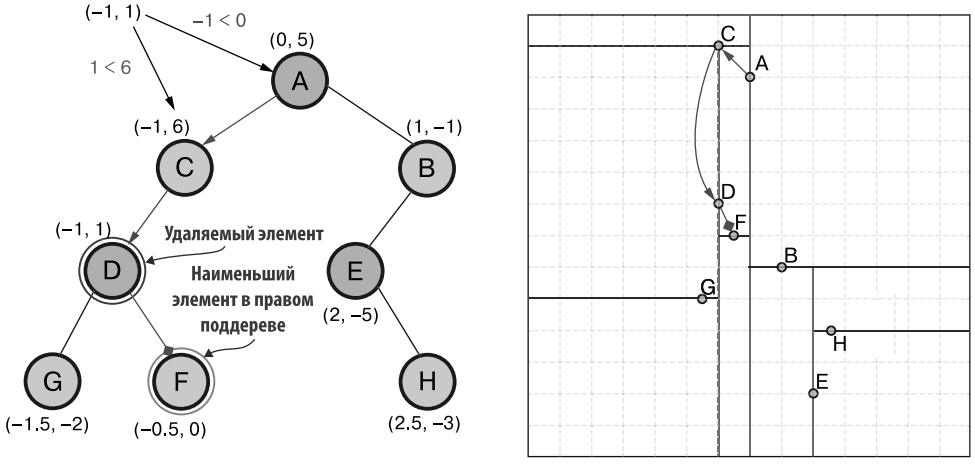


Рис. 9.13. Удаление точки D из нашего k-мерного дерева. Метод remove, как и в случае с двоичными деревьями поиска, состоит из (успешного) поиска удаляемого узла, после которого, если удаляемый узел является внутренним узлом хотя бы с одним поддеревом, выполняется поиск элемента, которым он может быть заменен. В этом примере удаляется внутренний узел

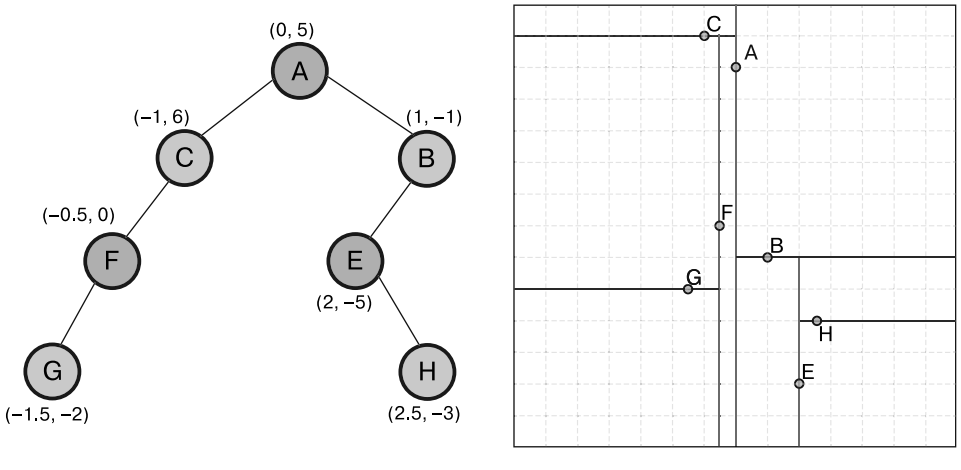


Рис. 9.14. k-мерное дерево после удаления точки D

Подобно методам `insert` и `search`, этот метод основан на операции удаления в двоичном дереве поиска. Однако есть две проблемы, которые заметно усложняют версию для k-мерного дерева, и обе они связаны с поиском замены для удаляемого узла.

Чтобы понять эти проблемы, нужно сделать шаг назад. В двоичных деревьях поиска при удалении узла можно столкнуться с одной из трех ситуаций (рис. 9.15).

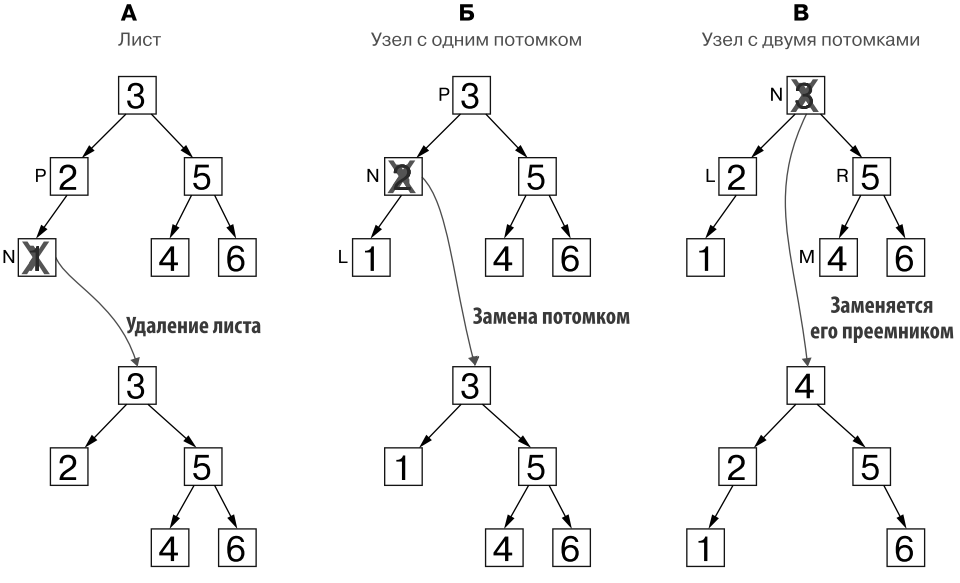


Рис. 9.15. Возможные случаи, возникающие при удалении узла из двоичного дерева поиска. *А*. Удаление листа. *Б*. Удаление узла с одним дочерним элементом. (Симметричный случай, когда дочерний элемент находится в правой ветви, обрабатывается аналогично.) *В*. Удаление узла с двумя дочерними элементами

1. Удаляемый узел является листом. В этой ситуации можно просто удалить узел из дерева, и все.
2. У удаляемого узла N есть только один потомок. Здесь простое удаление узла отсоединит его поддерево от основного дерева, но выйти из затруднительного положения можно, соединив родителя N с его дочерними элементами (независимо от того, находится ли он в левом или в правом поддереве N). Это не нарушит никаких инвариантов двоичного дерева поиска. Например, в случае, изображенном на рис. 9.15, *B*, узел N является левым дочерним узлом своего родителя P, поэтому он меньше или равен P. Аналогично все элементы в поддереве с корнем N будут меньше или равны P, в том числе L — потомок N.
3. Если удаляемый узел N имеет два дочерних элемента, то его нельзя просто заменить одним из его потомков (например, если заменить корень на рис. 9.15, *B*, правым дочерним элементом R, то будет необходимо связать левого потомка узла R с N, а это потребует наихудшего линейного времени (и к тому же все эти усилия будут предприняты впустую).

Вместо этого можно найти преемника[1] *N*. Это будет минимальный узел в его правом поддереве, назовем его *M*, или самый левый узел его правого поддерева. Отыскав преемника *M*, его можно удалить и заменить его значением значение *N*. Ни один инвариант при этом не будет нарушен, потому что *M* не меньше *N*, а *N*, в свою очередь, не меньше любого узла в своем левом поддереве.

Кроме того, *M*, безусловно, будет подходить под случай *А* или случай *Б*, потому что у самого левого узла не может быть левого потомка. Это означает, что удалить *M* будет легко и рекурсия остановится на *M*.

При переходе от обычных двоичных деревьев поиска к *k*-мерным деревьям заметны отличия между ними. Первое связано с тем, что на каждом уровне используется только одна координата для разделения точек на две ветви. Если нужно заменить узел *N* на уровне *i*, на котором используется координата $j = i \bmod k$, его преемник (для координаты *j*) будет находиться в правом поддереве *N*. Однако после перехода к дочернему элементу *N* этот узел будет использовать другую координату, $j_1 = (i + 1) \bmod k$, для разделения дочерних элементов. Как показано на рис. 9.16, такая ситуация означает, что преемник *N* не обязательно должен находиться в левом поддереве *R*.

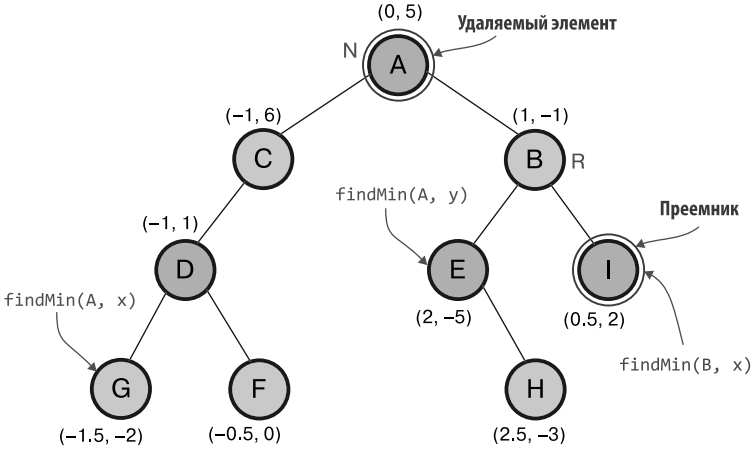


Рис. 9.16. Пример того, как потомок узла или в более общем смысле минимум поддерева в отношении определенной координаты может лежать в любом месте поддерева. Явно показаны результаты нескольких вызовов findMin для всего дерева и для узла B

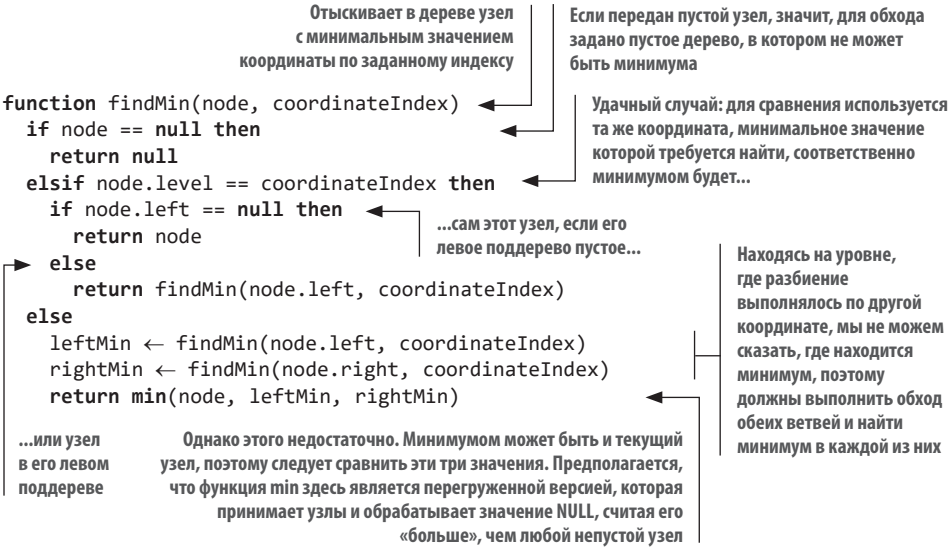
Это плохая новость. Потому что, в отличие от двоичного дерева поиска, где можно быстро пройти по правому поддереву *N* до его самого левого узла, чтобы найти преемника *N*, в *k*-мерном дереве это можно сделать только для уровня *l*, где $l \bmod k == i \bmod k$. На всех остальных уровнях придется пройти по обоим поддеревьям, чтобы найти минимум.

---

[1] Обратите внимание, что можно использовать предшественника *N* симметричным образом.

В листинге 9.7 показан псевдокод метода `findMin`. Первое отличие от двоичных деревьев поиска, которое можно заметить, — необходимость передачи индекса координаты, для которой ищется минимум. Например, можно вызвать `findMin(root, 2)`, чтобы во всем дереве найти узел с минимальным значением третьей координаты, при условии, что $k \geq 3$.

Листинг 9.7. Метод findMin




```
function findMin(node, coordinateIndex)
  if node == null then
    return null
  elsif node.level == coordinateIndex then
    if node.left == null then
      return node
    else
      return findMin(node.left, coordinateIndex)
  else
    leftMin ← findMin(node.left, coordinateIndex)
    rightMin ← findMin(node.right, coordinateIndex)
    return min(node, leftMin, rightMin)
```

Такая большая сложность метода `findMin`, к сожалению, отражается на времени его выполнения. Оно не может быть логарифмическим, как в случае с двоичными деревьями поиска, потому что приходится выполнять обход всех ветвей на всех уровнях, кроме тех, что соответствуют индексу координат, то есть в $(k - 1) / k$ случаях.

И на самом деле время работы `findMin` равно $O(n^{(k-1)/k})$. Если $k == 2$, то $O(n^{1/2}) = O(\sqrt{n})$, которое хуже логарифмического, но все же заметно лучше, чем время полного сканирования. Однако с ростом k это значение становится все ближе и ближе к $O(n)$.

Расширенная версия `findMin` решает проблему с координатами, и можно было бы надеяться просто задействовать ее в обычном методе `remove` двоичного дерева поиска, но, к сожалению, не все так просто. Есть еще одна проблема, которая усложняет ситуацию.

Вернемся к рис. 9.15, *B*: для двоичного дерева поиска имели место два счастливых случая, упрощающих удаление, — удаление листа и удаление узла с единственным потомком.

Для k-мерных деревьев легко удаляются только листья. К сожалению, даже если у удаляемого узла N есть только один дочерний элемент C, нельзя просто заменить N его дочерним элементом, потому что это изменит направление разбиения в C и всего его поддерева, как показано на рис. 9.17.

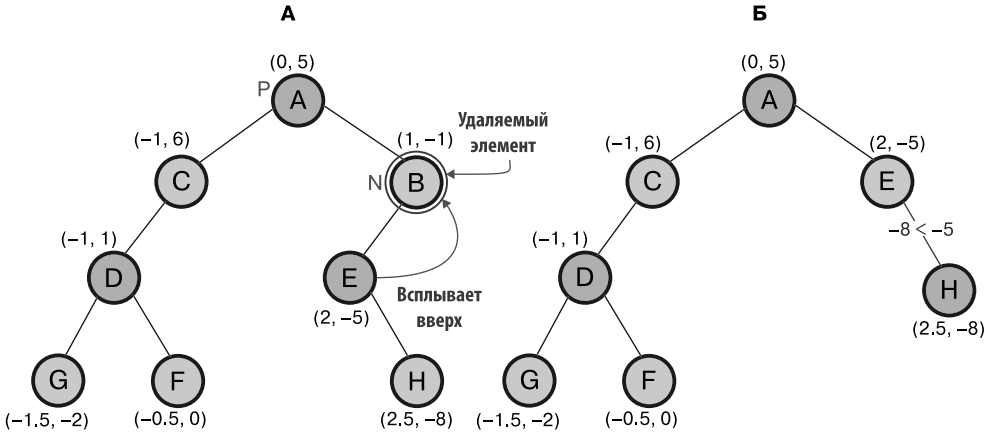


Рис. 9.17. Пример, когда для удаления узла из k-мерного дерева недостаточно просто заменить удаляемый узел его единственным потомком. *А*. Исходное дерево, из которого удаляется узел B. У этого узла есть только один потомок, поэтому в двоичном дереве поиска его можно просто заменить дочерним узлом. *Б*. Однако в k-мерном дереве это может привести к нарушению инвариантов k-мерного дерева, потому что перемещение узла на один уровень вверх изменяет координату, по которой выполняется разбиение

В этом примере удаляется узел *B*, имеющий только один дочерний узел и не имеющий правой ветви (симметричный случай работает аналогично). Если просто заменить *B* его дочерним узлом *E*, то последний окажется уровнем выше в дереве, как и все его дочерние элементы.

Итак, прежде узел *E* выполнял разделение узлов в своем поддереве по координате *x*, соответственно узел *H* находился справа от *E*, потому что *x*-координата (2,5) узла *H* больше, чем у *E* (2).

После перемещения *E* и его поддерева вверх оно должно будет использовать координату *y* для разделения узлов в своем поддереве. Но *y*-координата (−8) узла *H* меньше, чем у *E* (−5), поэтому узел *H* больше не может принадлежать правой ветви *E*, так как это нарушает инвариант *k*-мерного дерева.

Может показаться, что ситуация легко поправима, но не забывайте, что для этого придется переоценить каждый отдельный узел в поддереве *E* и перестроить его.

Потребуется время O(*n*), где *n* — количество узлов в поддереве, корнем которого является удаляемый узел.

Лучшим решением было бы заменить узел *N*, подлежащий удалению, его преемником или предшественником. Если у *N* есть только правый дочерний узел, то можно просто отыскать его преемника с помощью `findMin`, как описано в примере на рис. 9.16.

Можно ли заменить узел *N* его предшественником, когда *N* имеет только левого потомка? Как бы ни хотелось ответить утвердительно, проведя эти действия, можно столкнуться с другой проблемой.

Описывая метод `insert`, я упомянул, что порядок разрешения неоднозначности в `insert` влияет также на метод `remove`.

И действительно, на рис. 9.18 показан пример, когда этот аспект приобретает актуальность. Проблема в том, что, разрешая неоднозначность в `insert` и `search` и выбирая поворот вправо, мы неявно допускаем нарушение инварианта: для любого внутреннего узла *N* ни один узел в его левой ветви не будет иметь то же значение координаты, которая используется для разделения поддерева *N*. Это означает (обратите внимание на рис. 9.18), что ни один узел в левой ветви узла *B* не имеет координаты *y*, равной координате *y* узла *N*.

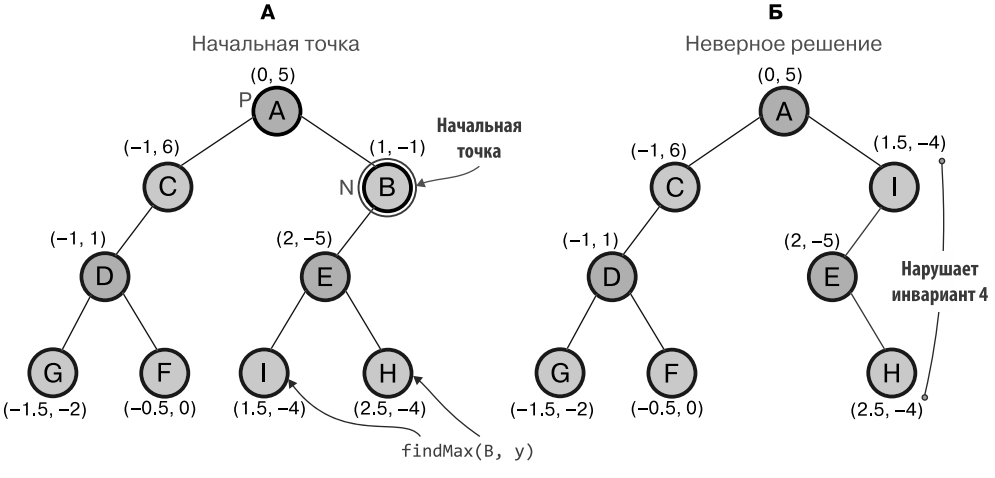


Рис. 9.18. Пример, показывающий, почему при удалении узла, имеющего только левый дочерний узел, нельзя заменить текущий узел минимальным значением из левой ветви. *Справа* показано, как H вызывает нарушение четвертого инварианта k-мерных деревьев, потому что он находится в левой ветви узла I, но имеет такое же значение координаты разделения, что и I

Однако если решиться заменить *N* максимальным значением из его левой ветви, то надо помнить, что, возможно, в старой левой ветви узла *N* имелся другой узел с той же координатой *y*. В нашем примере так и случилось бы, потому что имеется два узла с одинаковым максимальным значением *y*, узел *I* и узел *H*.

Переместив узел *I* на место узла *B*, мы лишимся полноценной возможности поиска, потому что после такого перемещения узел *H* никогда не будет найден методом `search`, описанным в листинге 9.3.

К счастью, выход из этой затруднительной ситуации не слишком сложен. Можно применить `findMin` к левой ветви, заменить *N* узлом *M*, найденным с помощью `findMin`, и подключить старую левую ветвь узла *N* справа к новому узлу, как показано на рис. 9.19.

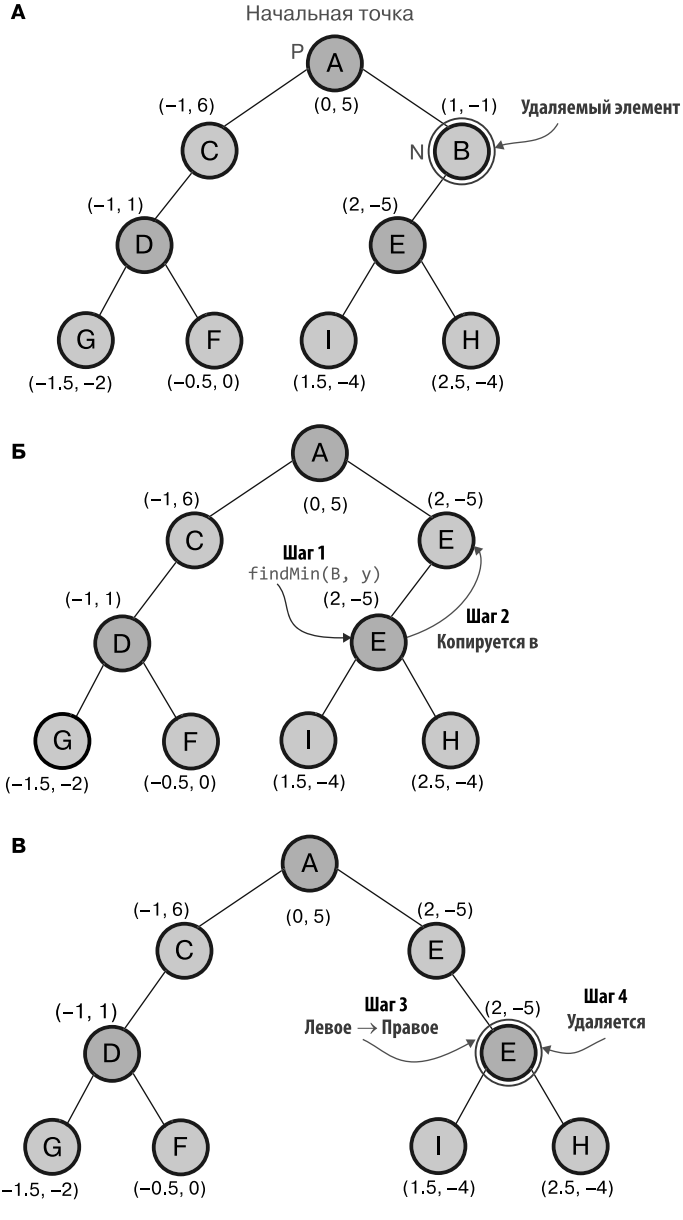
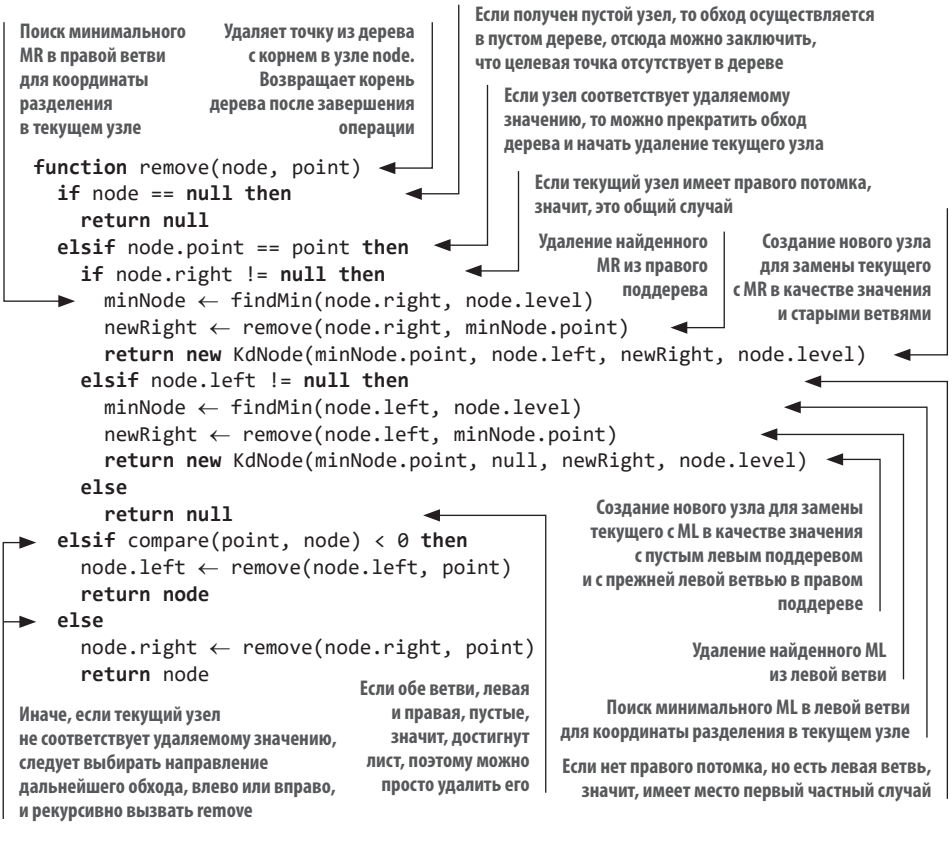


Рис. 9.19. Правильный порядок удаления узла N только с левым дочерним элементом. Как показано на рис. 9.18, найти максимум в левой ветви не получится. Вместо этого нужно найти минимальное значение M, использовать его для замены удаляемого узла, а затем подключить старую левую ветвь узла N справа к новому узлу. После этого нужно лишь удалить M из левой ветви, что потребует вновь вызвать remove. (К сожалению, мы не можем делать никаких предположений об этом вызове; он может иметь каскадный эффект и потребовать для удаления больше рекурсивных вызовов)

Далее остается только удалить старый узел M из правой ветви. Обратите внимание, что, в отличие от двоичных деревьев поиска, здесь нельзя делать никаких предположений относительно M, поэтому может потребоваться повторить эти шаги в рекурсивном вызове, удаляющем M.

В листинге 9.8 все высказанные соображения обобщены в реализации метода `remove` на псевдокоде.

Листинг 9.8. Метод remove



Учитывая, что вызовы `findMin` увеличивают общее время выполнения `remove`, оно больше не может быть логарифмическим (как для двоичных деревьев поиска). Чтобы оценить время работы этого метода более точно, снова обозначим его через $T_k(n)$, где k — размерность пространства значений, а n — количество узлов в дереве. Если внимательно рассмотреть каждое условное ветвление в алгоритме и предположить, что дерево сбалансировано, то:

- каждая ветвь условного оператора в строке 15 инициирует рекурсивный вызов примерно на половине узлов, что требует времени $T_k(n/2)$;
- блоки кода, выполняемые после условных операторов в строках 2 и 13, требуют только постоянного времени;

- условные операторы в строках 5 и 9 запускают блоки кода, которые создают новый узел за время O(1), вызывают `findMin` с временем выполнения $O(n^{1-1/k})$ и производят рекурсивные вызовы `remove`.

Последний случай — худший: неизвестно, где в ветви находится узел с минимальным значением. Он может оказаться далеко внизу дерева или в корне ветви. Поэтому в ситуации, подобной изображенной на рис. 9.17 (с отсутствующей левой или правой ветвью), в абсолютно худшем случае алгоритму может понадобиться рекурсивно вызывать `remove` на $n - 1$ узлах.

Однако мы предположили, что наше k-мерное дерево сбалансировано. В этих условиях левая и правая ветви должны иметь более или менее одинаковое количество узлов, поэтому если правая ветвь узла пустая, то наиболее вероятно, что левая ветвь имеет один узел, менее вероятно, что она имеет два узла, еще менее вероятно, что она имеет три узла и т. д. Для некоторой константы, например 5, можно сказать, что в ситуации, подобной той, что изображена на рис. 9.17, где узел имеет одну ветвь, маловероятно, что ветвь имеет больше, чем постоянное, количество узлов (скажем, пять). Соответственно, можно предположить, что во время вызова `remove` такой несбалансированный случай может встретиться в сбалансированном дереве самое большее постоянное количество раз. Или, точнее, при большом количестве удалений можно предположить, что амортизированное время выполнения блоков кода, начинающихся со строк 5 и 9, будет равно $T_k(n / 2)$.

Таким образом, наша рекуррентная формула приобретает вид:

$$T_k(n) = T_k(n/2) + O(n^{1-1/k}).$$

Используя третий случай основной теоремы, поскольку $1 - 1/k > \log_2(1) = 0$ и $(-/2)^{1-1/k} \leq n^{1-1/k}$, можно заключить, что амортизированное время выполнения `remove` на сбалансированном k-мерном дереве составит:

$$T_k(n) = O(n^{1-1/k}).$$

Другими словами, время выполнения `remove` определяется временем выполнения `findMin`. Это также означает, что в двумерном пространстве амортизированное время выполнения операции удаления будет равно $O(\sqrt{n})$.

9.3.5. Ближайший сосед

Теперь можно переходить к более интересной операции, поддерживаемой k-мерными деревьями: поиску ближайшего соседа (Nearest Neighbor, NN). Для начала ограничимся случаем поиска единственной точки в наборе данных, ближайшей к целевой (которая не обязательно должна находиться в том же наборе данных). Позже обобщим эту операцию для случая поиска произвольного числа m[1] точек, таких, что никакие другие точки в наборе данных не находятся ближе к целевой.

---

[1] Этот метод обычно называют поиском k ближайших соседей, но использование символа k может привести к путанице с размерностью дерева, поэтому для обозначения количества искомых точек будем использовать m или n.

В случае полного перебора нам пришлось бы сравнивать каждую точку, имеющуюся в наборе данных, с целевой, вычислять их относительные расстояния и запоминать ближайшую из них, как это обычно делается в неотсортированном массиве.

Однако *k*-мерное дерево, подобно отсортированному массиву, несет структурную информацию об относительном расстоянии и положении его элементов, и эту информацию можно использовать для более эффективного поиска.

В листинге 9.9 показан псевдокод, реализующий поиск ближайшего соседа. Однако, чтобы понять его, сначала нужно выяснить, как работает сам поиск ближайшего соседа.

Листинг 9.9. Метод nearestNeighbor

Иначе перед нами стоят три задачи: проверить, ближе ли текущий узел, чем ранее найденный ближайший сосед, обойти ветвь, расположенную на той же стороне разделения, где находится целевая точка, и проверить, можно ли исключить другую ветвь (или тоже нужно пройти ее)

Получен пустой узел, следовательно, обход осуществляется в пустом дереве, в котором не может быть соседа более близкого, чем уже найденный

Находит ближайшую точку к заданной цели target. Также принимает лучшего найденного на данный момент ближайшего соседа nn и расстояние до него nDist, чтобы упростить исключение. По умолчанию при вызове для корня эти значения равны null и infinity

```
function nearestNeighbor(node, target, (nnDist, nn)=(inf, null))
  if node == null then
    return (nnDist, nn)
  else
    dist ← distance(node.point, target)
    if dist < nnDist then
      (nnDist, nn) ← (dist, node.point)
    if compare(target, node) < 0 then
      closeBranch ← node.left
      farBranch ← node.right
    else
      closeBranch ← node.right
      farBranch ← node.left
    (nnDist, nn) ← nearestNeighbor(closeBranch, target, (nnDist, nn))
    if splitDistance(target, node) < nnDist then
      (nnDist, nn) ← nearestNeighbor(farBranch, target, (nnDist, nn))
    return (nnDist, nn)
```

Если это расстояние меньше, чем расстояние до текущего найденного ближайшего соседа, то нужно обновить сохраненные значения — ссылку на ближайшего соседа и расстояние до него

Проверяем, находится ли целевая точка с той же стороны от линии разделения, что и левая ветвь. Если это так, то, значит, левая ветвь ближе к целевой точке, иначе — дальше от нее

С помощью одной из вспомогательных функций, представленных в листинге 9.2, вычисляется расстояние между линией разделения, проходящей через текущий узел, и целевой точкой. Если это расстояние меньше, чем расстояние до текущего ближайшего соседа, то самая дальняя ветвь может содержать точки, расположенные ближе, чем текущий ближайший сосед (рис. 9.21)

Возвращаем ближайшую точку, найденную к данному моменту

Обход дальней ветви и обновление текущих хранимых значений: ссылки на ближайшего соседа и расстояние до него

Обходим выбранную ветвь в поисках ближайшего соседа. После этого обновляем хранимые значения со ссылкой на ближайшего соседа и расстояние до него, это необходимо, чтобы выполнить исключение более эффективно

Вычисляем расстояние от текущего узла до целевой точки target

Прежде всего следует заметить, что каждый узел дерева охватывает одну из прямоугольных областей, на которые разделено пространство, как показано на рис. 9.5 и 9.6. Поэтому сначала нужно найти область, содержащую целевую точку *P*. Это послужит хорошей отправной точкой для поиска, так как вполне вероятно, что точка, хранящаяся в листе, который охватывает эту область (в этом примере *G*), окажется среди ближайших точек к *P*.

Однако можно ли быть уверенными, что *G* является ближайшей точкой к *P*? Было бы идеально, но, к сожалению, в действительности все может сложиться по-другому. На рис. 9.20 показан этот первый шаг алгоритма — обход пути в дереве от корня до листа, чтобы найти наименьшую область, содержащую *P*.

Как видите, в каждой промежуточной точке проверяется ее расстояние до цели, потому что любая из них может оказаться ближе, чем лист. Даже если промежуточные точки охватывают более крупные области, чем листья, внутри каждой области нет ничего, что говорило бы о том, где могут лежать точки набора данных. Если бы точка *A* на рис. 9.20 находилась в точке (0, 0), то дерево имело бы ту же форму, но *P* была бы ближе к *A* (корень), чем *G* (лист).

И даже этого недостаточно. Найдя область, содержащую *P*, нельзя с уверенностью утверждать, что в соседних областях нет одной или нескольких точек, еще более близких, чем ближайшая точка, найденная во время этого первого обхода.

Рисунок 9.21 прекрасно иллюстрирует эту ситуацию. На данный момент мы обнаружили, что *D* является ближайшей точкой (среди встреченных во время обхода) к *P*, поэтому действительный ближайший сосед не может находиться на расстоянии большем, чем расстояние между *D* и *P*. Соответственно, можно нарисовать окружность (гиперсферу при большем числе измерений) с центром в *P* и радиусом, равным `dist(D, P)`. Если в этот круг попадают другие разбиения, то соответствующие им области могут содержать точки, расположенные ближе к цели, чем *D*, и нужно проверить их.

Как узнать, пересекается ли область с гиперсферой текущего ближайшего соседа? Это сделать просто: каждая область связана с создавшими ее разделительными линиями. Обход выполняется по одной стороне разделения (той, где находится *P*), но, если расстояние между разветвлением и *P* меньше расстояния до текущего ближайшего соседа, это говорит о том, что гиперсфера пересекает и другую область.

Для проверки всех разделов, пересекающих гиперсферу ближайшего соседа, нужно вернуться назад по пути обхода дерева. В этом примере следует вернуться к узлу *D* и проверить расстояние между *P* и вертикальной линией разделения, проходящей через *D* (которая, в свою очередь, является просто разницей координат *x* двух точек). Так как это расстояние меньше расстояния до *D*, текущего ближайшего соседа, то нужно посетить и другую ветвь *D*. Под «посетить» имеется в виду обход дерева по пути от *D* до листа, ближайшего к *P*. В процессе этого обхода обнаруживается узел *F*, находящийся ближе, чем *D*, поэтому обновляется хранимая ссылка на текущего ближайшего соседа (и расстояние до него: на рис. 9.21 можно видеть, что при этом уменьшился радиус гиперсферы ближайшего соседа, обозначающей область, где могут находиться более близкие соседи).

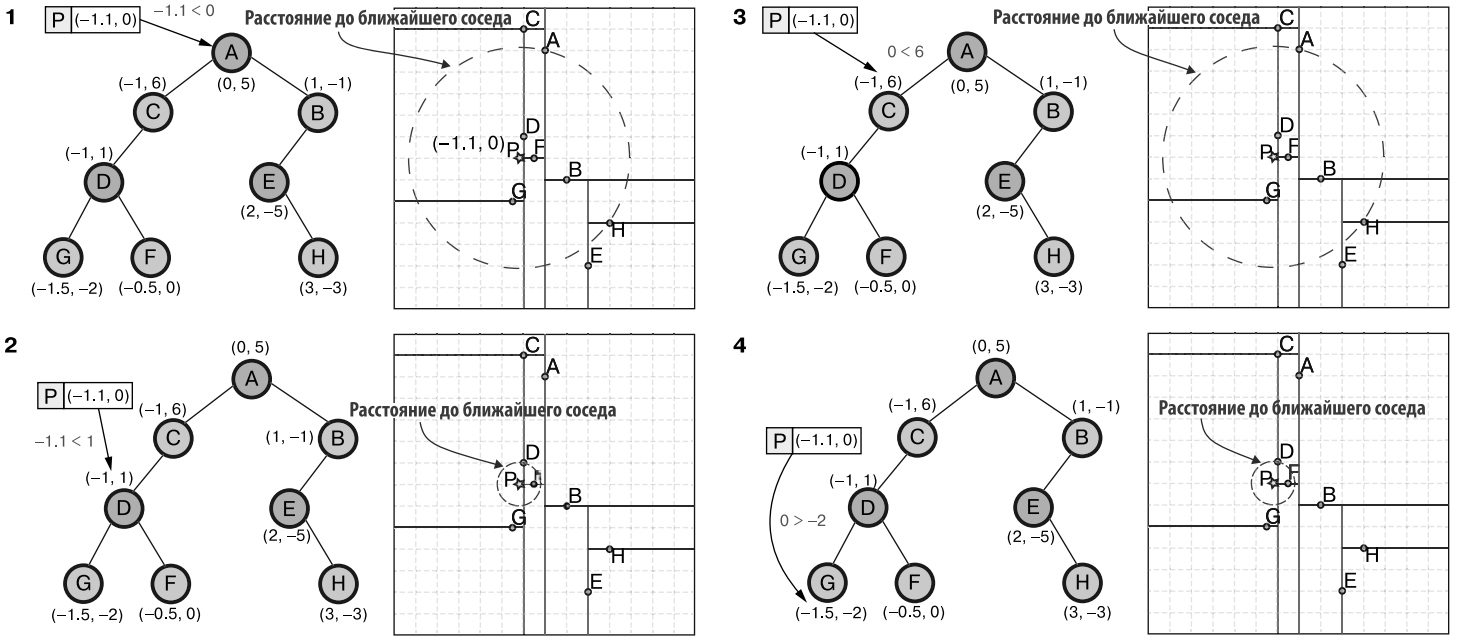


Рис. 9.20. Первые несколько шагов алгоритма поиска ближайшего соседа. Начальная фаза — поиск в дереве, во время которого определяется расстояние до каждого узла (точки), встреченного в пути: точнее, если нужно найти N ближайших соседей для P, то следует запомнить N наименьших найденных расстояний. В данном случае N = 1. Если поиск увенчался успехом, то определенно есть ближайший сосед на расстоянии 0. Иначе, если поиск заканчивается неудачей, нет уверенности в том, что найден действительно ближайший сосед. В этом примере точка на минимальном расстоянии, найденная в первой фазе поиска, — точка D, но, как видно на рисунке, нет уверенности в том, что другая ветвь дерева не содержит подходящей точки в круге с радиусом dist(D)

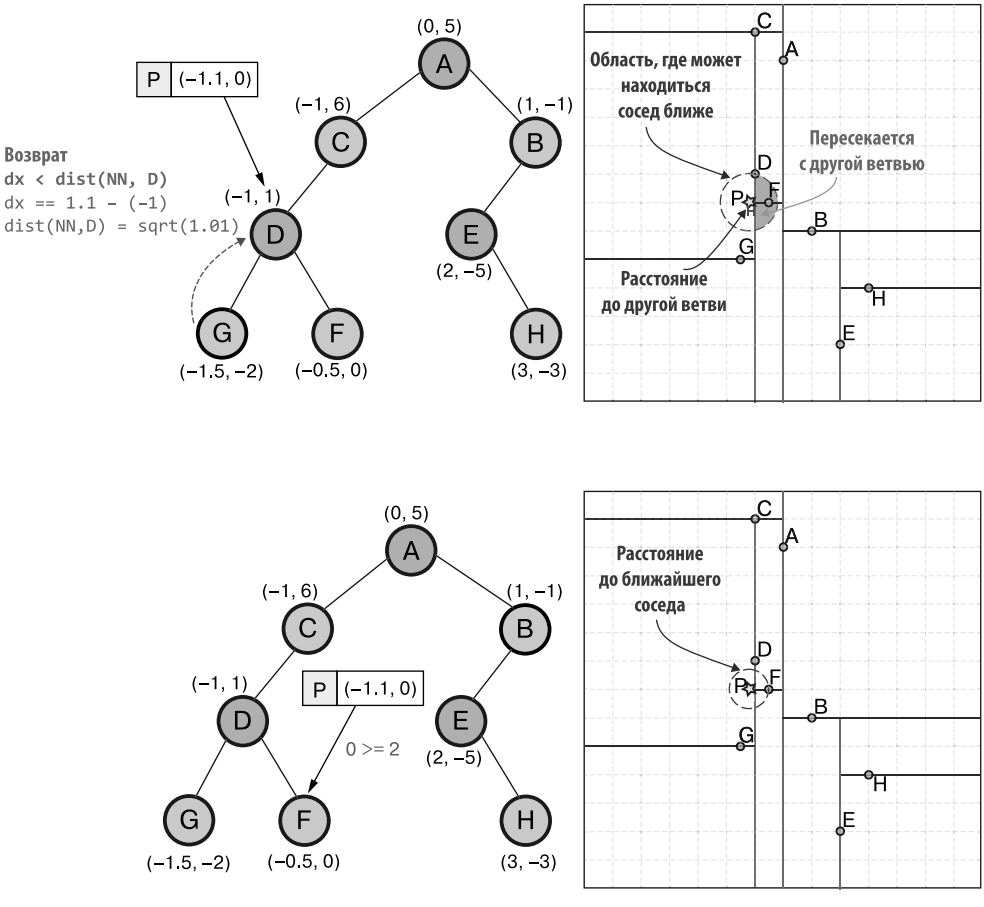


Рис. 9.21. Второй шаг после неудачного поиска — возврат назад для проверки других ветвей дерева, где могут находиться более близкие точки. Они действительно могут там быть, потому что в процессе обхода дерева циклически перебираются координаты для сравнения на каждом уровне, поэтому алгоритм не всегда движется в направлении ближайшей точки, но он вынужден выбирать ту или иную сторону относительно опорной точки, исходя только из одной координаты. Это означает, что ближайший сосед и целевая точка P могут быть расположены по разные стороны от опорной точки. В этом примере, достигнув точки D, создающей вертикальное разделение, следовало бы двигаться влево, как показано на рис. 9.20. К сожалению, целевая точка P и ее ближайший сосед лежат по разные стороны от линии разделения, проходящей через D. Поэтому, достигнув конца пути, нужно вернуться назад и проверить другие ветви

Мы закончили? Нет! Нужно продолжать возвращаться к корню. Далее происходит возврат к узлу *C*, но расстояние до его линии разделения больше радиуса гиперсферы текущего ближайшего соседа (к тому же он не имеет правой ветви), поэтому дальше происходит возврат к узлу *A*, как показано на рис. 9.22.

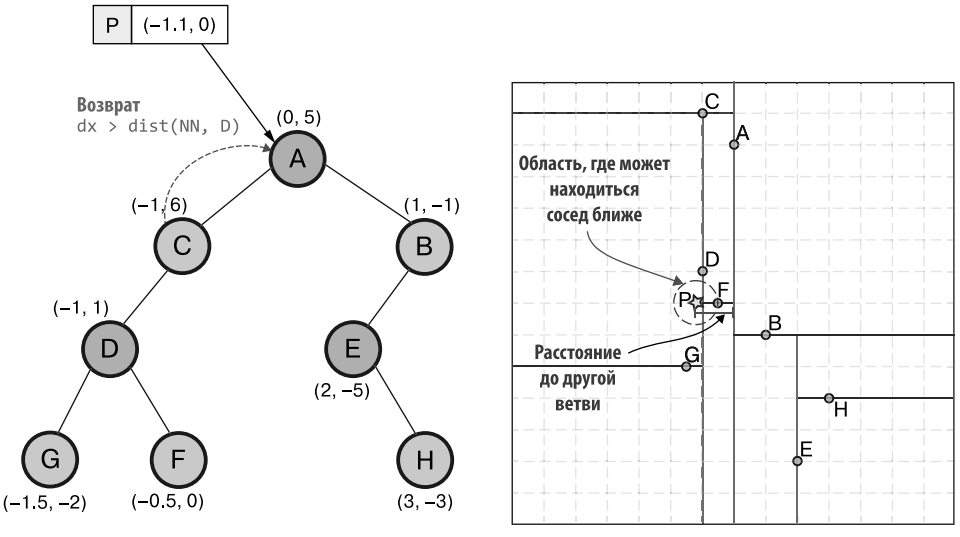


Рис. 9.22. Нужно вернуться к корню. Однако если бы требовалось проверять все возможные ветви дерева, это было бы не лучше, чем полное сканирование всего списка точек. Но есть возможность воспользоваться всей имеющейся информацией, чтобы сократить поиск. В геометрическом представлении справа показана наглядная подсказка, почему бесполезно проверять правую ветвь узла A. Если снова посмотреть на рис. 9.21, то можно заметить, что на самом деле уже было известно, а именно: что нельзя исключить правую ветвь D, не пройдя ее

На этапе поиска в узле *A* была выбрана левая ветвь, то есть *P* находится в левой полуплоскости. Если бы требовалось пройти по правому поддереву *A*, то координата *x* всех точек в этом поддереве была бы больше или равна координате *A*. Следовательно, минимальное расстояние между *P* и любой точкой в правом поддереве *A* не может быть меньше расстояния между *P* и его проекцией на вертикальную линию, проходящую через *A* (то есть линию разделения *A*). Другими словами, минимальное расстояние для любой точки справа от *A* будет по крайней мере абсолютным значением разницы между координатами *x* точек *P* и *A*.

Соответственно, можно отказаться от поиска в правой ветви *A*, а поскольку это корень, работа алгоритма заканчивается. Обратите внимание, что чем выше алгоритм поднимается по дереву, тем больше ветвей (и областей), которые можно исключить из поиска, и тем больше экономия.

Описанный метод можно обобщить для поиска произвольно большого множества ближайших точек в наборе данных — он также известен как метод поиска *n* ближайших соседей[1].

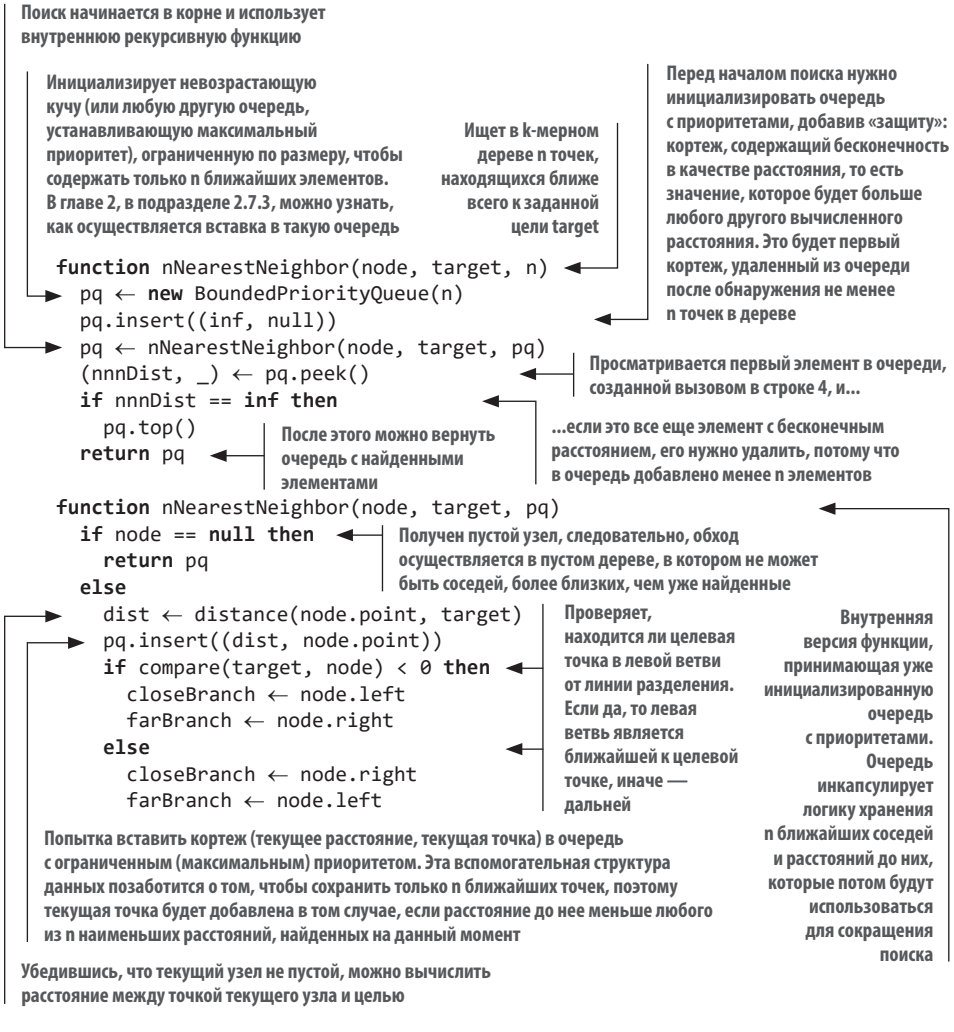
[1] Этот метод обычно называют поиском *k* ближайших соседей, но использование символа *k* может привести к путанице с размерностью дерева, поэтому для обозначения количества искомых точек будем использовать *m* или *n*.

Для этого нужно всего лишь:

- вместо расстояния до одной точки запоминать m кратчайших расстояний, чтобы найти m ближайших точек;
- на каждом шаге использовать расстояние до m-й ближайшей точки, чтобы нарисовать гиперсферу ближайшего соседа и сократить поиск;
- для хранения этих m расстояний можно использовать очередь с ограниченными приоритетами. Подобная структура данных описана в главе 2, в подразделе 2.7.3, где рассказывается о методе поиска m самых больших значений в потоке чисел.

В листинге 9.10 представлен псевдокод метода `nNearestNeighbor`.

Листинг 9.10. Метод nNearestNeighbor



```
    pq ← nNearestNeighbor(closeBranch, target, pq)
(nnnDist, _) ← pq.peek()
if splitDistance(target, node) < nnnDist then
  pq ← nNearestNeighbor(farBranch, target, pq)
return pq
```

Возвращает очередь с точками, найденными к данному моменту

Обход дальней ветви и обновление содержимого очереди с текущим множеством ближайших соседей

Непременно нужно обойти ближайшую ветвь в поисках ближайшего соседа. Сделать это и обновить очередь с приоритетами, хранящую расстояние до n-го ближайшего соседа, необходимо, чтобы выполнить исключение более эффективно

С использованием одной из вспомогательных функций в листинге 9.2 вычисляется расстояние между линией разделения, проходящей через текущий узел, и целевой точкой. Если это расстояние меньше расстояния до текущего n-го ближайшего соседа, хранимого в очереди, то это означает, что дальняя ветвь может содержать более близкие точки

Алгоритму нужно получить расстояние до n-го ближайшего соседа. Это можно сделать, получив кортеж на вершине очереди с ограниченными приоритетами. Обратите внимание: если в очереди меньше n элементов, то, поскольку в строке 3 в очередь добавляется кортеж с расстоянием, равным бесконечности, он будет находиться наверху кучи, пока не будет добавлено n точек, поэтому здесь расстояние nnnDist будет равно бесконечности, пока в очередь не будет добавлено n точек. Примечание: символ подчеркивания здесь играет роль заполнителя и означает, что нас интересует только значение второго элемента пары

Теперь поговорим о времени выполнения алгоритма поиска ближайшего соседа. Начнем с плохих новостей: в худшем случае, чтобы найти всех ближайших соседей, может потребоваться обойти все дерево, даже если оно сбалансировано.

На рис. 9.23 показано несколько примеров такого вырожденного случая. Второй пример сконструирован искусственно и представляет пограничный случай, когда все точки лежат на окружности, однако пример на рис. 9.23, *А*, показывает то же дерево, которое использовалось в предыдущих примерах, и демонстрирует, что даже на случайных сбалансированных деревьях можно специальным образом подобрать целевую точку и найти контрпримеры, когда метод показывает плохую производительность.

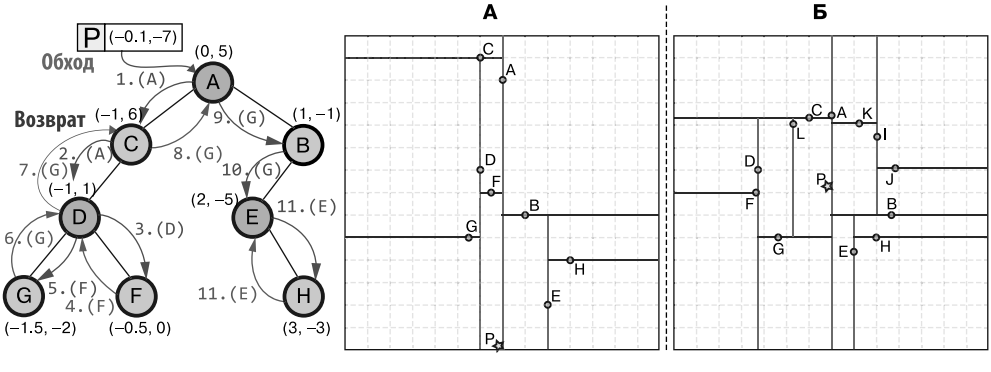


Рис. 9.23. Пограничные случаи для поиска ближайшего соседа, требующие обхода всего дерева. *А*. Пример, построенный на k-мерном дереве с рис. 9.20–9.22. Тщательно выбрав целевую точку, можно заставить алгоритм обойти все дерево, как показано на схеме *слева*. *Б*. Здесь можно видеть только пространственное представление крайнего случая, когда все точки лежат на окружности и в качестве целевой точки выбран центр этой окружности. Обратите внимание, что расстояние от P до линии разделения всегда будет меньше радиуса окружности (который, в свою очередь, является расстоянием до ближайшего соседа)

Так что, к сожалению, особого выбора у нас нет: в худшем случае время выполнения этого алгоритма равно $O(n)$. Это плохие новости. К счастью, есть чем подсластить горькую пилюлю.

Как оказывается, среднее время выполнения поиска ближайшего соседа в сбалансированном k-мерном дереве равно $O(2^k + \log(n))$. Доказательство этой вероятностной оценки особенно сложное и требует слишком много места, чтобы подробно описать его здесь. Вы можете найти его в оригинальной статье Джона Бентли (Jon Bentley), где впервые были представлены k-мерные деревья.

Тем не менее, чтобы дать вам хотя бы общее представление о том, почему это работает именно так, я предлагаю рассмотреть двумерное пространство: одна точка делит его на две половины, две точки образуют три области, три точки — четыре области и т. д. В общем случае n точек образуют $n + 1$ область. Если дерево сбалансировано и число n достаточно велико, то можно предположить, что все эти области имеют примерно одинаковые размеры.

Теперь предположим, что набор данных охватывает унитарную область[1]. Начиная поиск ближайшего соседа, сначала проходим по дереву от корня до ближайшего листа[2], а для сбалансированного дерева это означает обход $O(\log(n))$ узлов. Поскольку мы предположили, что все области имеют примерно одинаковые размеры, а областей у нас $n + 1$, то площадь области (тоже прямоугольной), охватываемой ближайшим листом, будет примерно равна $1/n$. Это означает, что существует достаточно высокая вероятность, что расстояние между целевой точкой и ближайшим соседом, найденным во время этого обхода, не превышает половины диагонали области, что, в свою очередь, меньше квадратного корня из площади области, или, другими словами, $\sqrt{1/n}$.

Следующий шаг алгоритма — возврат для посещения всех областей, находящихся в пределах этого расстояния от целевой точки. Если все области имеют правильную форму и одинаковый размер, то любая область в пределах заданного расстояния должна находиться в одном из соседних (к области нашего листа) прямоугольников. А из геометрии известно, что в ситуации с правильными прямоугольниками одинакового размера, которые можно аппроксимировать прямоугольной сеткой, каждый прямоугольник имеет восемь соседей. Однако из рис. 9.24 видно, что в среднем, даже если бы пришлось пройти не более восьми дополнительных ветвей, скорее всего, достаточно будет проверить только четыре из них, потому что только те, которые примыкают к текущей области, будут находиться в пределах данного расстояния. Отсюда общее среднее время выполнения получается равным $O(4 \times \log(n))$.

Если переход осуществляется в $\mathbb{R}^3$, число возможных соседей для каждого куба увеличится до 26, но минимальное расстояние будет равно $\sqrt[3]{1/n}$. Применяя аналогичные соображения, можно сделать вывод, что достаточно близко будет расположено

---

[1] Мы можем определить специальную единицу измерения площади, поэтому ее всегда можно вообразить.

[2] Одного листа, ближайшего к целевой точке, как показано в листинге 9.9.

не более восьми соседних областей, где могут находиться точки в пределах минимального расстояния, найденного до сих пор; те же рассуждения применимы при переходе к $\mathbb{R}^4$ и т. д.

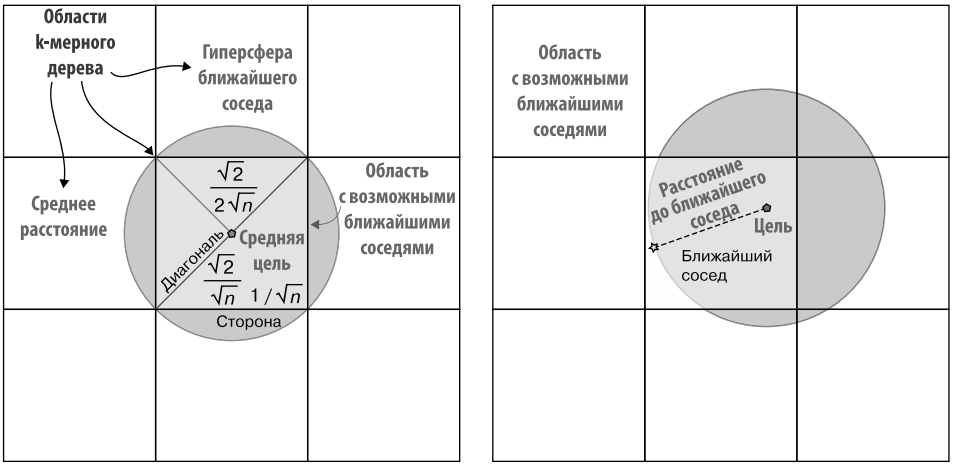


Рис. 9.24. Идеально правильное разбиение k-мерного дерева квадратными ячейками. *Слева* показано, что можно считать средним случаем, когда точка листа дерева находится в центре области. В этой ситуации максимальное расстояние до цели внутри области равно половине длины диагонали квадрата. Если нарисовать окружность, описывающую квадрат, она пересечет только четыре области, прилегающие к сторонам текущей. Справа показан пример другого, более общего и менее оптимистичного случая. Даже если расстояние больше среднего, гиперсфера с центром в целевом узле пересекает только четыре другие области. Конечно, в худшем случае она может пересекать восемь других областей

Таким образом, после обнаружения области, где находится целевая точка, нужно исследовать еще $O(2^k)$ точек, что дает общее время выполнения $O(2^k + \log(n))$. Для каждого конкретного набора данных k является константой, и теоретически эту величину можно игнорировать при анализе «О больших», но в общем случае k считается параметром, потому что производительность этого метода измеряется как в отношении размера, так и количества размерностей массива k-мерного дерева. Также очевидно, что при больших значениях k величина 2^k становится настолько большой, что начинает доминировать во времени выполнения, поскольку

$$\log(n) > 2^k \Leftrightarrow n > 2^{2^k},$$

и на практике для $k \geq 7$ уже нет шансов иметь достаточно большой набор данных, удовлетворяющий неравенству. Однако при $n > 2^k$ этот метод по-прежнему выглядит привлекательнее, чем поиск прямым перебором.

И последнее соображение: исключение ветвей из поиска сильно зависит от «качества» ближайшего соседа, найденного к текущему моменту. Чем меньше расстояние

до текущего ближайшего соседа, тем больше ветвей можно исключить и тем более высокую скорость можно получить. Поэтому так важно обновить это значение как можно раньше (в нашем коде это делается для каждого узла при первом его посещении) и сначала пройти по наиболее перспективным ветвям. Конечно, непросто определить, какая ветвь наиболее перспективная. Хорошим, хотя и несовершенным индикатором может служить расстояние между целевой точкой и линией разделения ветвей. Чем ближе цель, тем больше должно быть пересечение между гиперсферой ближайшего соседа (внутри которой еще можно найти точку, находящуюся ближе) и областью по другую сторону линии разделения. Вот почему дерево просматривается с использованием поиска в глубину: сначала выполняется возврат к более коротким ветвям в надежде, что по достижении более длинных ветвей, находящихся ближе к вершине дерева, их можно будет исключить.

9.3.6. Поиск области

Первоначально *k*-мерные деревья предназначались для поиска ближайших соседей, но эти же деревья оказались особенно эффективными в операциях другого типа: определения пересечения набора данных с заданной областью в *k*-мерном пространстве.

Теоретически область может иметь любую форму, но эта операция становится значимой, только если есть возможность эффективно исключить ветви из поиска, что сильно зависит от морфологии области.

С практической точки зрения основной интерес представляют два случая:

- гиперсферические области, как показано на рис. 9.25; определение точек, находящихся на определенном расстоянии от заданной точки;
- гиперпрямоугольные области, как показано на рис. 9.26; определение точек, значения которых находятся в заданных диапазонах.

В отношении сферических областей ситуация оказывается такой же, как и при поиске ближайшего соседа: в результаты нужно включить все точки, находящиеся в пределах некоторого расстояния от заданной точки. Это чем-то напоминает NN-поиск, когда расстояние до ближайшего соседа никогда не обновляется, и вместо одной точки (ближайшего соседа — nearest neighbor, NN) выявляются все точки, расположенные ближе заданного расстояния.

На рис 9.25 показано, как исключаются ветви из поиска. Дойдя до развилки, алгоритм должен пройти по ветви, находящейся по ту же сторону от разделительной линии, что и центр области поиска *P*. Для другой ветки проверяется расстояние между *P* и ее проекцией на линию разделения. Если это расстояние меньше или равно радиусу области поиска, значит, между этой ветвью и областью поиска есть область пересечения, поэтому нужно выполнить обход и этой ветви; иначе ветвь из поиска можно исключить.

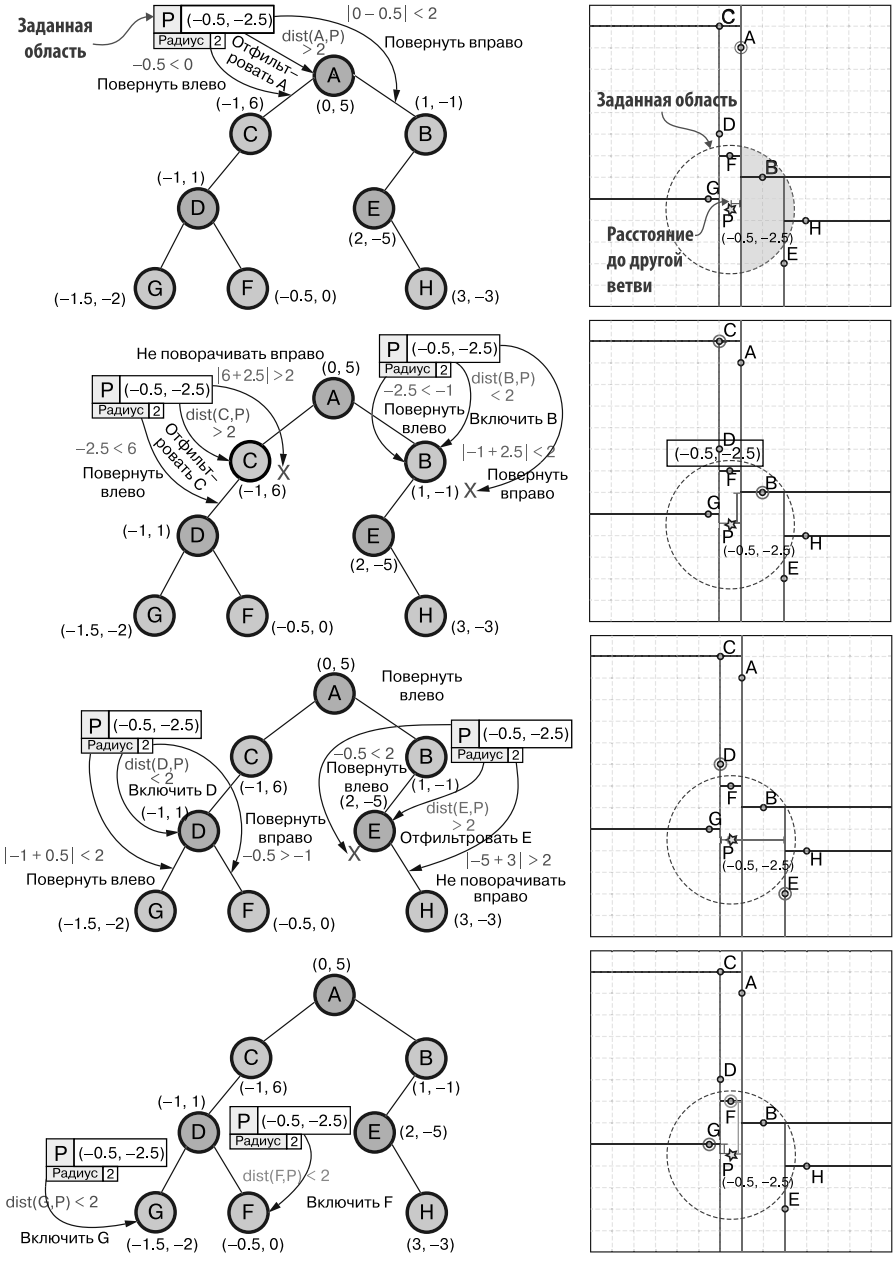


Рис. 9.25. Поиск области в k-мерном дереве возвращает все точки в дереве, попадающие в заданную гиперсферу. По сути, выполняется поиск точек в пределах заданного евклидова расстояния от центра сферы. Процесс начинается с корня, который не попадает в пределы сферы. Заданная точка находится в левой ветви A, поэтому ее нужно обойти, но даже притом что A находится за пределами сферы, линия разделения, проходящая через нее, пересекает сферу, поэтому часть сферы также пересекает правую ветвь A (выделена на верхнем левом рисунке). Далее для экономии места показано параллельное выполнение последующих шагов для всех ветвей на заданном уровне. (Это также может служить намеком, что перечисленные шаги могут выполняться параллельно)

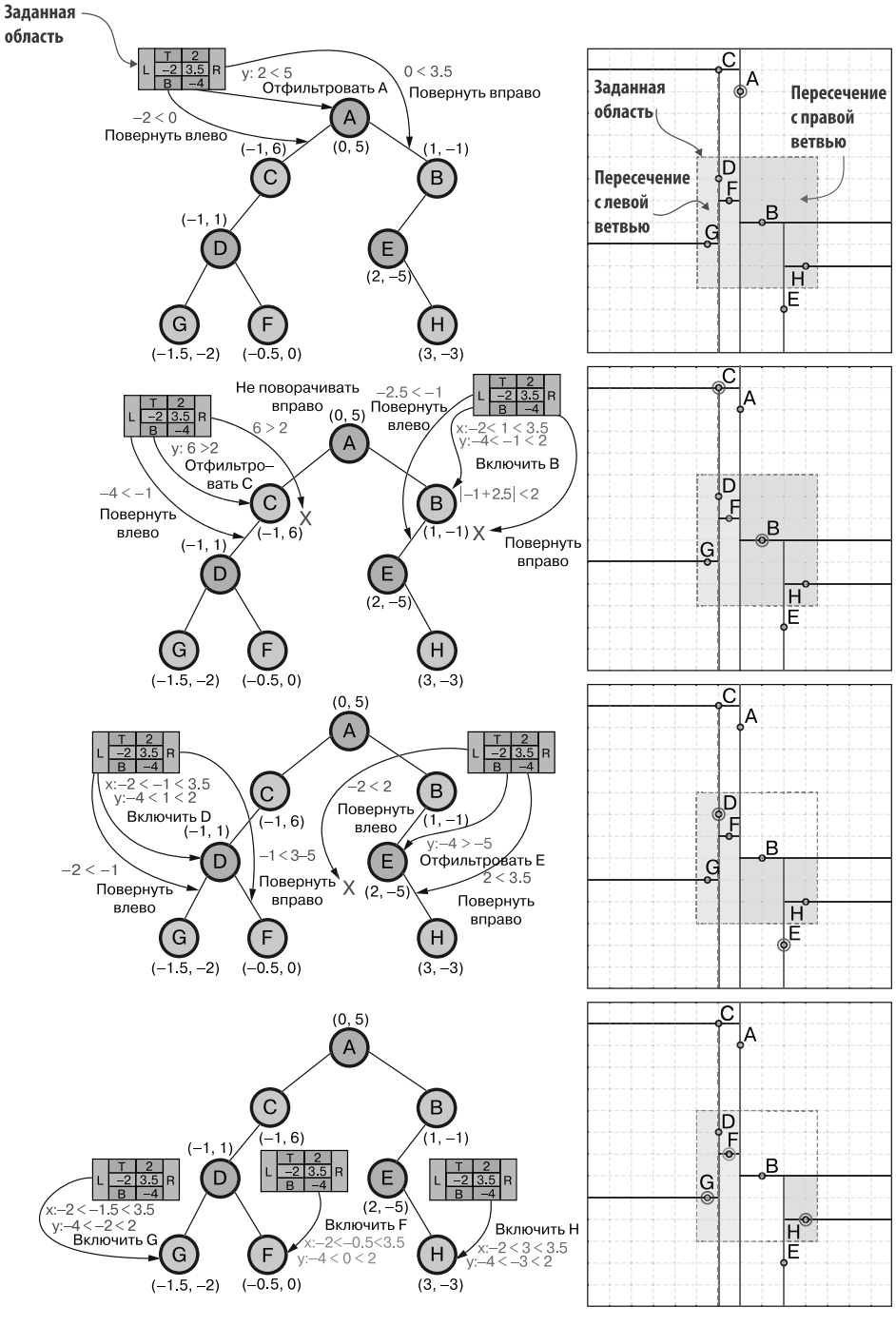
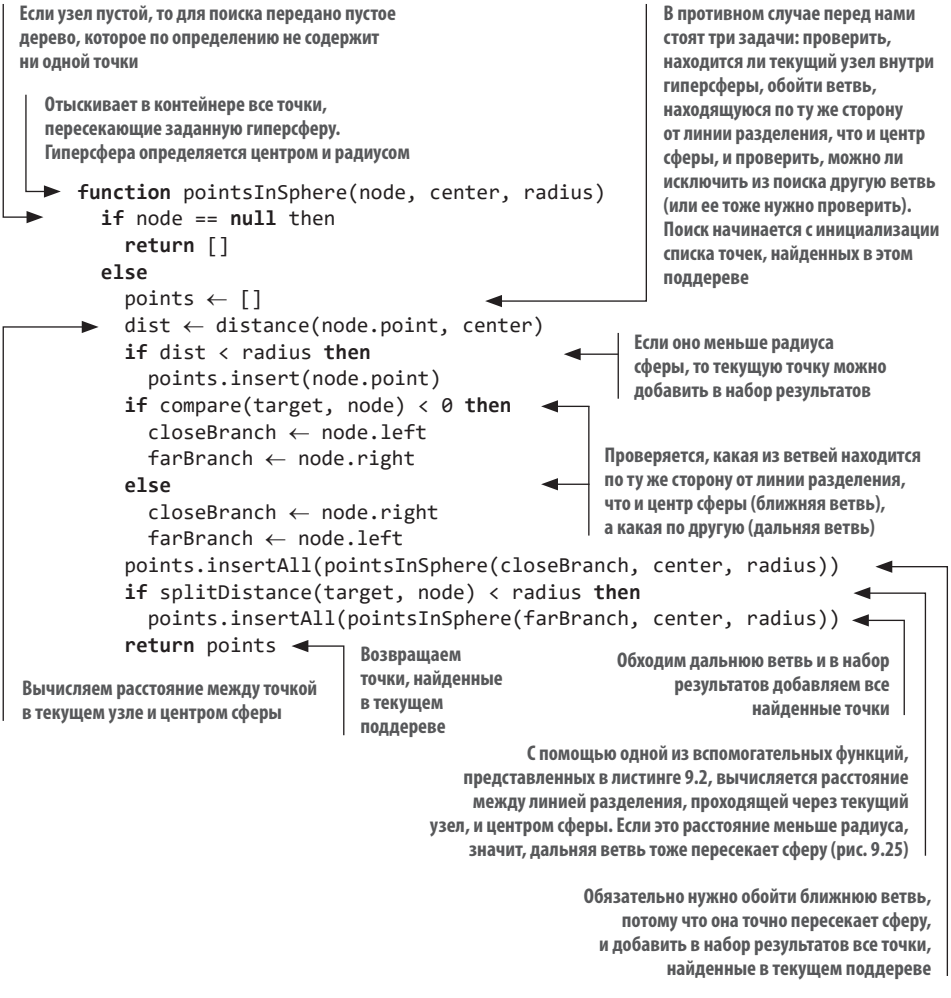


Рис. 9.26. Поиск области в k-мерном дереве возвращает все точки дерева, попадающие в заданный гиперпрямоугольник. По сути, выполняется поиск точек, каждая координата которых удовлетворяет двум неравенствам, то есть должна находиться в пределах диапазона. В частности, в этом двумерном примере координаты x точек будут находиться в диапазоне от –2 до 3,5, а координаты y — в диапазоне от –4 до 2

В листинге 9.11 показан псевдокод с реализацией этого метода. Как нетрудно заметить, приводимая реализация очень похожа на реализацию обычного NN-поиска.

Листинг 9.11. Метод pointsInSphere




```
function pointsInSphere(node, center, radius)
  if node == null then
    return []
  else
    points ← []
    dist ← distance(node.point, center)
    if dist < radius then
      points.insert(node.point)
    if compare(target, node) < 0 then
      closeBranch ← node.left
      farBranch ← node.right
    else
      closeBranch ← node.right
      farBranch ← node.left
    points.insertAll(pointsInSphere(closeBranch, center, radius))
    if splitDistance(target, node) < radius then
      points.insertAll(pointsInSphere(farBranch, center, radius))
    return points
```

Другой вариант поиска области — поиск области прямоугольной формы. Единственное отличие этого варианта поиска от `pointsInSphere` заключается в проверке необходимости исключить ветвь из поиска. Если предположить, что прямоугольник ориентирован вдоль декартовых осей, используемых для разделения, это упростит проверку, как показано на рис. 9.26. Но предположим, что мы находимся на горизонтальной линии разделения. В таком случае стоит выяснить, пересекает ли линия разделения область поиска, и если да, то нужно обойти обе ветви, а если она проходит выше или ниже области, то по ее положению можно определить, какую из ветвей

следует исключить из поиска. Для этого достаточно сравнить координату y точки текущего узла — назовем ее N_y — с верхней (R_t) и нижней (R_b) координатами y прямоугольной области. Всего возможно три случая:

- $R_b \leq N_y \leq R_t$ — нужно обойти обе ветви;
- $N_y > R_t$ — можно исключить левую ветвь;
- $R_b > N_y$ — можно исключить правую ветвь.

Аналогично выполняется проверка для вертикальных линий разделения, только вместо y проверяется координата x. Такой алгоритм проверки можно обобщить для k-мерных пространств, циклически перебирая измерения.

Этот метод особенно полезен для поиска значений в пределах простых границ для каждого признака в данных. Например, имея набор данных, включающий стаж работы сотрудников и размеры их заработной платы, можно отыскать всех сотрудников, проработавших в компании от двух до четырех лет и имеющих зарплату от 40 до 80 тысяч... и рекомендовать их на повышение[1]!

Этот поиск, реализованный в листинге 9.12, интерпретирует область поиска как прямоугольник с границами, параллельными осям признаков в наборе данных, что, в свою очередь, означает отсутствие зависимостей между признаками в наших границах. При более сложных условиях, смешивающих несколько признаков (например, зарплата ниже 15 тысяч за каждый год стажа в данной должности), границы области поиска были бы представлены отрезками общих линий, не параллельных осям.

В этом случае задача усложняется и нам может понадобиться применить что-то более сложное, например симплекс-метод[2], для поиска точек, удовлетворяющих условию попадания в диапазон.

Насколько эффективны два приведенных метода поиска областей? Как вы понимаете, это сильно зависит от самих областей. Оценим полный спектр вариантов, основываясь на двух крайних случаях:

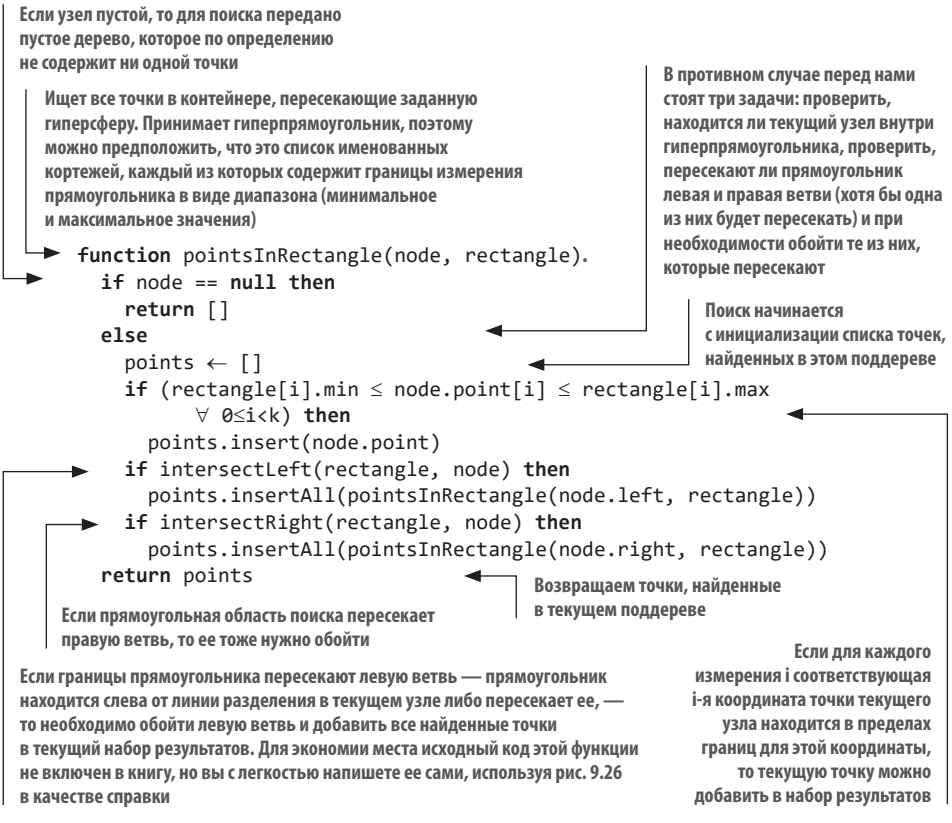
- для очень маленьких областей, пересекающих только одну ветвь, соответствующую листу, алгоритм просто будет следовать по пути к листу, исключая все остальные ветви, поэтому время выполнения будет равно $O(h)$, где h — высота дерева, то есть $O(\log(n))$ для сбалансированного дерева с n точками;
- когда область достаточно велика и пересекает все точки, алгоритму придется обойти все дерево и время выполнения будет равно $O(n)$.

---

1 В идеальном мире...

2 Симплекс-метод — это оригинальный алгоритм оптимизации. Он не связан с k-мерными деревьями и как таковой выходит за рамки этой главы, но представляет определенный интерес. Познакомиться с ним поближе можно здесь: https://en.wikipedia.org/wiki/Simplex_algorithm (https://ru.wikipedia.org/wiki/Симплекс-метод).

Листинг 9.12. Метод pointsInRectangle



Отсюда можно сделать заключение, что в худшем случае время выполнения равно O(n), даже если методы будут эффективно исключать ветви, когда это возможно.

9.3.7. Обзор всех методов

Как вы поняли, k-мерные деревья ускоряют поиск по всему набору данных по сравнению с полным перебором. В табл. 9.1 приводится краткая сводка по оценке эффективности методов, описанных в этой главе. Наихудшее время поиска (и удаления) ближайшего соседа по-прежнему является линейным (как и в методе полного перебора), но на практике амортизированная производительность на сбалансированных k-мерных деревьях немного или стабильно лучше. Немного — в низкоразмерных пространствах и стабильно — в среднеразмерных.

В высокоразмерных пространствах экспоненциальный член k становится доминирующим и делает нецелесообразной поддержку дополнительной сложности такой структуры данных.

Таблица 9.1. Операции с k-мерными деревьями и их производительность на сбалансированном k-мерном дереве с n

Операция	Время выполнения	Требуемый объем дополнительной памяти
`search`	O(log(*n*))	O(1)
`insert`	O(log(*n*))	O(1)
`remove`	$O(n^{1-1/k})^*$	O(1)
`findMin`	$O(n^{1-1/k})^*$	O(1)
`nearestNeighbor`	$O(2^k + \log(n))^*$	$O(m)^{**}$
`pointsInRegion`	O(*n*)	O(*n*)

* Амортизированное для k-мерных деревьев, хранящих k-мерные точки.

** Для поиска m ближайших соседей, когда m — константа, а не функция от n.

9.4. ОГРАНИЧЕНИЯ И ВОЗМОЖНЫЕ УЛУЧШЕНИЯ

Приступая к решению задачи поиска ближайшего склада в начале этой главы, мы с вами, по сути, не имели в арсенале средств ничего, кроме поиска методом полного перебора. Затем, предприняв несколько не самых удачных попыток, выяснили возможные ловушки и проблемы, стоящие на пути к решению такой задачи и, наконец, познакомились с k-мерными деревьями.

k-мерные деревья хороши тем, что обеспечивают значительное ускорение по сравнению с линейным поиском. И все же они имеют свои потенциальные проблемы, ниже я перечисляю их.

- Создать эффективную реализацию k-мерных деревьев очень непросто.
- k-мерные деревья не являются самобалансирующимися, поэтому лучше всего они работают, если созданы из стабильного набора точек, а количество вставок незначительно по сравнению с общим количеством элементов. К сожалению, в эпоху больших данных статические наборы данных не являются нормой.
- При работе с высокоразмерными пространствами k-мерные деревья становятся неэффективными. Как было показано в этой главе, время удаления и поиска ближайшего соседа находится в экспоненциальной зависимости от k, размерности набора данных и при достаточно больших значениях k все преимущества в производительности перед методом полного перебора могут сойти на нет. Также было показано, что поиск ближайших соседей работает эффективнее полного перебора при $n > 2^k$, поэтому, начиная с $k \approx 30$, необходимо иметь идеальный набор данных с миллиардами равномерно распределенных точек, чтобы k-мерное дерево превосходило по производительности метод полного перебора.

- *k*-мерные деревья плохо работают со страничной памятью, они неэффективны с точки зрения локальности ссылок, потому что точки хранятся в узлах дерева и не всегда в соседних областях памяти.
- *k*-мерные деревья хорошо справляются с поиском точек, но не способны работать с неточечными объектами, такими как фигуры и любые другие объекты с ненулевыми измерениями.

Неэффективность многомерных наборов данных обусловлена тем, что данные в них становятся очень разреженными. В то же время, выполняя обход *k*-мерного дерева во время поиска ближайшего соседа, есть возможность исключать из поиска ветви, не пересекающие гиперсферу с центром в целевой точке и с радиусом, равным минимальному найденному на данный момент расстоянию. Весьма вероятно, что эта сфера будет пересекать множество гиперкубов ветвей хотя бы в одном измерении.

Чтобы преодолеть эти ограничения, рекомендую попробовать применить несколько подходов. Их перечень ниже.

- Для выбора места разделения *k*-мерного пространства можно использовать разные эвристики:
  - не использовать линии разделения, проходящие через точки, а просто делить область на две сбалансированные половины (например, по количеству точек или по размеру подобластей, фактически выбирая среднее значение вместо медианы);
  - вместо циклического перебора измерений можно на каждом шаге выбирать измерение с наибольшим разбросом или дисперсией и сохранять выбор в узле дерева.
- Вместо хранения точек в узлах каждый узел мог бы описывать область пространства и поддерживать (прямо или косвенно) массив с фактическими элементами.
- Можно аппроксимировать поиск ближайшего соседа, например использовав *хеширование с учетом местоположения*.
- Можно поискать другие способы разделения *k*-мерного пространства, чтобы уменьшить разреженность.

Эвристики способны помочь в работе с некоторыми наборами данных, но в целом не решают проблему с высокоразмерными пространствами.

Метод аппроксимации не дает точного решения, но во многих случаях допустимо согласиться с неоптимальным результатом или вообще не иметь возможности определить идеальную метрику. Например, поразмышляйте о поиске документа, ближайшего к статье, или товара, ближайшего к выбранному для покупки, но отсутствующего в продаже. Мы пока отложим эту тему и в следующей главе углубимся в другой подход, основанный на SS-деревьях. А в главу 11 перенесем обсуждение практического применения поиска ближайшего соседа.

РЕЗЮМЕ

- Увеличение количества измерений в наборе данных обычно приводит к экспоненциальному увеличению сложности или необходимого объема памяти.
- Ограничить этот экспоненциальный всплеск можно, тщательно проектируя структуру данных, но его нельзя удалить полностью. Когда количество измерений слишком велико, очень трудно, если вообще возможно, поддерживать хорошую производительность.
- *k*-мерные деревья — это продвинутая структура данных, помогающая эффективнее обрабатывать пространственные запросы (поиск ближайших соседей и пересечений со сферическими или прямоугольными областями).
- *k*-мерные деревья прекрасно работают с низко- и среднеразмерными пространствами, но страдают от разреженности, характерной для высокоразмерных пространств.
- *k*-мерные деревья лучше подходят для работы со статическими наборами данных, потому что в этом случае есть возможность конструировать сбалансированные деревья, тогда как `insert` и `remove` не являются самобалансирующими операциями.

10 Деревья поиска по сходству: приближенный поиск ближайших соседей для выбора похожих изображений

В этой главе

- ✓ Обсуждение ограничений *k*-мерных деревьев.
- ✓ Выбор похожих изображений как пример практического применения метода ближайших соседей, когда *k*-мерных деревьев оказывается недостаточно.
- ✓ Знакомство с новой структурой данных, R-деревом.
- ✓ Знакомство с SS-деревьями, масштабируемым вариантом R-деревьев.
- ✓ Сравнение SS-деревьев с *k*-мерными деревьями.
- ✓ Приближенный поиск по сходству.

Эта глава структурирована несколько иначе, чем другие, просто потому что в ней продолжается обсуждение, начатое в главе 8, где была описана задача поиска в многомерных данных ближайших соседей некоторой обобщенной точки (возможно, не входящей в этот набор). В главе 9 состоялось знакомство с *k*-мерными деревьями — структурой данных, специально изобретенной для решения этой задачи.

В настоящее время *k*-мерные деревья считаются лучшим решением для индексации низко- и среднеразмерных наборов данных, целиком умещающихся в памяти. Но когда возникает необходимость обработки высокоразмерных данных или больших их наборов, не помещающихся в памяти, то *k*-мерных деревьев оказывается недостаточно и нужны более сложные структуры данных.

В этой главе мы сначала познакомимся с новой задачей, выходящей за рамки индексирования, а затем рассмотрим две новые структуры данных, R-деревья и SS-деревья, помогающие эффективно решать задачи из этой категории.

Приготовьтесь — это будет долгое путешествие (и длинная глава!) через самые сложные темы по сравнению с теми, что были рассмотрены до сих пор. Постараемся пройти этот путь постепенно: шаг за шагом, раздел за разделом, поэтому пусть вас не пугает размер этой главы!

10.1. ПРОДОЛЖАЕМ С ТОГО МЕСТА, НА КОТОРОМ ОСТАНОВИЛИСЬ

Вспомним, на чем мы остановились в предыдущей главе. Вы разрабатываете программное обеспечение для компании электронной коммерции, одной из его функций является поиск склада, продающего конкретный товар и ближайшего к любой точке на очень большой карте (см. рис. 9.4). Напомню также, что вам нужно обслуживать миллионы клиентов в день по всей стране, забирая товары с тысяч складов, опять же разбросанных на карте.

В разделе 8.2 было показано, что решение методом полного перебора, просматривающее весь список точек и сравнивающее каждую из них с целевой, не годится для практического применения и необходима совершенно иная структура данных, подходящая для многомерного индексирования. В главе 9 были введены в рассмотрение *k*-мерные деревья — значимая веха в индексировании многомерных данных. Они отлично вписались в решение задачи, использованной для примера в главах 8 и 9, согласно условиям которой необходимо было обрабатывать только двумерные данные. Единственная проблема, которая слегка портила картину, — наш набор данных был динамическим, поэтому операции вставки/удаления приводили к разбалансированию дерева. Но сохранялась возможность хотя бы время от времени перестраивать дерево (например, после изменения 1 % его элементов) и амортизировать стоимость операции, выполняя ее в фоновом процессе. При этом сохранялась старая версия дерева и либо приостанавливались вставка/удаление, либо повторно применялись эти операции к новому дереву после его создания и перевода в статус текущего.

В этом конкретном случае существовали обходные пути, но в других ситуациях может так не повезти. На самом деле *k*-мерные деревья имеют внутренние непреодолимые ограничения:

- *k*-мерные деревья не являются самобалансирующимися, поэтому лучше всего они работают, если созданы из стабильного набора точек, а количество вставок незначительно по сравнению с общим количеством элементов;
- проклятие размерности: при работе с высокоразмерными пространствами *k*-мерные деревья становятся неэффективными, потому что время поиска находится

в экспоненциальной зависимости от размерности набора данных. Для точек в k-мерном пространстве, когда $k \approx 30$, k-мерные деревья не дают никаких преимуществ перед полным перебором;

- k-мерные деревья плохо работают со страничной памятью; они неэффективны с точки зрения локальности ссылок, потому что точки хранятся в узлах дерева и не всегда в соседних областях памяти.

10.1.1. Новый более сложный пример

Чтобы показать практическую ситуацию, когда k-мерные деревья оказываются неэффективным решением, отложим в сторону задачу поиска склада и представим другой сценарий, в котором перенесемся на десять лет вперед. Ваша компания электронной коммерции выросла и теперь продает не только продукты, но также электронику и одежду. Близится 2010 год, и клиенты ожидают ценных рекомендаций при просмотре нашего каталога. Но, что еще более важно, отдел маркетинга компании надеется, что вы, как технический директор, обязательно поможете увеличить продажи, показывая клиентам предложения, которые им действительно нравятся.

Например, если клиент просматривает смартфоны (некогда самый популярный продукт в каталоге электроники!), то приложение должно показать ему больше смартфонов с аналогичным соотношением «цена/качество». Если покупательница просматривает коктейльное платье, она должна увидеть больше платьев, похожих на то, которое ей, возможно, нравится.

Эти две задачи выглядят (и отчасти это верно) совершенно разными, но обе сводятся к одной и той же ключевой цели: по заданному товару, обладающему определенными характеристиками, найти один или несколько похожих товаров. Очевидно, что способы извлечения этих характеристик для потребительской электроники и платьев сильно различаются!

Давайте сосредоточимся на платьях. Как показано на рис. 10.1, задача сводится к тому, чтобы, имея изображение платья, найти в каталоге другие товары, похожие на него, — это очень непростая задача даже в наши дни!

Способ извлечения признаков из изображений полностью изменился за последние десять лет. В 2009 году из изображений извлекались края, углы и другие геометрические элементы с использованием десятков алгоритмов, специализированных для каждого конкретного признака, а затем вручную (в буквальном смысле) создавались признаки более высокого уровня.

В настоящее время эта задача решается с применением глубокого обучения, когда на большом наборе данных обучается CNN[1] и затем применяется ко всем изображениям в каталоге для получения их векторов признаков.

---

[1] Convolutional Neural Network (сверточная нейронная сеть) — разновидность нейронных сетей, особенно хорошо подходящая для обработки изображений.

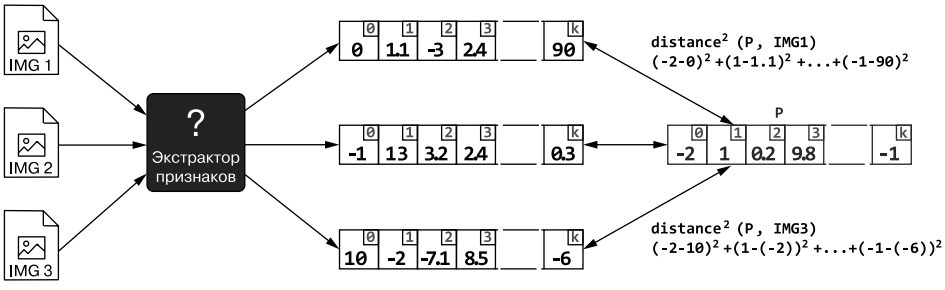


Рис. 10.1. Извлечение характеристических признаков из набора данных с изображениями. Каждое изображение преобразуется в вектор признаков (посредством экстрактора признаков, который мы будем считать «черным ящиком», потому что здесь нас не интересует алгоритм создания этих векторов). Затем, чтобы найти товар P, вектор признаков P сравнивается с векторами признаков других изображений путем вычисления расстояний между ними на основе некоторой метрики. (В данном примере используется евклидово расстояние. Обратите внимание, что при поиске минимального из этих евклидовых расстояний иногда можно вычислять квадраты расстояний и отказаться от операции извлечения квадратного корня)

Однако и в наши дни после получения векторов признаков встает тот же вопрос: какой алгоритм применить для эффективного поиска векторов, наиболее похожих на заданный?

Это та же самая задача, которую мы рассмотрели в главе 8 на примере двумерных данных, только на этот раз она затрагивает огромный набор данных (с десятками тысяч изображений/векторов признаков) и кортежи, содержащие сотни признаков.

В отличие от извлечения признаков алгоритмы поиска несильно изменились за последние десять лет, а структуры данных, представленные в этой главе и изобретенные в период от начала 1990-х до начала 2000-х, по-прежнему являются предпочтительным выбором для эффективного поиска в векторном пространстве.

10.1.2. Преодоление недостатков k-мерных деревьев

В главе 9 было упомянуто несколько возможных решений задач, обсуждавшихся в предыдущем разделе:

- вместо линий разделения, проходящих через точки в наборе данных, области можно делить на две сбалансированные половины по количеству точек или размерам подобластей;
- вместо циклического перебора измерений можно на каждом шаге выбирать измерение с наибольшим разбросом или дисперсией и сохранять сделанный выбор в каждом узле дерева;
- вместо точек в узлах можно хранить описания областей и связывать их (прямо или косвенно) с массивами, содержащими фактические элементы.

Перечисленные решения лежат в основе структур данных, которые мы обсудим в этой главе: R- и SS-деревьев.

10.2. R-ДЕРЕВО

Первой обсудим структуру, являющуюся дальнейшим развитием *k*-мерных деревьев, которая называется R-деревом. Не будем вдаваться в детали ее реализации, но рассмотрим идею этого решения, принцип и механику его действия.

R-деревья были представлены в 1984 году Антонином Гуттманом (Antonin Guttman) в статье *R-Trees. A Dynamic Index Structure For Spatial Searching*.

Основой для них послужили B-деревья[1] — сбалансированные деревья с иерархической структурой. В частности, Гуттман использовал в качестве отправной точки деревья B+, в которых только листовые узлы содержат данные, а внутренние узлы содержат только ключи и служат для иерархического разделения данных.

10.2.1. Шаг назад: знакомство с B-деревьями

На рис. 10.2 показан пример B-дерева, точнее, дерева B⁺. Эти структуры данных предназначались для индексации одномерных данных путем разделения их на страницы[2], что обеспечивало эффективное хранение на диске и быстрый поиск (за счет минимизации количества загружаемых страниц и, следовательно, количества обращений к диску).

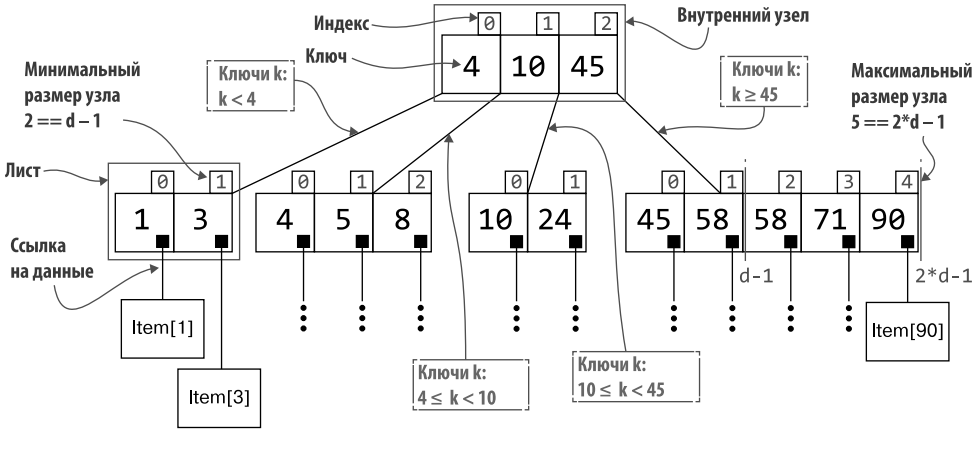


Рис. 10.2. Организация дерева B⁺. В примере показано дерево B⁺ с коэффициентом ветвления d == 3

[1] B-дерево — это самобалансирующееся дерево, оптимизированное для эффективного хранения больших наборов данных на диске.

[2] Страницы памяти, или просто страницы.

Каждый узел (и внутренние узлы, и листья) в B-дереве содержат от $d - 1$ до $2d - 1$ ключей, где d — фиксированный параметр для каждого дерева, его коэффициент ветвления[1], то есть количество (минимальное в данном случае) потомков каждого узла. Единственное исключение — корень, который может содержать меньше ключей, чем $d - 1$. Ключи хранятся в упорядоченном списке, это необходимо для быстрого (логарифмического) поиска. На самом деле каждый внутренний узел с m ключами, $k_0, k_1 \ldots k_{m-1}$, $d - 1 \le m < 2d - 1$, имеет ровно $m + 1$ потомков, $C_0, C_1 \ldots C_{m-1}, C_m$ таких, что $k < k_0$ для каждого ключа k в поддереве с корнем в C_0; $k_0 \le k < k_1$ для каждого ключа k в поддереве с корнем в C_1; и т. д.

Ключи и элементы в B-дереве хранятся в узлах, каждый ключ/элемент хранится ровно один раз в одном узле, и все дерево хранит ровно n ключей, если набор данных содержит n элементов. В дереве B^+ внутренние узлы хранят только ключи, и только листья хранят пары, каждая из которых содержит ключ и ссылки на элементы. Это означает, что дерево B^+, насчитывающее n элементов, имеет n листьев, а ключи из внутренних узлов хранятся также во всех их потомках (например, как показано на рис. 10.2, ключи 4, 10 и 45 хранятся также в листьях).

Хранение ссылок на элементы в листьях вместо размещения самих элементов в дереве служит двойной цели:

- узлы получаются проще, и их проще размещать в памяти и удалять;
- такой подход позволяет хранить все элементы в массиве или в непрерывном блоке памяти, обеспечивая лучшую локальность соседних элементов.

Когда подобные деревья задействуются для хранения огромных коллекций больших элементов, эти свойства позволяют эффективно использовать подкачку памяти. Дерево с маленькими узлами с большей вероятностью уместится в памяти, тогда как элементы данных вполне могут храниться на диске и загружаться по мере необходимости. Поскольку также высока вероятность, что после обращения к элементу X приложению потребуется обратиться к одному из соседних элементов, то загружая в память весь лист B-дерева, содержащий X, можно минимизировать количество операций чтения с диска.

Неудивительно, что с момента изобретения[2] B-деревья легли в основу многих механизмов баз данных SQL и даже в наши дни именно эта структура данных остается предпочтительной для хранения индексов.

10.2.2. От B-дерева к R-дереву

R-деревья распространяют основные идеи, заложенные в основу деревьев B^+, на многомерный случай. Если для одномерных данных каждый узел соответствует интервалу (диапазону от самого левого до самого правого ключа в его поддереве,

[1] Вы наверняка помните, что мы уже обсуждали фактор ветвления в главе 2, когда знакомились с d-ичными кучами.

[2] См., например: https://sqlity.net/en/2445/b-plus-tree/.

они же, в свою очередь, являются его минимальным и максимальным ключами), то в R-деревьях каждый узел N охватывает прямоугольник (или гиперпрямоугольник в общем случае), углы которого определяются минимумом и максимумом каждой координаты по всем точкам в поддереве с корнем в N.

Подобно B-деревьям, R-деревья тоже являются параметрическими. Вместо фактора ветвления d, определяющего минимальное количество записей на узел, R-деревья требуют, чтобы их клиенты определили два параметра при создании:

- M — максимальное количество записей в узле; это значение обычно выбирается таким, чтобы полный узел умещался в странице памяти;
- m — такое, что $m \leq M/2$, минимальное количество записей в узле. Как будет показано далее, этот параметр косвенно определяет минимальную высоту дерева.

Учитывая значения этих двух параметров, R-деревья следуют нескольким инвариантам.

1. Каждый лист содержит от m до M точек (за исключением корня, который может иметь меньше m точек).
2. С каждым листовым узлом L связан гиперпрямоугольник R_L — наименьший прямоугольник, содержащий все точки листа.
3. Каждый внутренний узел имеет от m до M потомков (за исключением корня, у которого может быть меньше m потомков).
4. С каждым внутренним узлом N связан ограничивающий (гипер-)прямоугольник R_N — наименьший прямоугольник, ребра которого параллельны декартовым осям, содержащий все ограничивающие прямоугольники N потомков.
5. Корневой узел имеет не менее двух потомков, если только он не является листом.
6. Все листья находятся на одном уровне.

Свойство 6 говорит о сбалансированности R-деревьев, а из свойств 1 и 3 вытекает, что максимальная высота R-дерева, содержащего n точек, равна $\log_m(n)$.

Если при вставке в каком-либо узле на пути от корня к листу, содержащему новую точку, количество записей превысит величину M, то его придется разделить и создать два узла, каждый из которых содержит половину элементов.

Если при удалении в каком-либо узле количество записей станет меньше, чем m, то его придется объединить с одним из соседних узлов на том же уровне.

Инварианты 2 и 4 требуют дополнительных усилий для их поддержки, но эти ограничивающие прямоугольники, определенные для каждого узла, необходимы для быстрого поиска по дереву.

Прежде чем описывать работу методов поиска, рассмотрим подробнее пример R-дерева на рис. 10.3 и 10.4. Будем придерживаться двумерного случая, потому что его проще визуализировать, но, как всегда, необходимо понимать, что настоящие деревья могут содержать трех-, четырех- и даже 100-мерные точки.

Если сравнить между собой рис. 10.3 и 9.6, показывающие организацию одного и того же набора данных в *k*-мерном дереве, сразу становится очевидным, насколько эти два разбиения разные:

- R-деревья создают прямоугольные области на декартовой плоскости, а *k*-мерные деревья разбивают плоскость по линиям;
- *k*-мерные деревья делят области, чередуя измерения, а R-деревья перемещаются по измерениям не циклически. Прямоугольники на каждом уровне могут разбиваться по любому или даже по всем измерениям одновременно;
- ограничивающие прямоугольники в разных поддеревьях могут перекрываться; точно так же могут перекрываться ограничивающие прямоугольники потомков, имеющих одного и того же родителя. Однако, и это очень важно, ни один прямоугольник не выходит за пределы ограничивающей рамки родителя;
- каждый внутренний узел определяет так называемую *ограничивающую оболочку*, которая для R-деревьев является наименьшим прямоугольником, содержащим все ограничивающие оболочки потомков узла.

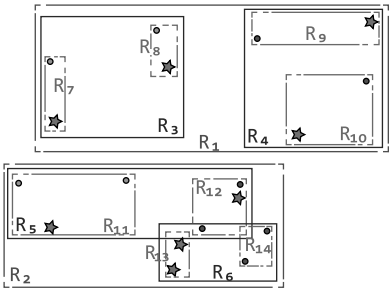


Рис. 10.3. Представление (допустимого) R-дерева в декартовой плоскости для карты городов, изображенной на рис. 9.4 (названия городов опущены во избежание путаницы). Это R-дерево содержит 12 ограничивающих прямоугольников от R_1 до R_{12}, организованных в иерархическую структуру. Обратите внимание, что прямоугольники могут перекрываться и перекрываются, как показано в нижней половине

На рис. 10.4 демонстрируется, как эти свойства транслируются в древовидную структуру данных; здесь разница с *k*-мерными деревьями еще более очевидна!

Каждый внутренний узел — это список прямоугольников (от *m* до *M*, как уже упоминалось), а листья — это списки (опять же от *m* до *M*) точек. Каждый прямоугольник фактически определяется своими потомками и может быть описан итеративно в терминах дочерних элементов. Из практических соображений, таких как сокращение времени выполнения методов поиска, обычно принято сохранять ограничивающую рамку для каждого прямоугольника.

Поскольку прямоугольники могут быть параллельны только декартовым осям, они определяются двумя вершинами: двумя кортежами с *k* координатами, один из которых содержит минимальные значения каждой координаты, другой — максимальные.

Обратите внимание, что, в отличие от *k*-мерных деревьев, R-дерево может работать с объектами ненулевых размеров, просто рассматривая их ограничивающие рамки как частные случаи прямоугольников, это показано на рис. 10.5.

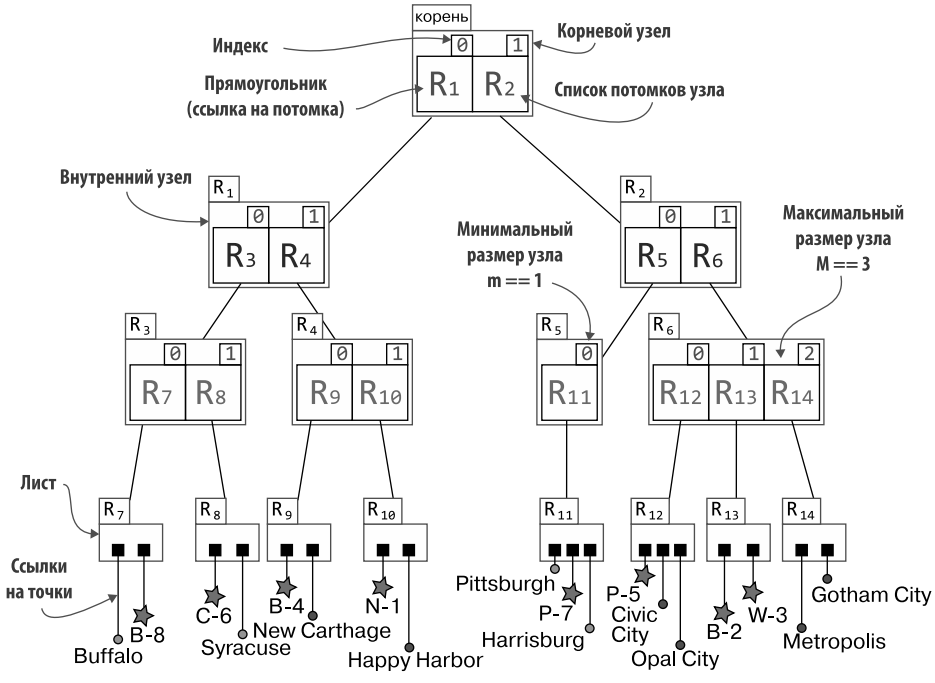


Рис. 10.4. Древовидное представление R-дерева, изображенного на рис. 10.3. Это R-дерево имеет следующие параметры: m == 1 и M == 3. Внутренние узлы содержат только ограничивающие рамки, а листья — фактические точки (или, вообще говоря, k-мерные элементы). В оставшейся части главы будем использовать более компактное представление узлов, изображая только списки дочерних элементов

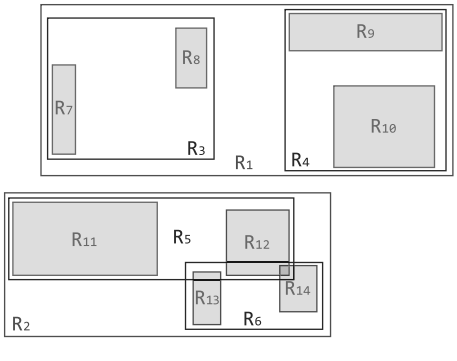


Рис. 10.5. Элементы R-деревьев, помимо точек, тоже могут быть прямоугольниками или объектами с ненулевыми размерами. В этом примере объекты с R_7 по R_{14} являются элементами дерева, а объекты с R_3 по R_6 — листьями

10.2.3. Вставка точек в R-дерево

Теперь у вас может возникнуть обоснованный вопрос: как из исходного набора данных получить R-дерево, изображенное на рис. 10.5. Ведь я просто показал его и предложил вам принять это как данность.

Операция вставки в R-дерево аналогична операции вставки в B-дерево, и обе они имеют много общих шагов с SS-деревьями, поэтому не будем тратить силы и время на их подробное описание здесь.

Но в общих чертах отмечу: чтобы вставить новую точку, нужно выполнить следующие шаги.

1. Найти лист, куда следует поместить новую точку *P*. Возможны три случая:
   а) *P* лежит точно внутри *R* — одного из листовых прямоугольников. В таком случае достаточно просто добавить *P* в *R* и перейти к следующему шагу;
   б) *P* лежит в области, где перекрываются два и более ограничивающих прямоугольника листьев, как, например, точка на рис. 10.6, находящаяся в области пересечения прямоугольников R_{12} и R_{14}. В этом случае нужно решить, куда добавить *P*; эвристика, используемая для принятия таких решений, будет определять форму дерева (одним из примеров эвристик может служить просто добавление точки в прямоугольник с меньшим количеством элементов);
   в) *P* лежит за пределами всех прямоугольников на уровне листьев. В таком случае нужно найти ближайший лист *L* и добавить *P* в него (опять же для принятия решения можно использовать более сложную эвристику, чем простое евклидово расстояние).

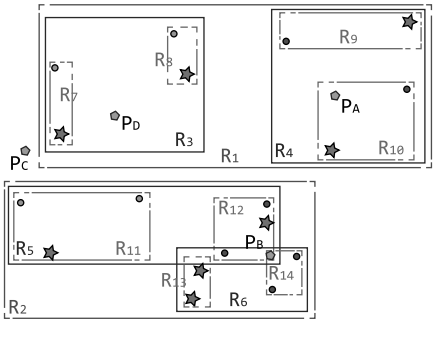


Рис. 10.6. Выбор прямоугольника листа R-дерева, в который должна быть добавлена точка: новая точка может находиться внутри прямоугольника листа (PA), на пересечении двух или более прямоугольников листа (PB) или вне любого листа (PC и PD)

2. Добавить точку в прямоугольник листа *R*, а потом проверить, сколько точек он содержит:
   а) если после добавления новой точки лист имеет не более *M* точек, то на этом выполнение операции вставки заканчивается;
   б) в противном случае нужно разбить *R* на два новых прямоугольника, R_1 и R_2, и перейти к шагу 3.
3. Удалить *R* из родительского R_P и добавить в R_P новые листья R_1 и R_2. Если после этого количество потомков в R_P превысило величину *M*, его нужно разделить и рекурсивно повторить этот шаг:
   а) если *R* был корнем, то его, очевидно, нельзя удалить из родителя. В таком случае просто создается новый корень и в него добавляются два потомка, R_1 и R_2.

Чтобы завершить описанный здесь алгоритм вставки, нужно определить несколько эвристик, помогающих решить проблему неоднозначности при выборе одного из перекрывающихся прямоугольников, и, что еще более важно, то, как разбивать прямоугольник в шагах 2 и 3.

Этот выбор вместе с эвристикой выбора поддерева для вставки определяет поведение и форму R-дерева (не говоря уже о его производительности).

За прошедшие годы было изучено несколько эвристик, каждая из которых призвана оптимизировать один или несколько способов использования дерева. Эвристика разделения может быть особенно сложной для внутренних узлов, потому что разделяются не просто точки, а k-мерные фигуры. На рис. 10.7 показано, как легко упрощенная эвристика может привести к неэффективному разделению.

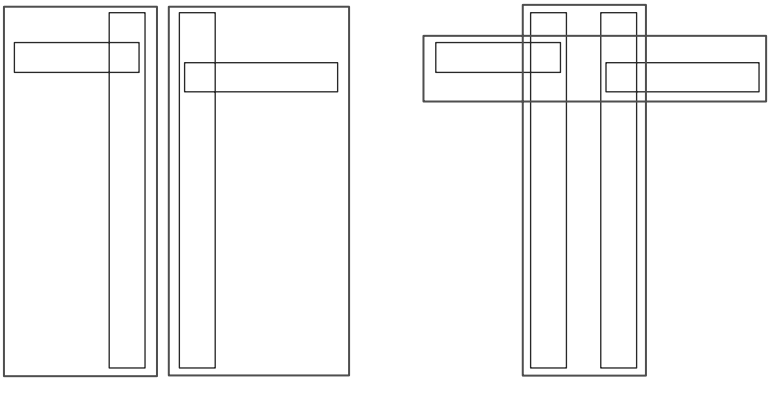


Рис. 10.7. Пример плохого и хорошего разбиения прямоугольника внутреннего узла, взятый из оригинальной статьи Антонина Гуттмана

Углубленное обсуждение эвристик выходит за рамки этого раздела, поэтому за дополнительной информацией я отсылаю любознательного читателя к оригинальной статье Антонина Гуттмана. На данный момент, однако, уже можно показать, что сложность обработки гиперпрямоугольников и получения хороших разбиений (и слияний после удалений) является одной из основных причин, которые привели к появлению SS-деревьев.

10.2.4. Поиск

Поиск точки или ближайшего соседа (Nearest Neighbor, NN) точки в R-деревьях очень похож на поиск в k-мерных деревьях. Нужно пройти по дереву, исключая ветви, которые не могут содержать точки или, в случае поиска ближайшего соседа, заведомо дальше текущего минимального расстояния.

На рис. 10.8 показан пример (неудачного) поиска точки в нашем примере R-дерева. Напомню, что неудачный поиск — это первый шаг для вставки новой точки, выполнив который можно найти прямоугольник (или прямоугольники в данном случае), куда следует добавить новую точку.

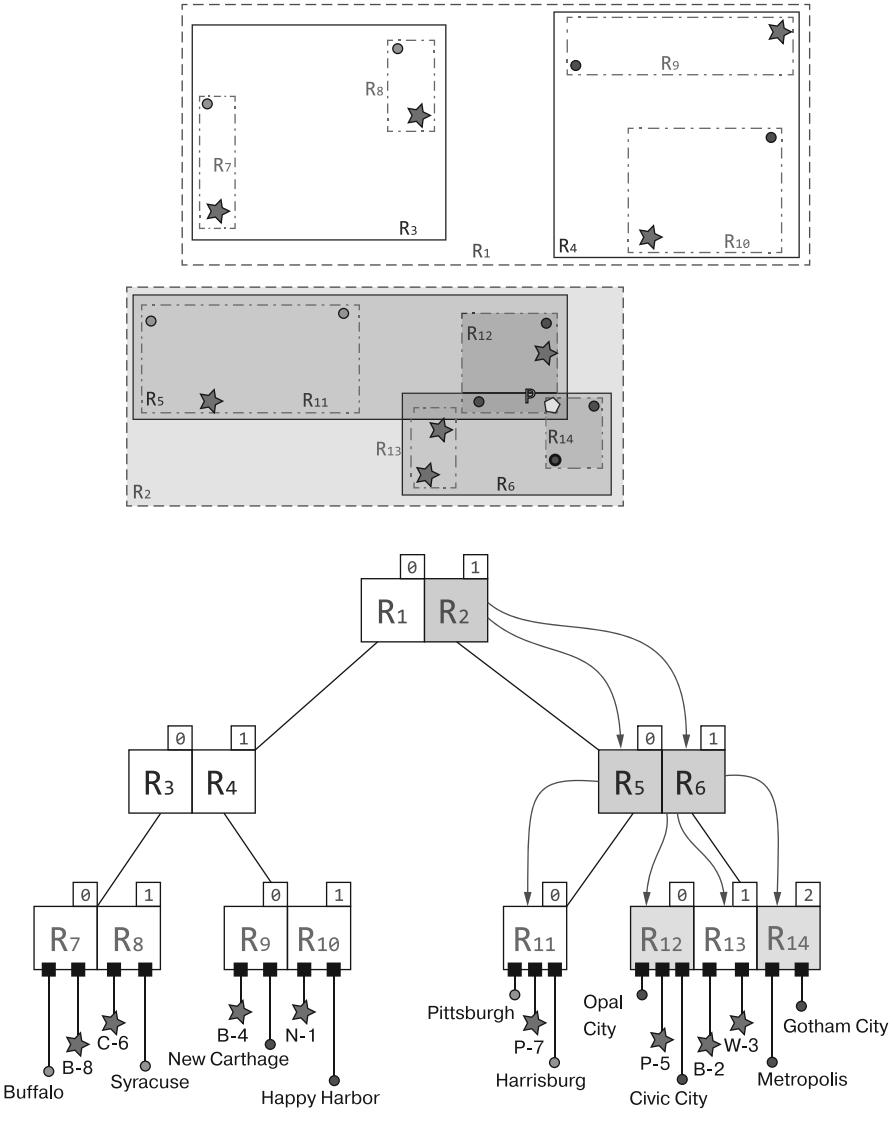


Рис. 10.8. Неудачный поиск в R-дереве, изображенном на рис. 10.4 и 10.5. Путь поиска выделен как в декартовом, так и в древовидном представлении, а изогнутые стрелки показывают ветви в дереве, по которым осуществляется обход. Обратите внимание на более компактное представление дерева по сравнению с рис. 10.5

Поиск начинается с корня, сравниваются координаты точки P с границами каждого прямоугольника, R_1 и R_2; P может находиться только в пределах R_2, соответственно, это единственная ветвь, по которой нужно пройти.

На следующем шаге выполняется обход потомков узла R_2 — R_5 и R_6. И тот и другой узлы могут содержать P, поэтому нужно обойти обе ветви на этом уровне (как показано двумя изогнутыми стрелками, исходящими из R_2 в нижней половине на рис. 10.8).

Это означает, что нужно обойти потомков обоих прямоугольников R_5 и R_6 и проверить узлы от R_{11} до R_{14}. Из них только R_{12} и R_{14} могут содержать P, соответственно, именно в этих прямоугольниках точки проверяются на последнем шаге. Ни один из них не содержит P, поэтому метод поиска может вернуть `false` и при необходимости два листовых прямоугольника, куда можно вставить P.

Поиск ближайшего соседа работает аналогично, но вместо проверки принадлежности точки каждому прямоугольнику он сохраняет расстояние до текущего ближайшего соседа и проверяет, ближе ли каждый прямоугольник к нему (иначе он может исключить его из поиска). Это чем-то напоминает поиск прямоугольной области в k-мерных деревьях, как описано в подразделе 9.3.6.

Не будем углубляться в поиск ближайших соседей в R-деревьях. Теперь, получив общее представление об этой структуре данных, можно переходить к следующему этапу ее эволюции — SS-деревьям.

Стоит также отметить, что R-деревья не гарантируют хорошую производительность в худшем случае, но на практике они часто работают лучше, чем k-мерные деревья, поэтому долгое время они считались стандартным де-факто решением для задач поиска по сходству и индексации многомерных наборов данных.

10.3. ДЕРЕВЬЯ ПОИСКА ПО СХОДСТВУ

В разделе 10.2 были рассмотрены некоторые ключевые свойства R-деревьев, влияющие на их форму и производительность. Вспомним их:

- эвристика разделения;
- критерии выбора поддерева для добавления новых точек (если имеются перекрытия);
- метрика расстояния.

Для R-деревьев предполагалось, что в роли ограничивающих оболочек для узлов используются ориентированные прямоугольники — гиперпрямоугольники, параллельные декартовым осям. Если снять это ограничение, то форма ограничивающей оболочки станет четвертым свойством более общего класса деревьев поиска по сходству.

И действительно, по своей сути основное отличие R-деревьев от SS-деревьев в их самых базовых вариантах заключается в форме ограничивающих оболочек. Как показано на рис. 10.9, в этом варианте (построенном на основе R-деревьев) вместо прямоугольников используются сферы.

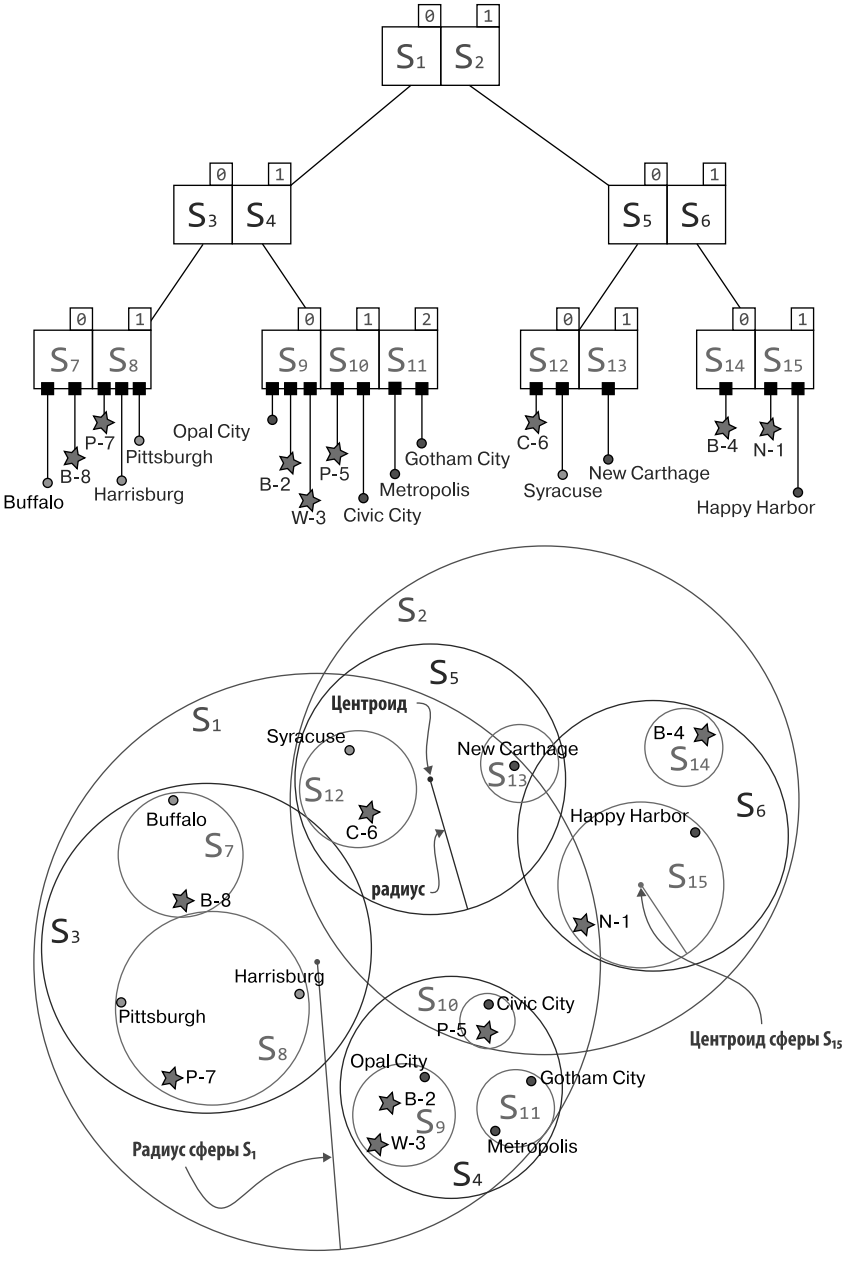


Рис. 10.9. Представление возможного SS-дерева, охватывающего тот же набор данных, что и на рис. 10.4 и 10.5, с параметрами m == 1 и M == 3. Древовидная структура похожа на R-дерево. Чтобы избежать путаницы, здесь изображены только центроиды и радиусы нескольких сфер. Для изображения дерева используется компактное представление (как показано на рис. 10.5 и 10.8)

Это отличие может показаться несущественным, но имеются убедительные теоретические и практические доказательства того, что использование сфер уменьшает среднее количество листьев, проверяемых при поиске по сходству (ближайших соседей или областей). Мы обсудим этот момент более подробно в подразделе 10.5.1.

Итак, каждый внутренний узел N — это сфера, определяемая центром и радиусом. Эти два свойства однозначно и полностью задаются потомками N. Центр N фактически является центроидом потомков N[1], а радиус — максимальным расстоянием между центроидом и точками N.

Честно говоря, сказав, что единственная разница между R-деревьями и SS-деревьями заключается в форме ограничивающих оболочек, я несколько погрешил против истины. Выбор другой формы ограничивающих оболочек заставляет также принять другую эвристику разделения. В SS-деревьях вместо того, чтобы пытаться уменьшить перекрытие сфер при разделении, принято стремиться уменьшить дисперсию каждого из вновь созданных узлов. Поэтому исходная эвристика разделения выбирает измерение с наибольшей дисперсией, а затем разбивает отсортированный список дочерних элементов, чтобы уменьшить дисперсию по этому измерению (более подробно мы поговорим об этом в подразделе 10.3.2, когда будем обсуждать операцию вставки).

Подобно R-деревьям, SS-деревья имеют два параметра, m и M, — минимальное и максимальное количество дочерних элементов, которые может иметь каждый узел (кроме корня).

И подобно тому, как это происходит в R-деревьях, ограничивающие оболочки в SS-дереве могут перекрываться. Чтобы уменьшить перекрытие, некоторые варианты, такие как *деревья SS*$^+$, вводят пятое свойство (также используемое в вариантах R-деревьев, таких как *деревья R**), еще одну эвристику, используемую при вставке, которая управляет основными изменениями при реструктуризации дерева. Мы еще обратимся к деревьям SS$^+$ далее в этой главе, а пока сосредоточимся на реализации простых SS-деревьев.

На первом шаге к псевдореализации нашей структуры данных, как всегда, определим моделирующий ее псевдокласс. В этом случае для моделирования SS-дерева понадобится класс, представляющий узлы дерева. Чтобы обеспечить доступ к SS-дереву после его построения через узлы, достаточно иметь указатель на его корень. Для удобства, как показано в листинге 10.1, мы включили эту ссылку и значения параметров m и M в класс `SsTree`, так же как и размерность k каждой записи данных, и будем предполагать, что все эти значения доступны в каждом узле дерева.

Как уже известно, SS-деревья (как и R-деревья) имеют узлы двух разных типов, листья и внутренние узлы, которые различаются структурой и поведением. Первые

[1] Центроид определяется как центр масс набора точек, координатами которого являются взвешенные суммы координат точек. Если N — лист, его центр — это центр тяжести точек, принадлежащих N; если N — внутренний узел, то его центром является центр масс центроидов дочерних узлов.

хранят k-мерные кортежи (ссылки на точки в нашем наборе данных), а вторые — только ссылки на своих потомков (которые тоже являются узлами дерева).

Чтобы максимально упростить реализацию и сделать ее независимой от языка программирования, будем хранить в каждом узле два массива — массив потомков и массив точек, а различать листья и внутренние узлы будем с помощью логического флага. Массив потомков будет пустым в листьях, а массив точек — во внутренних узлах.

Листинг 10.1. Классы SsTree и SsNode

```
class SsNode
  #type tuple(k)
  centroid
  #type float
  radius

  #type SsNode[]
  children

  #type tuple(k)[]
  points

  #type boolean
  Leaf

  function SsNode(leaf, points=[], children=[])

class SsTree
  #type SsNode
  root
  #type integer
  m
  #type integer
  M
  #type integer
  k

  function SsTree(k, m, M)
```

Обратите внимание, что на рис. 10.9 узлы дерева представлены в виде списка сфер, каждая из которых имеет ссылку на потомка. Можно было бы добавить тип `SsSphere` и сохранить ссылку на единственный дочерний узел каждой сферы в поле этого нового типа. Однако это не лучшее решение, оно привело бы к дублированию данных (потому что тогда `SsNode` и `SsSphere` содержали бы поля, определяющие центроиды и радиусы), и создало бы лишний уровень косвенности. Просто имейте в виду, что на диаграммах SS-деревьев, которые приводятся на этих страницах, компоненты узлов дерева на самом деле являются их потомками.

Одной из эффективных альтернатив трансляции этих рассуждений в код в объектно-ориентированном программировании является использование наследования,

когда определяется общий абстрактный класс (класс, из которого нельзя создавать экземпляры) или интерфейс, а также два производных класса (один для листьев и другой для внутренних узлов), имеющих общие данные и поведение (определяемые базовым абстрактным классом), но реализованных по-разному. В листинге 10.2 показана возможная реализация этого шаблона на псевдокоде.

Листинг 10.2. Альтернативная реализация SsNode: SsNodeOO

```
abstract class SsNodeOO
  #type tuple(k)
  centroid
  #type float
    radius

class SsInnerNode: SsNodeOO
  #type SsNode[]
  children
  function SsInnerNode(children=[])

class SsLeaf: SsNodeOO
  #type tuple(k)[]
  points
  function SsLeaf(points=[])
```

Реализация с использованием наследования может привести к некоторому дублированию кода и требует дополнительных усилий, чтобы разобраться в ней. Но она дает, пожалуй, более чистое решение, устраняя логику выбора типа узла, которая в противном случае потребовалась бы в каждом методе класса.

Мы не будем использовать этот пример в оставшейся части главы, но усердный читатель может применить его в качестве отправной точки для экспериментов с реализацией SS-деревьев на основе этого шаблона.

10.3.1. Поиск в SS-дереве

Теперь можно приступать к описанию методов класса `SsNode`. Естественнее было бы начать с операции вставки (в конце концов, прежде чем выполнять поиск в дереве, его нужно построить), но в то же время для многих древовидных структур данных первым шагом в операции вставки (или удаления) является поиск узла, куда следует вставить новый элемент.

Следовательно, прежде чем вставить новый элемент, нам понадобится вызвать метод поиска (имеется в виду *точный поиск элемента*). Далее вы увидите, что этот шаг в методе вставки немного отличается от обычного поиска, и все же понять алгоритм вставки будет проще после знакомства с особенностями обхода SS-дерева.

На рис. 10.10 и 10.11 показаны этапы вызова метода поиска `search` в нашем примере SS-дерева. Справедливости ради отмечу, что SS-дерево, которое используется

в оставшейся части главы, получено из дерева, изображенного на рис. 10.9. Обратите внимание, что некоторые точки (оранжевые звезды) немного сместились и были убраны все метки точек — их заменили буквы от A до W. Это сделано, чтобы не загромождать диаграммы и сделать их более ясными. По той же причине мы будем обозначать точку для поиска/вставки/удаления в этом и следующих разделах как *Z* (чтобы избежать конфликтов с точками, уже имеющимися в дереве).

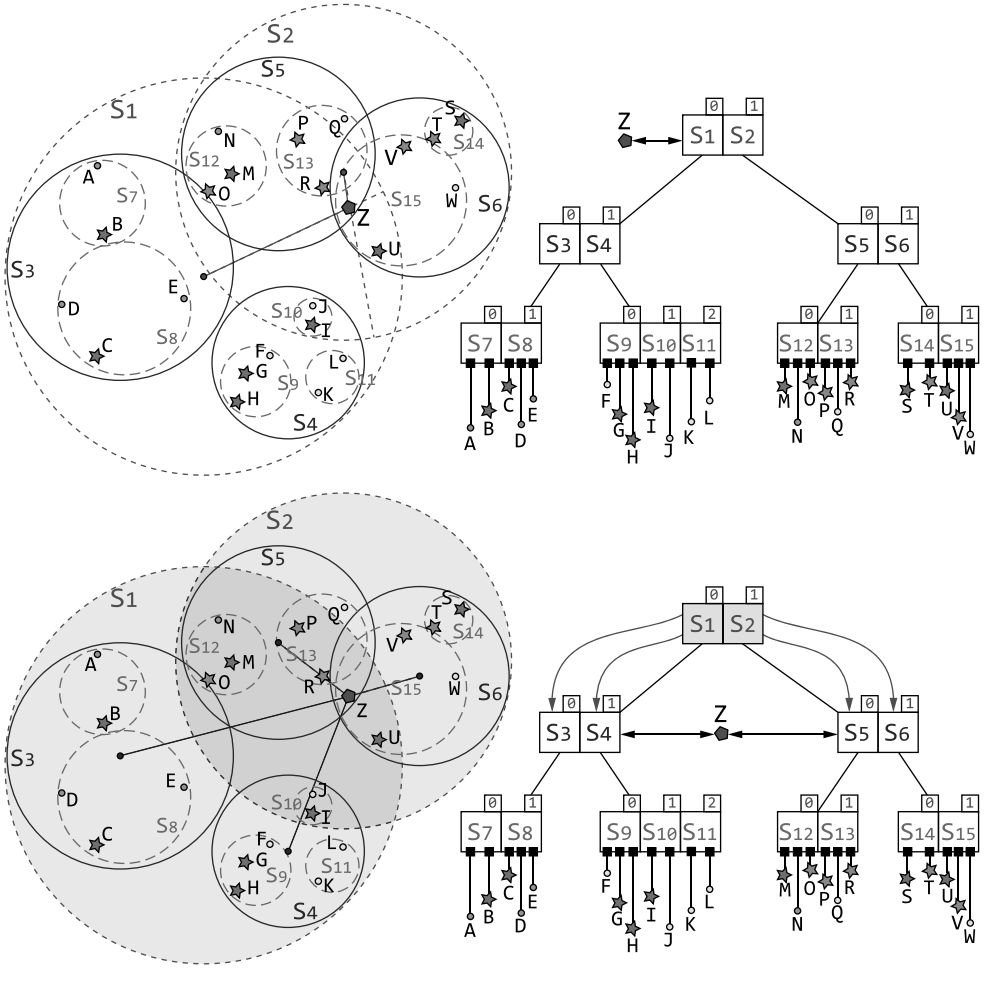


Рис. 10.10. Поиск в SS-дереве: первые несколько шагов поиска точки Z. Представленное здесь SS-дерево получено из дерева на рис. 10.9 с небольшими изменениями: чтобы не загромождать диаграммы, названия элементов были заменены буквами от A до W. *Вверху:* первый шаг поиска — сравнение Z со сферами в корне дерева: для каждой из них вычисляется расстояние от Z до центроида и сравнивается с радиусом сферы. *Внизу:* поскольку Z попадает в сферы S_1 и S_2, нужно выполнить обход обеих ветвей и проверить сферы от S3 до S6 на попадание в них точки Z

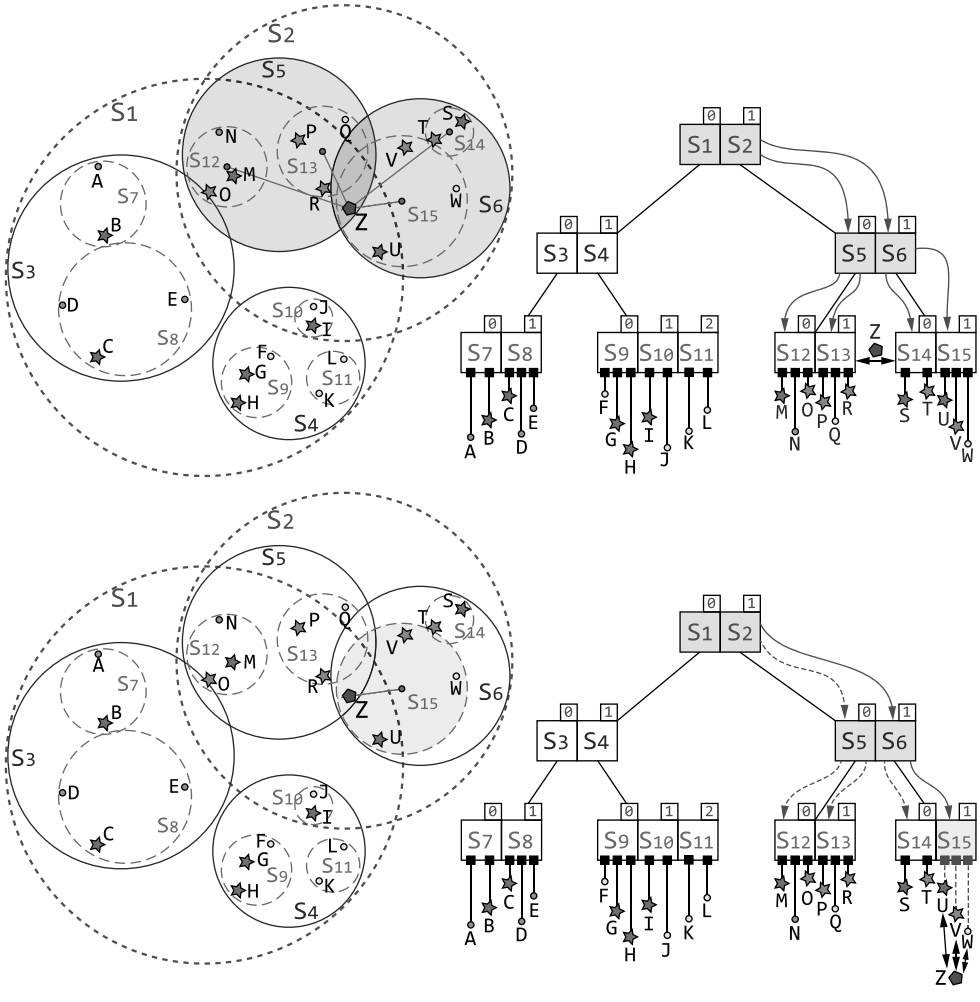


Рис. 10.11. Поиск в SS-дереве. Продолжаются действия с рис. 10.10: выполняется обход дерева до листьев. На каждом шаге выделены сферы, чьи дочерние элементы обходятся в данный момент (другими словами, на каждом шаге объединение выделенных сфер представляет наименьшую область, где может лежать искомая точка)

Продолжая пример с набором изображений в каталоге товаров, предположим, что теперь нужно проверить, находится ли конкретное изображение *Z* в нашем наборе данных. Один из способов сделать это — сравнить *Z* со всеми изображениями в наборе данных. Для сравнения двух изображений может потребоваться некоторое время (особенно если, например, все изображения имеют одинаковый размер и нет возможности быстро проверить любое другое тривиальное свойство изображения, чтобы исключить явно разные пары). Напомню, что набор данных, предположительно, содержит десятки тысяч изображений, соответственно, пойдя по этому пути, мы

должны быть готовы сделать длительный перерыв на кофе (или, в зависимости от быстродействия оборудования, оставить машину работать на ночь).

Конечно же, вы, уважаемый читатель, наверное, уже усвоили, что не нужно отчаиваться, потому что всегда находится более удачная альтернатива!

И действительно, как упоминалось в начале главы, можно создать набор векторов признаков для изображений в рассматриваемом наборе данных, получить вектор признаков для Z — назовем его F_Z — и выполнить поиск в пространстве векторов признаков вместо прямого поиска в наборе изображений.

Однако, сравнение F_Z с десятками или сотнями тысяч других векторов тоже может оказаться продолжительной и дорогостоящей процедурой с точки зрения времени, памяти и количества обращений к диску.

Если каждая страница памяти, хранящаяся на диске, может содержать M векторов признаков, то придется выполнить n / M обращений к диску и прочитать $n \times k$ значений с плавающей точкой.

Именно здесь на помощь приходит SS-дерево. Используя SS-дерево, содержащее до M элементов в каждом узле и по крайней мере с $m \leq M / 2$, можно уменьшить количество страниц, загружаемых с диска, до[1] $2 \times \log_M(n)$, а количество читаемых значений с плавающей точкой до ~ $k \times M \times \log_M(n)$.

В листинге 10.3 показан псевдокод метода `search` SS-дерева. Разбираясь с ним, можно следовать шагам, показанным на рис. 10.10 и 10.11. Первоначально `node` будет ссылаться на корень дерева, а не на лист; затем произойдет переход непосредственно к строке 7 и начнется циклический перебор дочерних узлов, в данном случае S_1 и S_2.

Для каждого из них вычисляется расстояние между целью (точкой Z на рисунке) и центроидами сфер, как показано в листинге 10.4 с реализацией метода `SsNode::intersectsPoint` на псевдокоде. Поскольку для обеих сфер вычисленное (евклидово) расстояние меньше их радиусов, это означает, что одна (или обе) может содержать искомую целевую точку, поэтому нужно обойти обе ветви, S_1 и S_2.

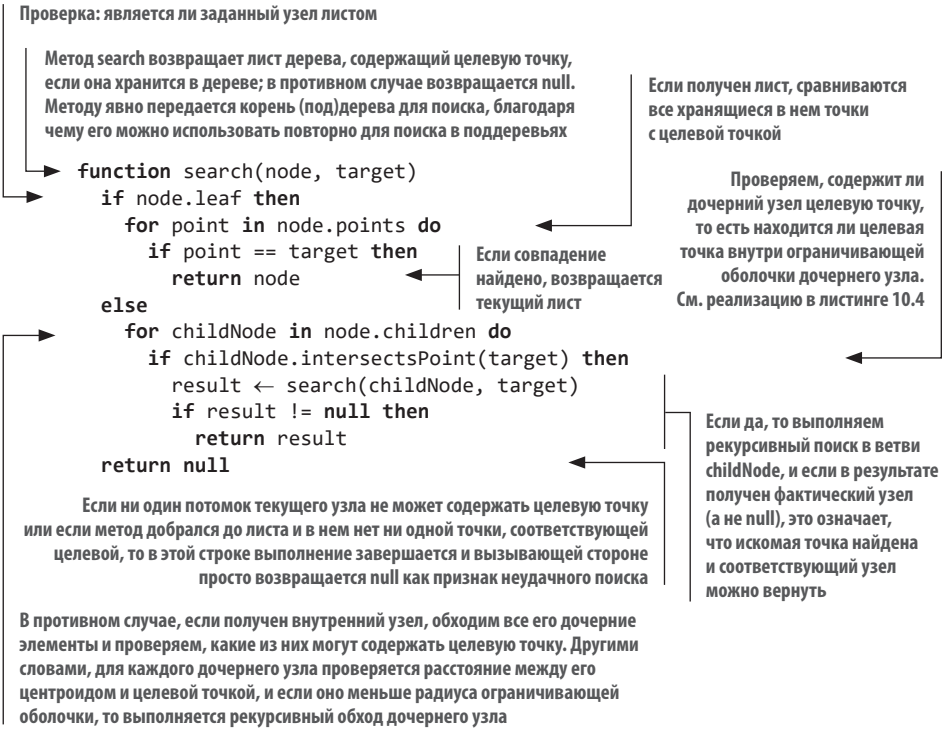
Это также видно на рис. 10.10, где точка Z явно лежит на пересечении сфер S_1 и S_2.

Следующие несколько шагов на рис. 10.10 (нижняя половина) и 10.11 выполняются одними и теми же строками кода, которые циклически перебирают дочерние элементы узла, пока не будет достигнут лист. Стоит отметить, что эта реализация выполняет обход узлов в глубину: она будет последовательно спускаться к листьям и возвращаться при первой же возможности. Для экономии места на этих рисунках пути обхода показаны параллельно, что вполне можно реализовать, немного изменив код. Однако эти изменения будут зависеть от конкретного языка программирования, поэтому будем придерживаться более простого и менее затратного последовательного варианта.

---

[1] Высота дерева не превосходит $\log_m(n)$ поэтому, если $m == M / 2$ (случай с наибольшей высотой), $\log_{N / 2}(n) \sim= \log_M(n)$.

Листинг 10.3. Метод search



Иногда метод будет проходить по ветвям, в которых ни один из дочерних элементов не может содержать целевую точку. Так обстоит дело, например, с узлом, содержащим S_3 и S_4. Выполнение в таких случаях просто завершится в строке 12 листинга 10.3, и вызывающая сторона получит `null`. В нашем примере метод первой просмотрит ветвь S_1, а затем цикл `for-each` в строке 7 перейдет к ветви S_2.

Когда наконец будут достигнуты листья S_{12}—S_{14}, запустится цикл в строке 3, проверяющий точки листа в поисках точного совпадения. Если совпадение найдено, то метод может вернуть текущий лист в качестве результата (конечно, мы предполагаем, что дерево не содержит дубликатов).

В листинге 10.4 показана простая реализация метода, проверяющего, попадает ли искомая точка в пределы ограничивающей оболочки узла. Реализация действительно очень простая, потому что использует только базовые геометрические вычисления. Но обратите внимание, что функция `distance` является структурным параметром SS-дерева, она может вычислять евклидово расстояние в k-мерном пространстве или какую-то другую метрику[1].

[1] Если удовлетворяет требованиям допустимости метрики: всегда неотрицательна, равна нулю только при вычислении расстояния между точкой и самой собой, симметрична и соблюдает треугольное неравенство.

Листинг 10.4. Метод SsNode::intersectsPoint

Метод intersectsPoint класса SsNode. Принимает точку и возвращает true, если точка находится внутри ограничивающей оболочки узла

```
function SsNode:: intersectsPoint(point)
  return distance(this.centroid, point) <= this.radius
```

Поскольку ограничивающая оболочка — это гиперсфера, достаточно получить расстояние от точки в аргументе point до центроида узла и сравнить его с радиусом. Здесь distance может быть любой метрической функцией, включая (по умолчанию) функцию евклидова расстояния в R^k

10.3.2. Вставка

Как уже упоминалось, операция вставки начинается с поиска. Для более простых деревьев, таких как двоичные деревья поиска, неудачный поиск возвращает единственный узел, куда можно добавить новый элемент, но для SS-деревьев возникает та же проблема, что и для R-деревьев, которую мы кратко обсудили в разделе 10.2: поскольку узлы могут и будут перекрываться, в дереве может обнаружиться несколько листьев, подходящих для вставки новой точки.

Это настолько важно, что указанный аспект упомянут как второе свойство, определяющее форму SS-дерева. Чтобы выбрать ветвь или один из листьев для вставки новой точки, нужен некоторый эвристический метод.

SS-деревья изначально использовали простую эвристику: на каждом шаге выбиралась ветвь, центроид которой находится ближе всего к вставляемой точке (те редкие неоднозначности, с которыми придется столкнуться, можно разрешать произвольно).

Само по себе неидеальное решение, потому что может привести к ситуации, подобной той, что изображена на рис. 10.12. Здесь новая точка *Z* могла бы быть добавлена в лист, уже охватывающий ее, но вместо этого оказывается в другом листе, ограничивающая оболочка которого расширяется, чтобы принять *Z*, и в результате перекрывает другой лист. Также возможно, хотя и маловероятно, что выбранный лист на самом деле не является ближайшим к цели. На каждом уровне выбирается ближайший узел, поэтому, если дерево плохо сбалансировано, может случиться так, что в какой-то момент в процессе обхода метод наткнется на перекошенную сферу с центроидом, отстоящим далеко от маленького листа, как, например, в случае со сферой S_6 на рис. 10.12, чей потомок S_{14} находится далеко от ее центроида.

С другой стороны, использование этой эвристики значительно упрощает код и уменьшает время его выполнения, что позволяет получить оценку для наихудшего случая (для этой операции) $O(\log_m(n))$, потому что обход производится только по одному пути от корня к листу. Если бы потребовалось пройти все ветви, пересекающие Z, то в худшем случае пришлось бы обойти все листья.

Кроме того, необходимость обрабатывать разные случаи, когда не найдено ни одного листа, только один лист или более одного листа, подходящего для вставки `Z`, усложнила бы код.

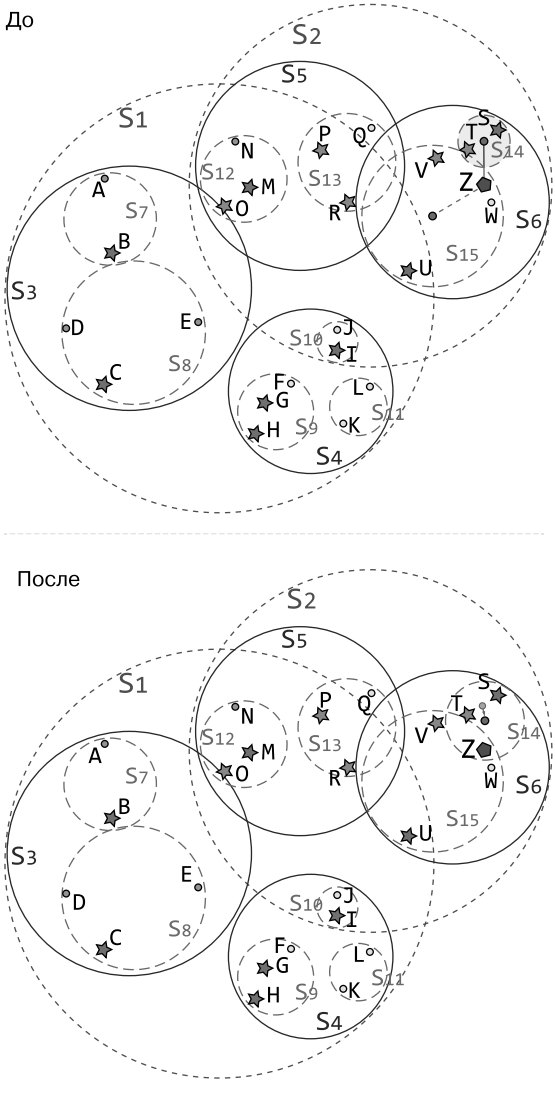
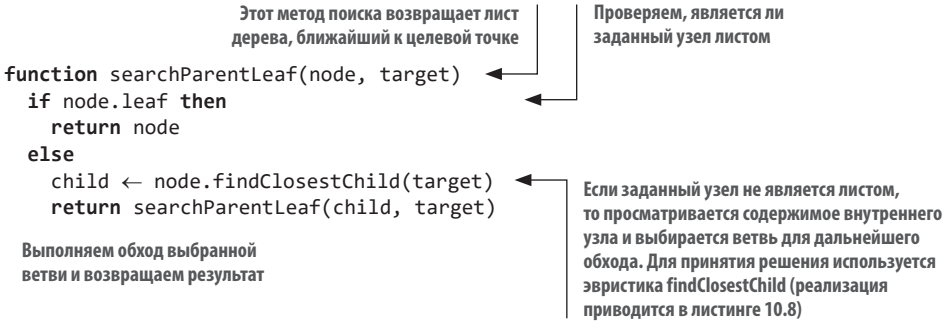


Рис. 10.12. Пример, когда точка Z добавляется в ближайший лист S_{14}, чья ограничивающая оболочка в результате становится больше и перекрывается ограничивающей оболочкой другого существующего листа S_{15}, в пределах которой умещается Z. Как показано внизу, в результате добавления новой точки в сферу S_{14} ее центроид смещается

Итак, используем оригинальную эвристику, описанную в первой статье, где были представлены SS-деревья (листинг 10.5). Ее можно считать упрощенной версией метода поиска, описанного в подразделе 10.3.1, потому что она проходит только один путь в дереве. На рис. 10.13 показана разница при вызове метода поиска для того же дерева и точки (для сравнения см. рис. 10.10 и 10.11).

Листинг 10.5. Метод searchParentLeaf



Однако код в листинге 10.5 предназначен только для иллюстрации выполнения обхода. В самом методе `insert` вызывать его отдельно не будем, а интегрируем в код метода, потому что поиск ближайшего листа — лишь первый шаг. До завершения операции вставки еще далеко, и может понадобиться вернуться назад. Вот почему мы реализуем `insert` как рекурсивную функцию, и каждый раз, когда рекурсивный вызов возвращает значение, будет происходить возврат в пути от корня к текущему узлу.

Предположим, в результате поиска принято решение, что Z нужно добавить в некоторый лист L, который уже содержит j точек. Известно, что $j \geq m > 1$, поэтому лист не пустой, но могут возникнуть три совершенно разные ситуации.

1. Если L уже содержит Z, то ничего делать не нужно, потому что поддержка дубликатов не предполагается (в противном случае можно обратиться к оставшимся двум случаям).
2. $j < M$ — в этом случае Z добавляется в список дочерних элементов листа L, повторно вычисляется центроид и радиус сферы для L и действия завершаются. Описанная ситуация показана на рис. 10.12, где $L == S_{14}$. В левой части рисунка видно, что в результате добавления Z в S_{14} центроид и радиус ограничивающей оболочки обновляются.
3. $j == M$ — самый сложный случай, потому что, если добавить еще одну точку в L, нарушится инвариант, требующий, чтобы лист содержал не более M точек. Единственный способ решить эту проблему — разделить точки листа на два набора и создать два новых листа, которые будут добавлены в N — родительский узел узла L. К сожалению, при этом есть риск оказаться в той же ситуации, когда у N уже имелось M дочерних элементов. И снова единственный способ решить проблему — разбить дочерние элементы N на два набора (определив две сферы), удалив N из родительского узла P и добавив две новые сферы в P. Очевидно, что теперь в P тоже может оказаться $M + 1$ потомков! Проще говоря, нужно выполнить возврат к корню и остановиться, только добравшись до узла, имеющего менее M дочерних элементов, или достигнув корня. Если понадобится разделить корень, то нужно создать новый корень с двумя дочерними элементами, а высота дерева при этом увеличится на 1 (и это единственный случай, когда такой рост может произойти).

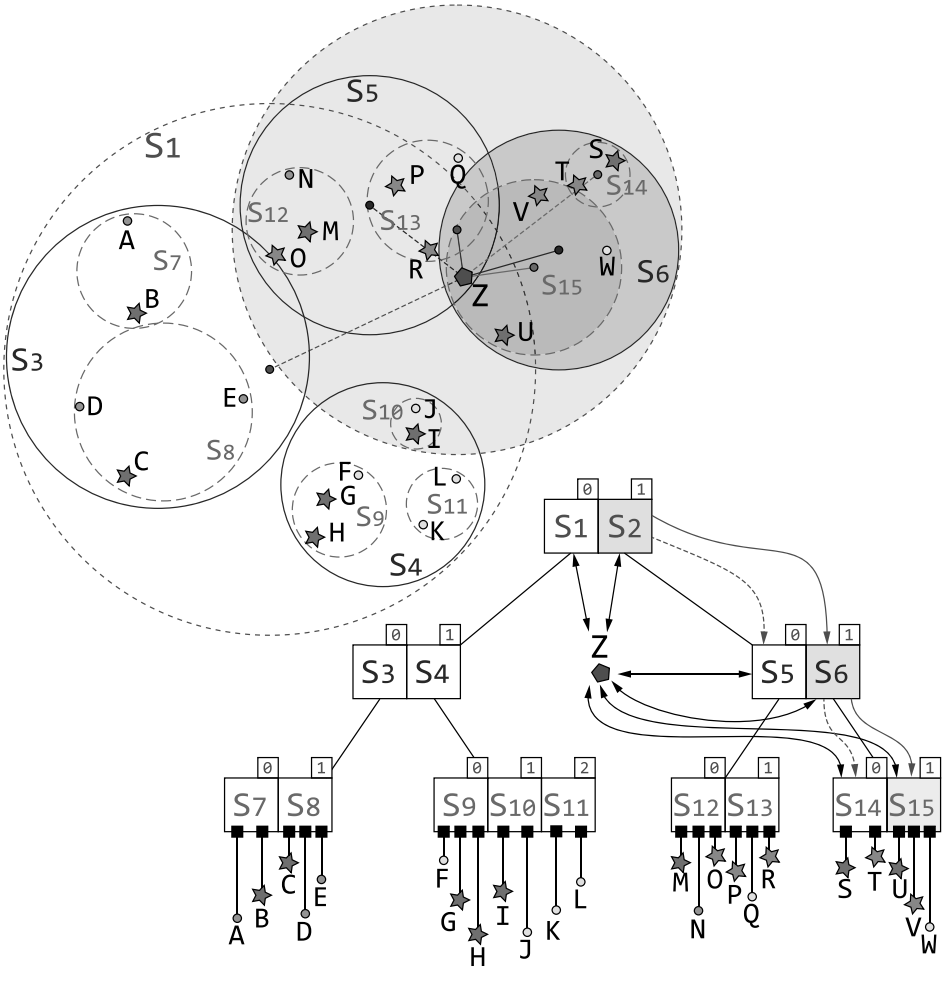


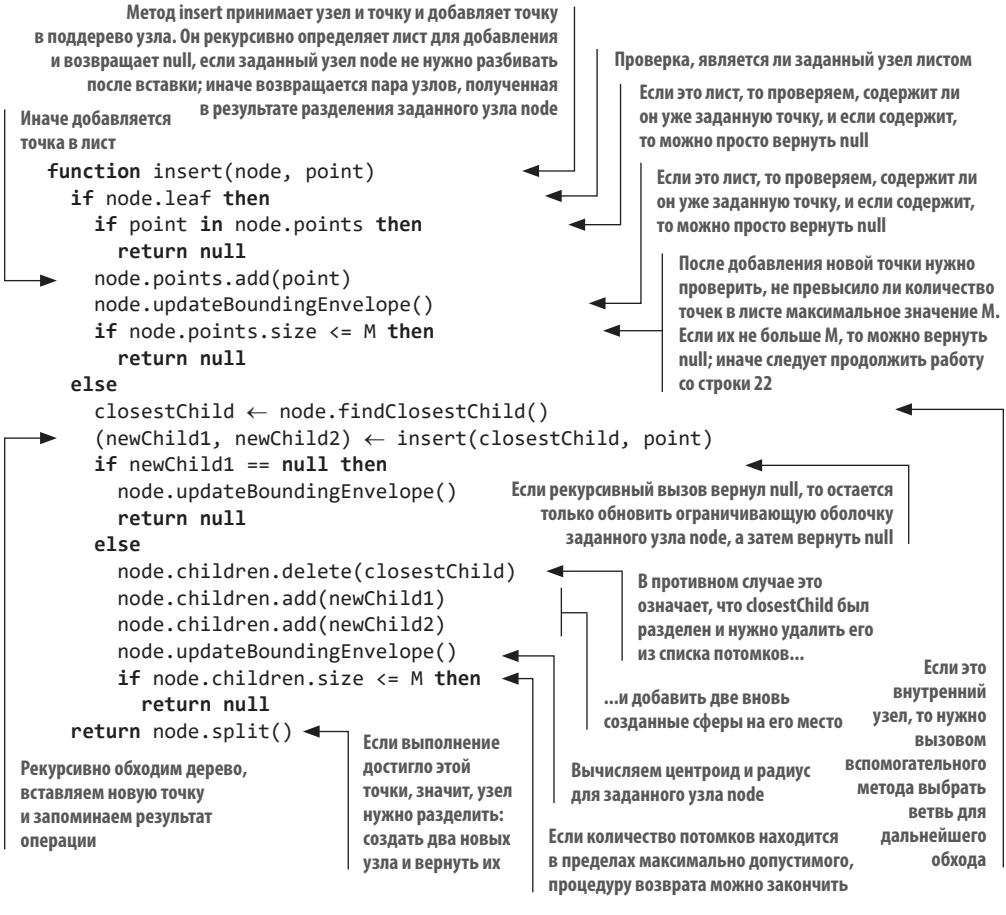
Рис. 10.13. Пример обхода дерева в методе searchParentLeaf. В отличие от рис. 10.10 и 10.11, для экономии места здесь шаги сведены в одну диаграмму. Такое компактное представление возможно благодаря тому факту, что обход производится только по одному пути. Обратите внимание, как на каждом шаге вычисляется расстояние между Z и центроидом текущего узла и выбирается только ветвь с кратчайшим расстоянием (обведенная более толстой сплошной линией). На этом рисунке расстояния выделены тем же цветом, что и соответствующие сферы, а отрезки, изображающие расстояния, одним концом упираются в центры сфер, для которых они вычисляются, поэтому легко определить расстояние до корневого узла, сфер на уровне 1 и т. д. Пройденные ветви сфер выделены на обоих изображениях

В листинге 10.6 показана реализация метода `insert` с использованием только что описанных случаев:

- обход дерева, эквивалентный методу `searchParentLeaf`, производится в строках 10 и 11;

- случай 1 обрабатывается в строке 3, где возвращается `null`, чтобы сообщить вызывающей стороне, что дальнейшие действия не требуются;
- случай 2 обрабатывается строками 6 и 18, что также приводит к возврату `null` из метода;
- случай 3, который является наиболее сложным вариантом, реализован в строках 19 и 20;
- возврат производится в строках с 12-й по 21-ю.

Листинг 10.6. Метод insert




```
function insert(node, point)
  if node.leaf then
    if point in node.points then
      return null
    node.points.add(point)
    node.updateBoundingEnvelope()
    if node.points.size <= M then
      return null
  else
    closestChild ← node.findClosestChild()
    (newChild1, newChild2) ← insert(closestChild, point)
    if newChild1 == null then
      node.updateBoundingEnvelope()
      return null
    else
      node.children.delete(closestChild)
      node.children.add(newChild1)
      node.children.add(newChild2)
      node.updateBoundingEnvelope()
      if node.children.size <= M then
        return null
  return node.split()
```

Рисунки 10.14 и 10.15 иллюстрируют третий случай, когда точка вставляется в лист, который уже содержит *M* точек. В общем случае вставка в SS-дерево следует алгоритму вставки в B-дерево. Единственная разница заключается в способе разделения узлов (в B-деревьях список элементов просто разбивается на две половины). Конечно, в B-деревьях ссылки на дочерние элементы и их порядок также обрабатываются по-разному, как было показано в разделе 10.2.

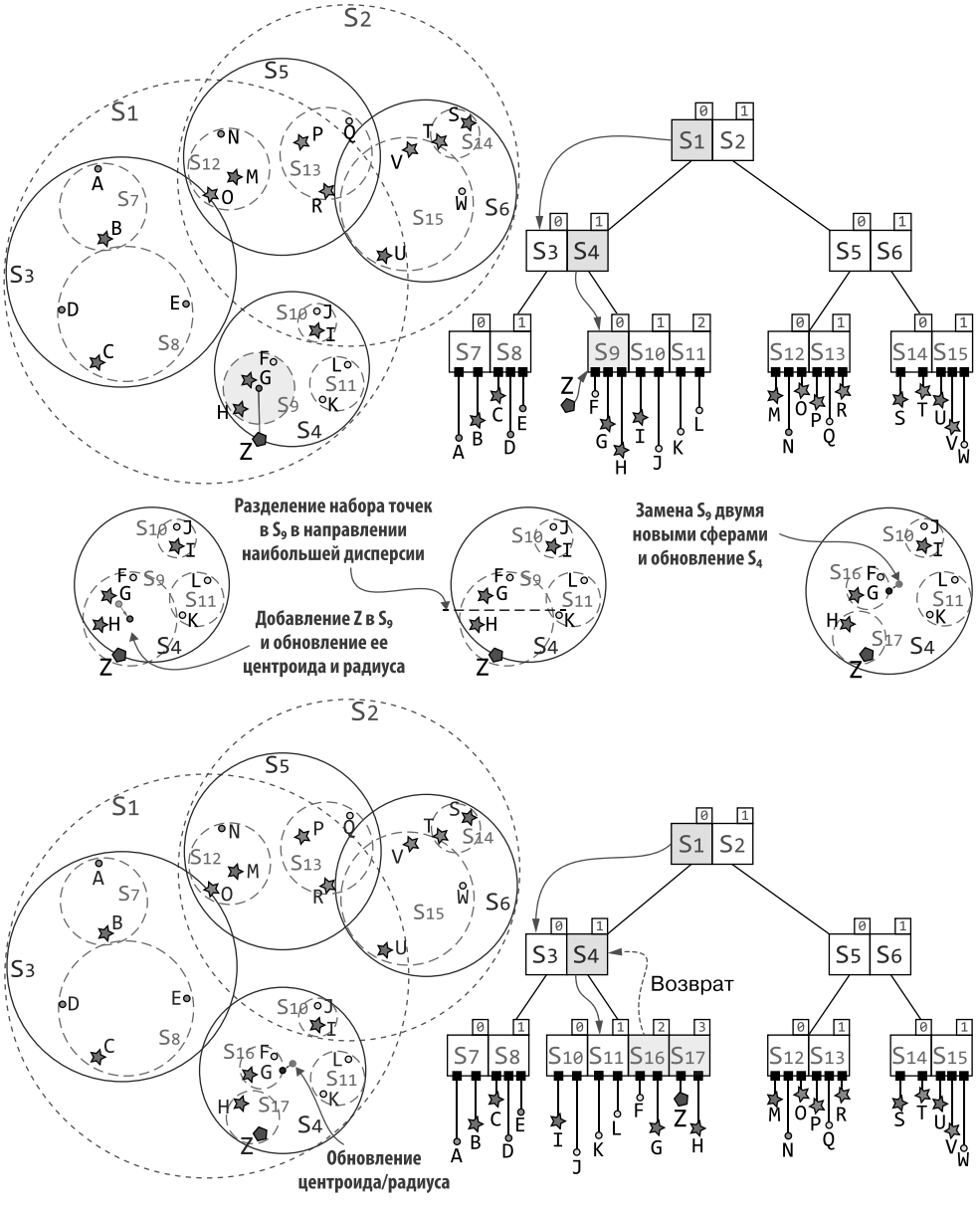


Рис. 10.14. Вставка точки в полный лист. *Вверху:* поиск нужного листа. *В центре:* крупный план соответствующей области. S_9 необходимо обновить, пересчитав ее центроид и радиус. Затем можно выбрать направление, вдоль которого точки имеют наибольшую дисперсию (у в этом примере), и разделить точки так, чтобы дисперсия двух новых наборов точек была минимальной. Наконец, S_9 удаляется из списка потомков ее родителя S_4 и добавляются два новых листа, содержащих два набора точек (они получены в результате разделения). *Внизу:* окончательный результат; теперь нужно обновить центроид и радиус S_4 и выполнить возврат

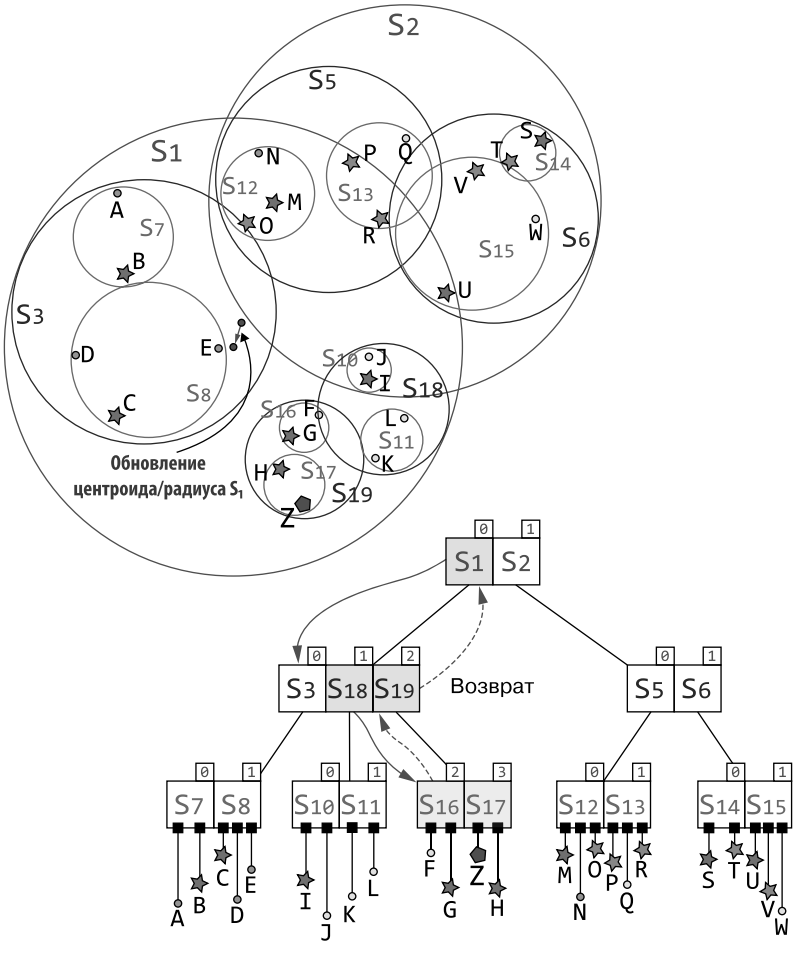


Рис. 10.15. Возврат в методе insert после разделения листа. В продолжение рис. 10.14: после разделения листа S_9 на узлы S_{16} и S_{17} производится возврат к S_4 — родителю узла S_9 и в него добавляются два только что созданных новых листа, как показано в конце на рис. 10.14. Теперь S_4 имеет четырех потомков, на одного больше, чем можно. Его тоже нужно разделить. Здесь показан результат разделения S_4 на два новых узла, S_{18} и S_{19}, которые будут добавлены в S_1 — родитель узла S_4, к которому мы вернемся в процессе возврата. Поскольку теперь у него только три потомка (и M == 3), для него просто заново вычисляются центроид и радиус для ограничивающей оболочки S_1. И на этом можно прекратить поиск с возвратом

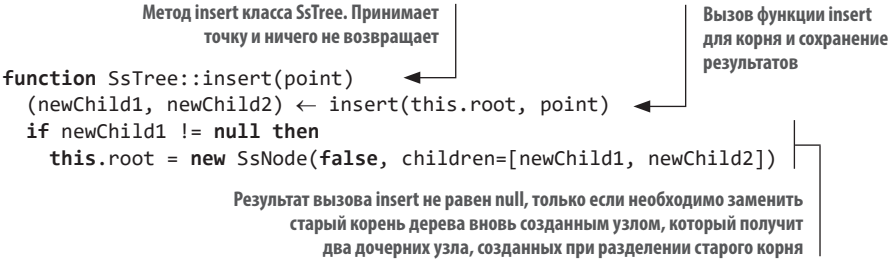
В листинге 10.6 с реализацией метода `insert` использовано несколько вспомогательных функций[1]. Однако есть еще один случай, который не был упомянут. Что следует сделать, если по достижении корня обнаружится, что его тоже нужно разделить?

[1] Одно из золотых правил чистого кода — разбивать длинные сложные методы на более мелкие, чтобы каждый метод был сосредоточен на решении единственной задачи.

Этот случай не рассматривается как часть метода в листинге 10.6, потому что он требует обновления корня дерева, а эта операция должна выполняться над экземпляром класса дерева, который имеет доступ к корню.

Поэтому рассмотрим явную реализацию метода `insert` дерева. Напомню, что на самом деле рассматриваются только методы, определенные в классах структур данных (`KdTree`, `SsTree` и т. д.), а не в классах узлов (таких как `SsNode`). Но обычно первые опускаются в тех случаях, когда они являются простыми обертками вокруг методов узлов. В листинге 10.7 показано, как выполнить разделение корня. Кроме того, хочу подчеркнуть еще раз: этот фрагмент кода — единственная точка роста нашего дерева.

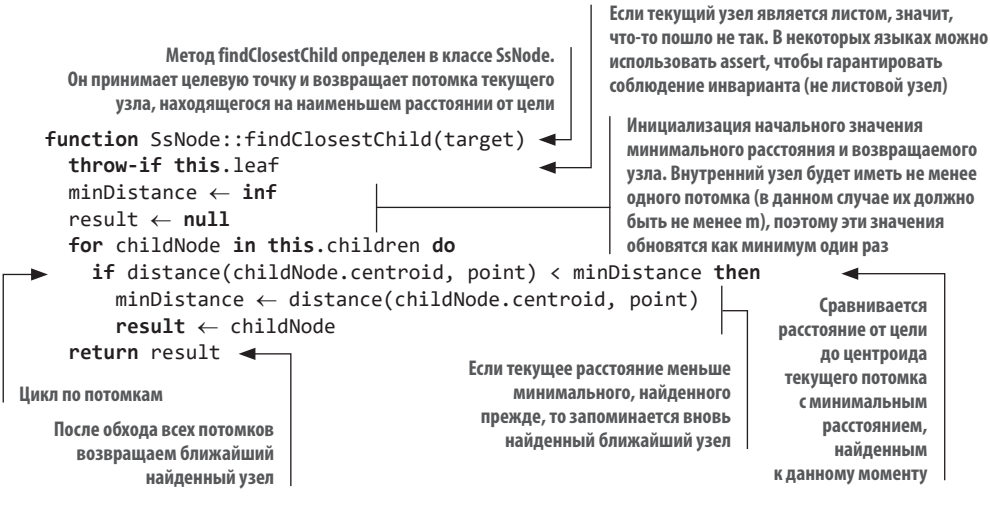
Листинг 10.7. Метод SsTree::insert



10.3.3. Вставка: дисперсия, средние значения и проекции

Теперь подробно рассмотрим вспомогательные методы, вызываемые в листинге 10.6, начав с эвристического метода (см. листинг 10.8), определяющего ближайший дочерний узел к точке *Z*. Как уже говорилось, для этого определения нужно просто пройти по потомкам узла, вычислить расстояния между их центроидами и *Z* и выбрать ограничивающую оболочку, находящуюся на минимальном расстоянии.

Листинг 10.8. Метод SsNode::findClosestChild



На рис. 10.16 показано, что происходит, когда возникает необходимость разделить лист. Сначала повторно вычисляется радиус и центроид листа после включения новой точки, а затем вычисляется дисперсия координат $M + 1$ точек по k осям, чтобы найти ось с наибольшей дисперсией. Это особенно полезно при работе с перекошенными наборами точек, такими как S_9 в нашем примере, и помогает уменьшить объем сфер и, соответственно, площади их перекрытия.

Взглянув на рис. 10.16, можно заметить, как разделение по оси X дало бы два множества с точками G и H с одной стороны и F и Z — с другой. После сравнения этой ситуации с результатом с рис. 10.14 отпадают все сомнения в том, какой конечный результат лучший!

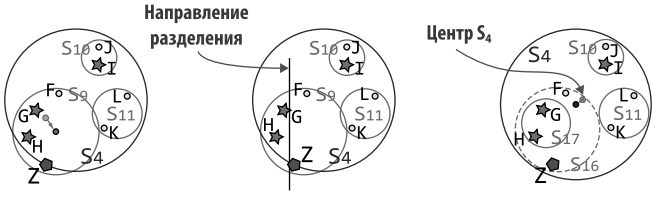


Рис. 10.16. Разделение листа в неоптимальном направлении. В этом случае ось X является направлением с минимальной дисперсией. Сравнивая окончательный результат с рис. 10.14, можно заметить, что, хотя форма S_4 существенно не изменилась, сфера S_{16} увеличилась более чем в два раза и теперь полностью охватывает S_{17}, а это означает, что любой поиск, нацеленный на S_{17}, также встретит на своем пути S_{16}

Конечно, в реальности результат не всегда получается таким идеальным. Если направление максимальной дисперсии повернуто на некоторый угол относительно оси X (представьте, например, что те же точки повернуты на 45° по часовой стрелке относительно центроида листа), то разделение ни по одному из направлений не даст оптимального результата. Однако в среднем это более простое решение действительно помогает.

Итак, как выполняется разделение? Начнем с листинга 10.9, где показан метод определения направления с максимальной дисперсией. Это простой метод поиска глобального максимума в линейном пространстве.

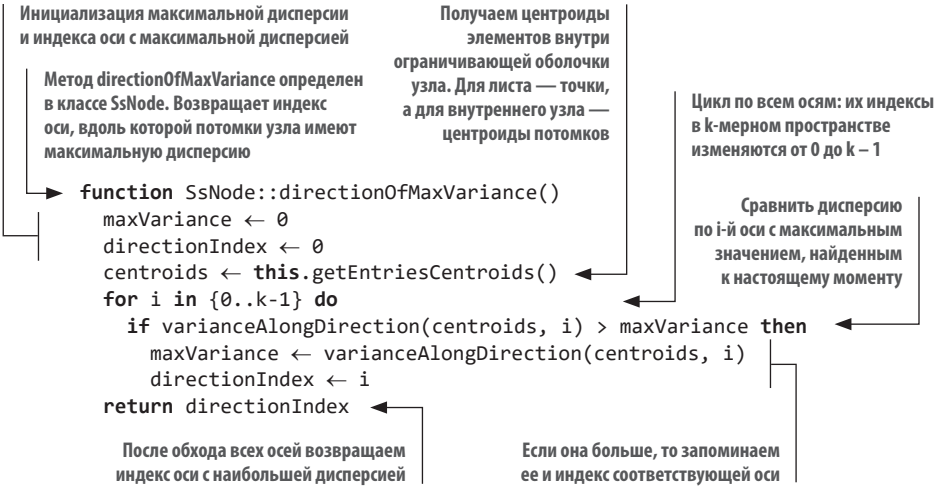
Дисперсию нужно вычислять на каждом шаге цикла `for` в строке 5. Возможно, сейчас самое время напомнить, что такое дисперсия и как она вычисляется. Для множества S действительных чисел среднее значение μ вычисляется как отношение суммы значений элементов к мощности множества:

$$\mu = \frac{1}{|S|}\sum_{s \in S} s.$$

После вычисления среднего значения можно определить дисперсию (обычно обозначаемую σ^2) как среднее значение квадратов разностей между средним значением S и каждым из его элементов:

$$\sigma^2 = \frac{1}{|S|}\sum_{s \in S} (s - \mu)^2.$$

Листинг 10.9. Метод SsNode::directionOfMaxVariance



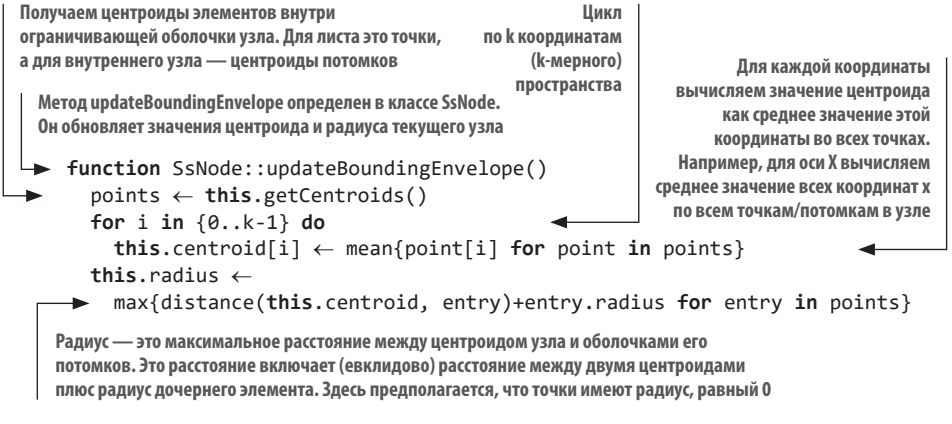
Итак, для набора n точек $P_0 \ldots P_{n-1}$, каждая точка P_j с координатами $(P_{(j,0)}, P_{(j,1)} \ldots P_{(j,k-1)})$, формулы среднего значения и дисперсии в направлении i-й оси будут иметь вид:

$$\mu_i = \frac{1}{n}\sum_{j=0}^{n-1} P_{j,i};$$

$$\sigma_i^2 = \frac{1}{n}\sum_{j=0}^{n-1}\left(P_{j,i} - \mu_i\right)^2.$$

Эти формулы легко переводятся в программный код, и в большинстве языков программирования есть уже готовые реализации метода вычисления дисперсии, поэтому не будем показывать здесь псевдокод. Вместо этого посмотрим, как обе функции вычисления дисперсии и среднего используются в методе `updateBoundingEnvelope` (листинг 10.10), который вычисляет центроид и радиус узла.

Листинг 10.10. Метод SsNode::updateBoundingEnvelope



Этот метод вычисляет координаты центроида узла как центр масс дочерних центроидов. Напомню, что для листьев дочерними элементами являются только содержащиеся в них точки, а для внутренних узлов — другие узлы.

Центр масс — это k-мерная точка, каждая координата которой является средним значением той же координаты всех остальных дочерних центроидов[1].

Получив новый центроид, нужно обновить радиус ограничивающей оболочки узла. Он определяется как минимальный радиус, при котором ограничивающая оболочка включает ограничивающие оболочки всех дочерних элементов текущего узла. Его, в свою очередь, можно определить как максимальное расстояние между центроидом текущего узла и любой точкой в его потомках. На рис. 10.17 показано, как и почему упомянутые расстояния вычисляются для каждого дочернего элемента: это сумма расстояния между двумя центроидами и радиуса дочернего элемента (пока мы предполагаем, что точки имеют радиус, равный 0, приведенное определение также работает для листьев).

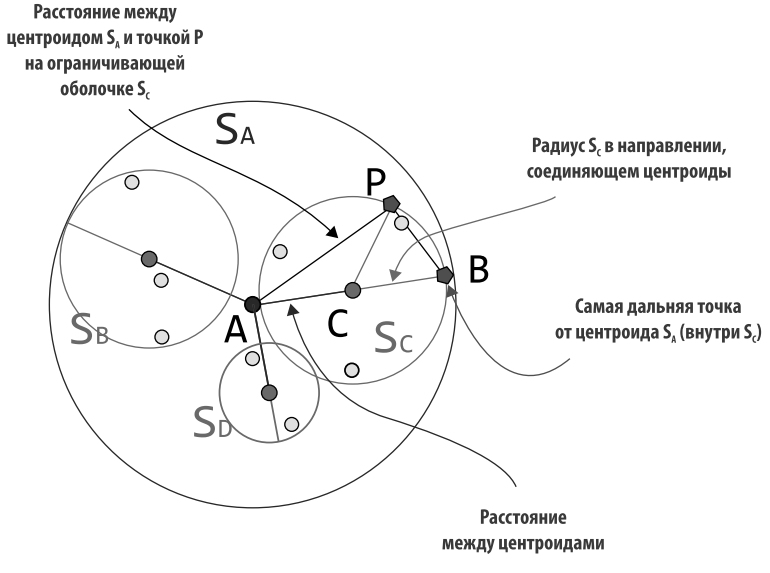


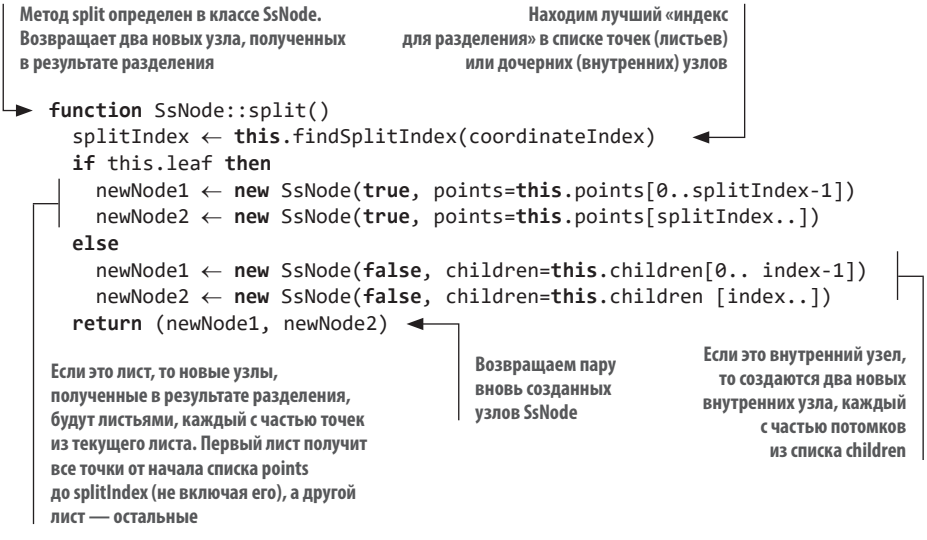
Рис. 10.17. Вычисление радиуса внутреннего узла. Точка в S_C, которая находится дальше от центроида A сферы S_A, является точкой на ограничивающей оболочке, которая дальше всего от C в направлении, противоположном центроиду S_A, и поэтому ее расстояние равно сумме расстояния A — C между двумя центроидами плюс радиус S_C. Если выбрать другую точку P на ограничивающей оболочке, ее расстояние от A должно быть меньше расстояния A — B, потому что метрика по определению должна подчиняться неравенству треугольника, а два других ребра треугольника ACP — это AC и CP, то есть радиус S_C. Вы можете проверить, что это правило верно для любой другой оболочки на рисунке

[1] Предполагается, что центроидом точки является сама точка. Также будем считать, что точки имеют радиус, равный 0.

10.3.4. Вставка: разделение узлов

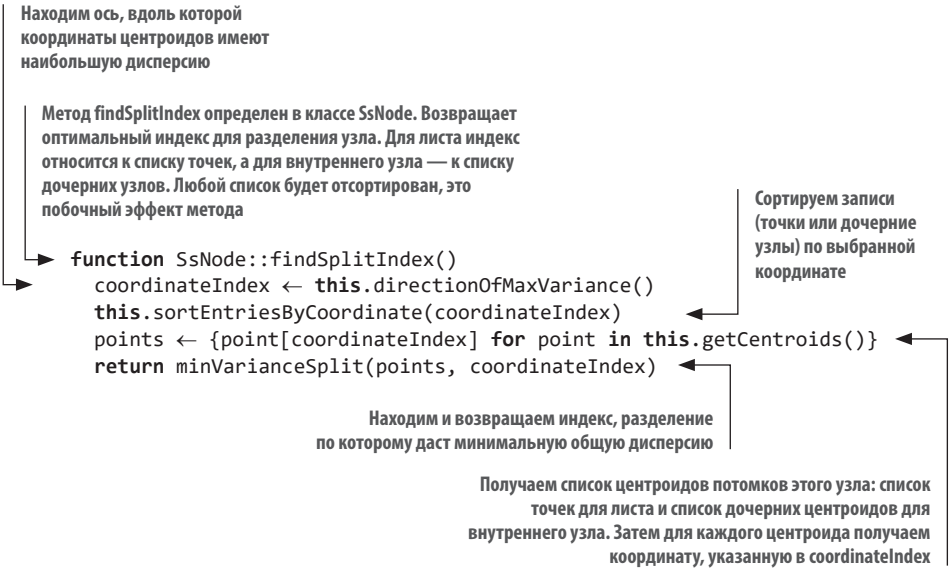
Теперь можно перейти к реализации метода `split`, показанной в листинге 10.11.

Листинг 10.11. Метод SsNode::split



Этот метод выглядит относительно простым, потому что бóльшую часть работы выполняет вспомогательный метод `findSplitIndex`, представленный в листинге 10.12.

Листинг 10.12. Метод SsNode::findSplitIndex



После выявления направления с максимальной дисперсией метод сортирует[1] точки или дочерние элементы (в зависимости от того, что это за узел — лист или внутренний узел) по значениям тех же координат, а затем, получив список центроидов, разделяет его опять же по направлению максимальной дисперсии. Ниже будет показано, как это сделать.

Но сначала отмечу: мы снова столкнулись с методом, возвращающим центроиды элементов в ограничивающей оболочке узла, поэтому, по всей видимости, пришло время определить его! Как упоминалось выше, метод имеет двоякую логику:

- если узел является листом, то метод должен вернуть содержащиеся в нем точки;
- иначе он должен вернуть центроиды дочерних элементов узла.

Реализация метода на псевдокоде показана в листинге 10.13.

Листинг 10.13. Метод SsNode::getEntriesCentroids

Метод getEntriesCentroids определен в классе SsNode. Возвращает центроиды элементов внутри ограничивающей оболочки узла

```
function SsNode::getEntriesCentroids()
  if this.leaf then
    return this.points
  else
    return {child.centroid for child in this.children}
```

Иначе возвращаем список всех центроидов потомков этого узла. Для обозначения этого списка здесь используется конструкция, обычно называемая генератором списков, или списковым включением (приложение А)

Если это лист, то возвращается список его точек

После получения индекса точки разделения можно разделить элементы фактически. Теперь нам понадобится условный оператор, чтобы по-разному обработать листья и внутренние узлы: конструктору узлов нужно передать правильные аргументы для создания узла нужного типа. После создания новых узлов остается только вернуть их.

Но не расслабляйтесь, это еще не финал! Я понимаю, что мы уже довольно много времени провели в этом разделе, но пока не хватает еще одной детали, чтобы завершить реализацию метода `insert`: вспомогательной функции `splitPoints`.

Сам метод может показаться тривиальным, но в действительности реализовать его правильно не так-то просто. Скажем так: нужно немного подумать.

Сначала рассмотрим пример, а затем напишем для него какой-нибудь псевдокод. На рис. 10.18 показаны шаги, которые нужно выполнить, чтобы получить желаемое разделение. Начнем с узла, содержащего восемь точек. Неизвестно, да и не должно быть известно, являются ли они точками из набора данных или центроидами узлов; для метода это не имеет никакого значения.

---

[1] Внимание: функция, возвращающая значение и имеющая побочный эффект, — не самое хорошее решение. Лучше было бы использовать косвенную сортировку. Здесь реализовано самое простое решение просто из-за ограниченного объема книги, но вы имейте в виду и альтернативный вариант.

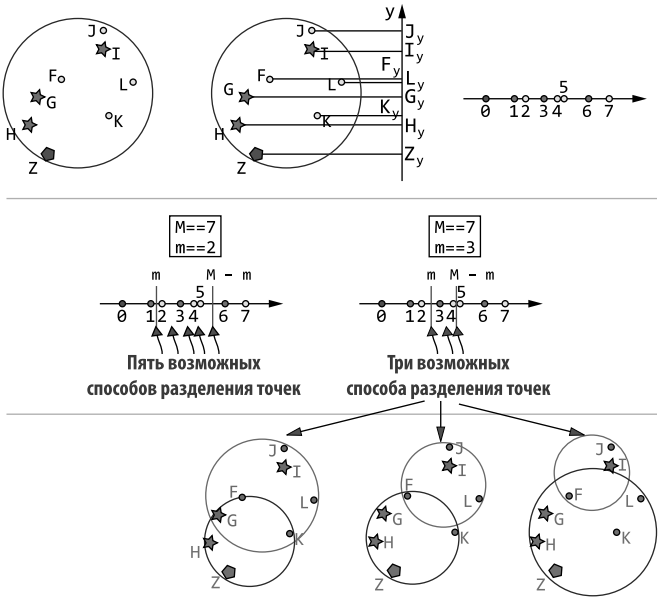


Рис. 10.18. Разделение набора точек в направлении максимальной дисперсии. *Вверху:* ограничивающая оболочка и ее точки, которые нужно разделить; направление максимальной дисперсии ориентировано вдоль оси Y *(в центре)*, поэтому все точки проецируются на эту ось. *Справа* для удобства ось повернута по часовой стрелке на 90°, а метки точек заменены на их индексы. *В середине:* учитывая, что количество точек равно 8, можно заключить, что M должно быть равно 7. Тогда m может быть любым значением ≤ 3. Поскольку в алгоритме выбирается один индекс разделения, при этом точки разбиваются на два раздела и в каждом разделе должно оказаться не менее m точек, то в зависимости от фактического значения m может иметься разное количество вариантов для выбора индекса разделения. *Внизу:* здесь показаны три возможных разделения для случая, когда m == 3: индекс разделения может быть 3, 4 или 5. Выбирается вариант, для которого сумма дисперсий для двух наборов точек минимальна

Допустим, что мы вычислили направление максимальной дисперсии и оно совпадает с направлением оси *Y*. Теперь нужно спроецировать точки на эту ось, что эквивалентно рассмотрению только координаты *y* точек в принятой системе координат.

Для большей наглядности проекции показаны на схеме. По той же причине ось *Y* и проекции повернуты на 90° по часовой стрелке и убраны метки точек, а вместо них оставлены индексы, следующие в порядке возрастания слева направо. В программном коде пришлось бы отсортировать точки по координате *y* (как это было сделано в листинге 10.12), после чего можно было бы просто прочитать их индексы. Другое решение — использовать косвенную сортировку и создание таблицы с индексами отсортированных и неотсортированных точек, но последнее существенно усложнит оставшийся код.

В нашем примере имеются восемь точек, которые нужно разделить. Отсюда можно сделать вывод, что параметр M, максимальное количество листьев/потомков для узла дерева, равен 7, а m, минимальное количество элементов, может быть равно только 2 или 3 (технически может быть и 1, но такой выбор приводит к получению перекошенных деревьев, поэтому не стоит создавать деревья с параметром $m == 1$).

Хочу еще раз напомнить, что значение m должно выбираться во время создания SS-дерева, и поэтому на момент вызова `split` оно уже зафиксировано. Здесь же мы просто рассуждаем о том, как этот выбор влияет на порядок выполнения разделения и в конечном счете на структуру дерева.

И действительно, этот параметр имеет решающее значение для метода `split`, потому что каждый из двух созданных разделов должен иметь не менее m точек. Следовательно, поскольку разделение производится по одному индексу[1], индекс разделения может иметь значения от m до $M - m$. В нашем примере, как показано в центре на рис. 10.18, это означает:

- если $m == 2$, то можно выбрать любой индекс от 2 до 6 (пять вариантов);
- если $m == 3$, то можно выбрать любой индекс от 3 до 5 (три варианта).

Теперь предположим, что остановились на значении $m == 3$. Внизу на рис. 10.18 показаны получившиеся разделения для каждого из трех вариантов выбора индекса разделения. Нам нужно отдать предпочтение тому, который минимизирует дисперсию для обоих узлов (обычно минимизируется сумма дисперсий), но мы минимизируем дисперсию только в том направлении, в каком выполняется разделение, поэтому в примере вычислим только дисперсию координаты y двух наборов точек. В отличие от R-деревьев на данном этапе не станем пытаться минимизировать перекрытие ограничивающих оболочек, однако, как оказывается, минимизация дисперсии в направлении, где она наибольшая, косвенно приводит к уменьшению среднего перекрытия новых узлов.

Кроме того, когда мы начнем знакомиться с деревьями SS$^+$, то проблему перекрытия ограничивающих оболочек будем решать отдельно.

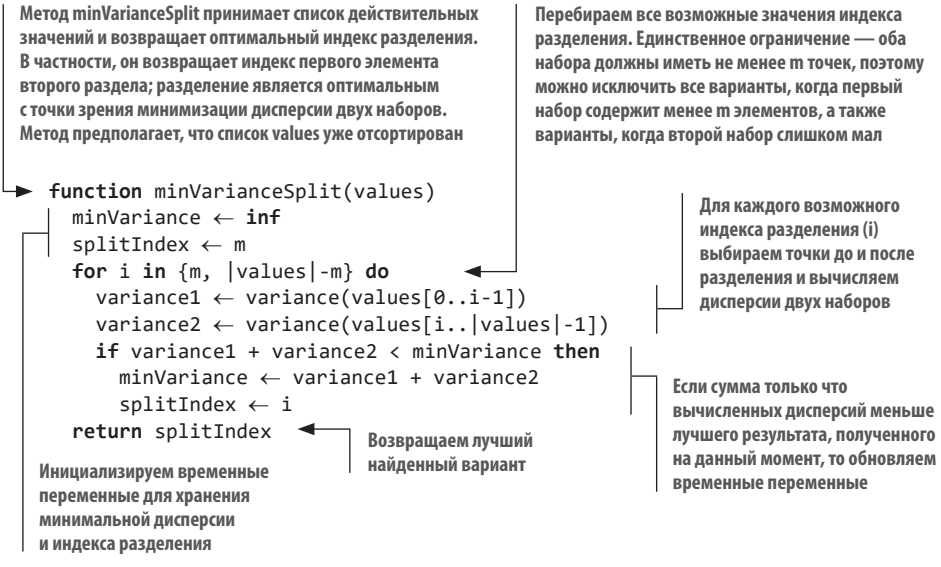
А пока, чтобы покончить с методом вставки, рассмотрим листинг 10.14 с реализацией метода `minVarianceSplit`. Как уже упоминалось, это просто линейный поиск индекса разделения точек среди $M - 2(m - 1)$ возможных вариантов.

На этом можно завершить раздел о методе `SsTree::insert`. Может показаться, что путь был очень долгим, и так оно и есть: это, пожалуй, самый сложный код, который мы рассматривали. Не торопитесь, перечитайте последние подразделы еще несколько раз, если это поможет вам достичь полного понимания. А теперь приготовьтесь: далее мы приступим к методу удаления, который, как мне кажется, еще более сложный.

---

[1] В случае с SS-деревом разбивается упорядоченный список точек, для чего выбирается индекс разделения, в результате все точки слева от индекса попадают в один раздел, а все точки справа — в другой.

Листинг 10.14. Метод minVarianceSplit



10.3.5. Удаление

Подобно методу `insert`, метод `delete` в SS-деревьях тоже в значительной степени основан на реализации метода `delete` B-дерева. Первый обычно считается настолько сложным, что во многих руководствах его вообще опускают (для экономии места) или отодвигают его обсуждение как можно дальше. SS-деревья, конечно, еще сложнее.

Но одно из преимуществ R-деревьев и SS-деревьев перед *k*-мерными деревьями, заключается в том, что последние гарантированно сбалансированы, только если создаются на основе статического набора данных, а первые два могут оставаться сбалансированными даже при хранении динамических наборов данных и выполнении большого количества операций вставки и удаления. Поэтому отказ от обсуждения реализации метода `delete` будет выглядеть странно, учитывая одну из основных причин, по которой нам понадобилась эта структура данных.

Первый (и самый простой) шаг — найти точку, которую нужно удалить, или, точнее, найти лист, содержащий эту точку. Если в методе `insert` потребовалось бы пройти только один путь к ближайшему листу, то в `delete` необходима полноценная реализация алгоритма поиска, описанного в подразделе 10.3.1. Однако в `insert` требовалось выполнять возвраты, поэтому вместо вызова метода `search` придется реализовать тот же обход в этом новом методе[1].

[1] Мы могли бы повторно использовать search как первый вызов в delete (и searchClosestLeaf в insert), если бы в каждом узле хранили указатель на его родителя, чтобы иметь возможность подниматься вверх по дереву, когда это необходимо.

Найдя правильный лист L и точку Z в дереве (иначе не нужно было бы выполнять какие-либо изменения), получаем несколько возможных ситуаций — простую, сложную и очень сложную.

1. Если лист содержит более m точек, то достаточно просто удалить Z из L и обновить его ограничивающую оболочку.
2. В противном случае после удаления Z в L останется только $m - 1$ точек, и тем самым будет нарушен один из инвариантов SS-дерева. Есть несколько вариантов решения этой проблемы:
   а) если L — корень, то ничего не нужно делать, потому что для корня это допустимо;
   б) если у L есть хотя бы один братский узел S с более чем m точками, то можно переместить одну точку из S в L. Но нужно быть очень осторожными, выбирая ближайшую к L точку (среди всех братских узлов с не менее чем $m + 1$ точками), потому что эта операция может вызвать значительное расширение ограничивающей оболочки L (если достаточное количество точек есть только в братских узлах, находящихся далеко от L) и разбалансировать дерево;
   в) если ни один братский узел не может «одолжить» точку, то придется объединить L с одним из них. И снова нужно выбирать, с каким братским узлом объединить L. Для решения этой задачи можно использовать разные стратегии:
      - выбрать ближайший братский узел;
      - выбрать братский узел с наибольшей площадью перекрытия с L;
      - выбрать братский узел, объединение с которым минимизирует отклонение координат (по всем осям).

Случай 2, *в*, явно самый сложный в реализации. Случай 2, *б*, однако, относительно простой, потому что, к счастью, одно из отличий от B-деревьев заключается в отсутствии необходимости сортировать дочерние элементы узла и выполнять повороты при перемещении одной точки из S в L. В середине и внизу на рис. 10.19 можете видеть результат «заимствования» узлом S_3 одного из потомков узла S_4 — S_9, что выглядит довольно просто. Конечно, самое сложное — выбрать, у какого из братских узлов «позаимствовать» потомка и какого именно потомка следует переместить. В случае 2, *б*, слияние двух узлов приведет к тому, что у их родителя останется на одного потомка меньше; поэтому необходимо будет вернуться назад и убедиться, что этот узел все еще имеет не менее m дочерних элементов. Это показано вверху и в центре на рис. 10.19. Но есть и хорошая новость — внутренние узлы можно обрабатывать точно так же, как листья, и, следовательно, повторно использовать ту же логику (и практически тот же код) для листьев и внутренних узлов.

Случаи 1 (внизу на рис. 10.19) и 2, *а*, тривиальны, и их легко реализовать. Именно тот факт, что, добравшись до корня, не нужно делать никаких дополнительных действий (в противоположность операции вставки), делает метод `SsTree::delete` тривиальным.

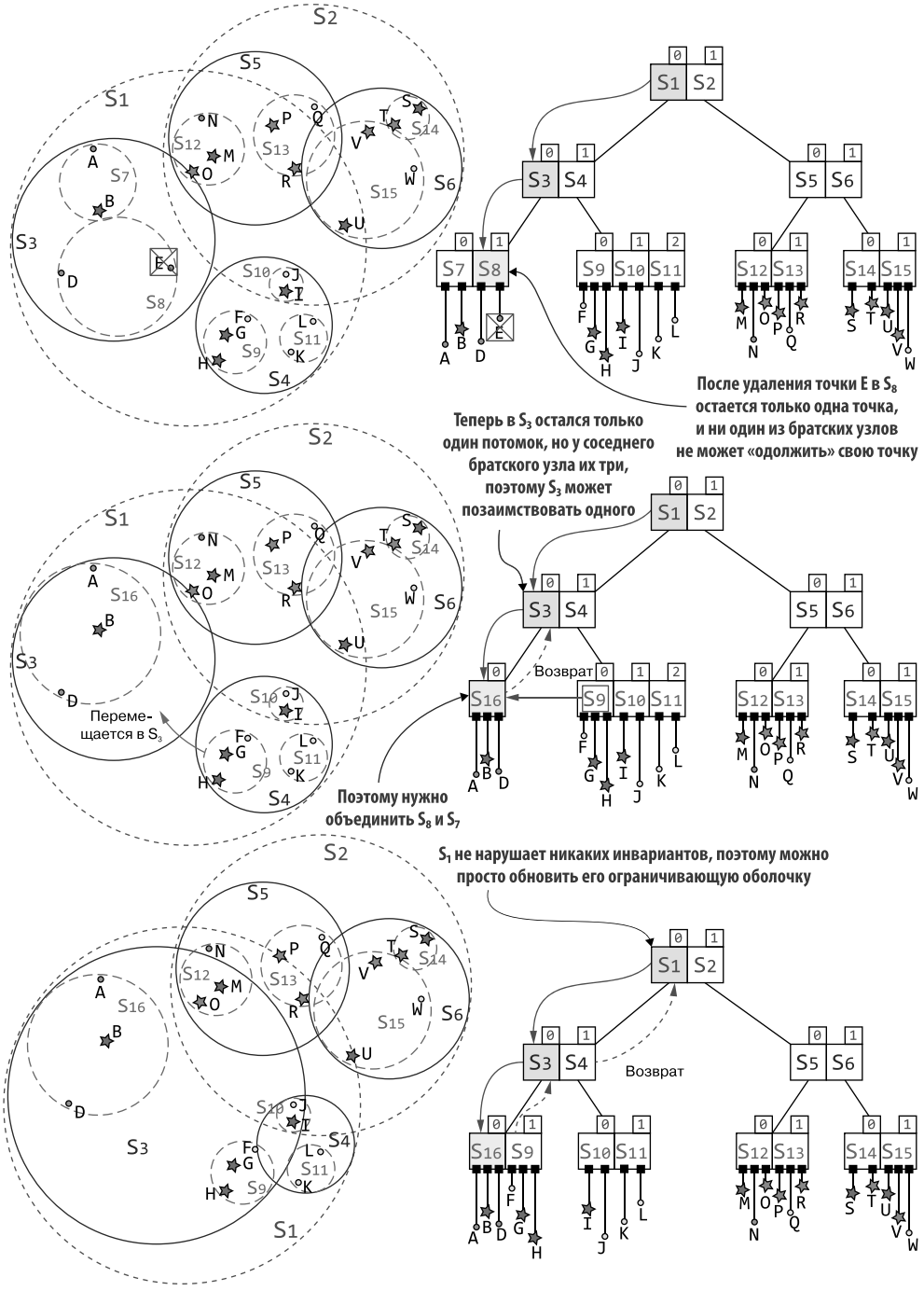
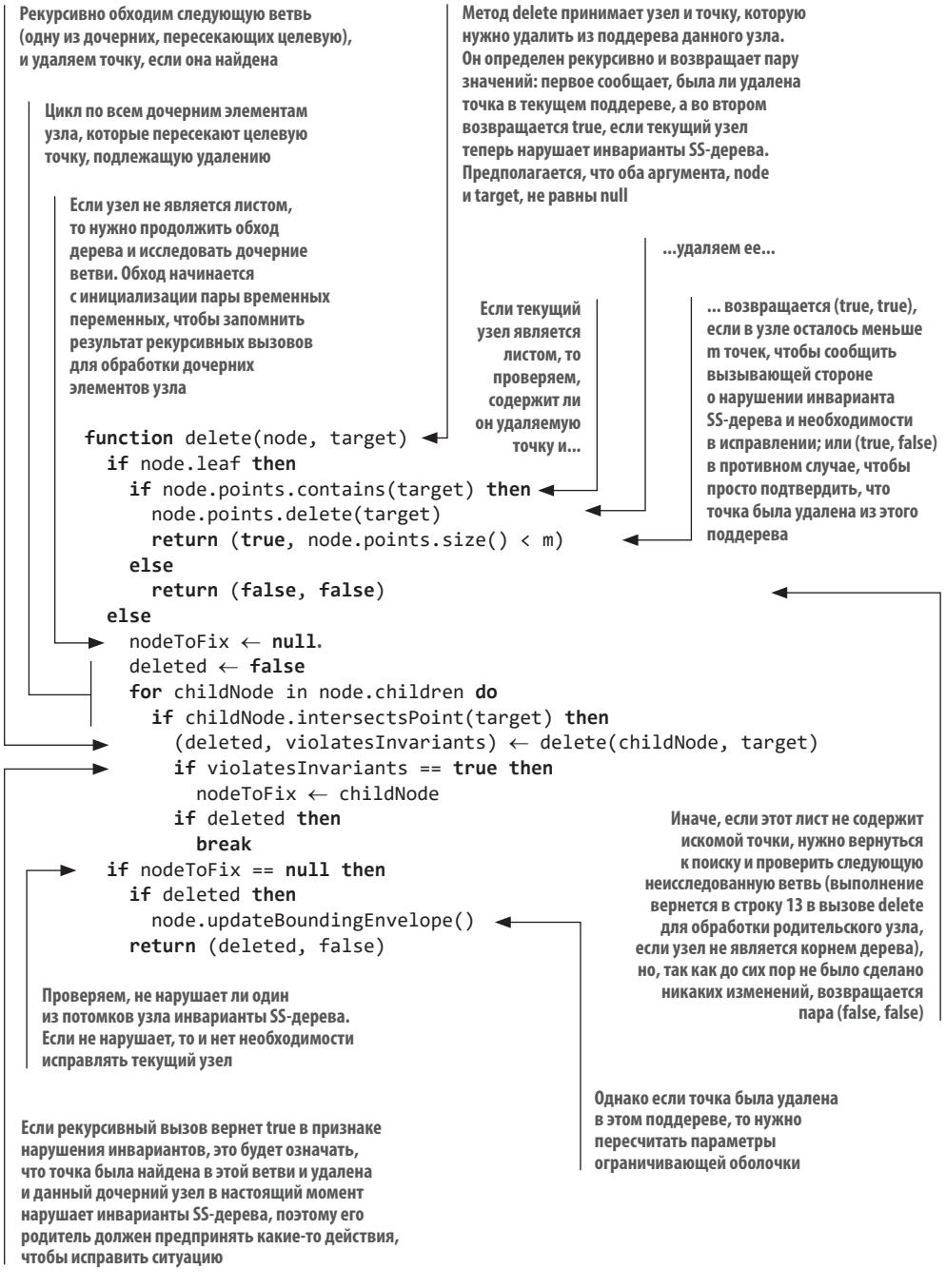
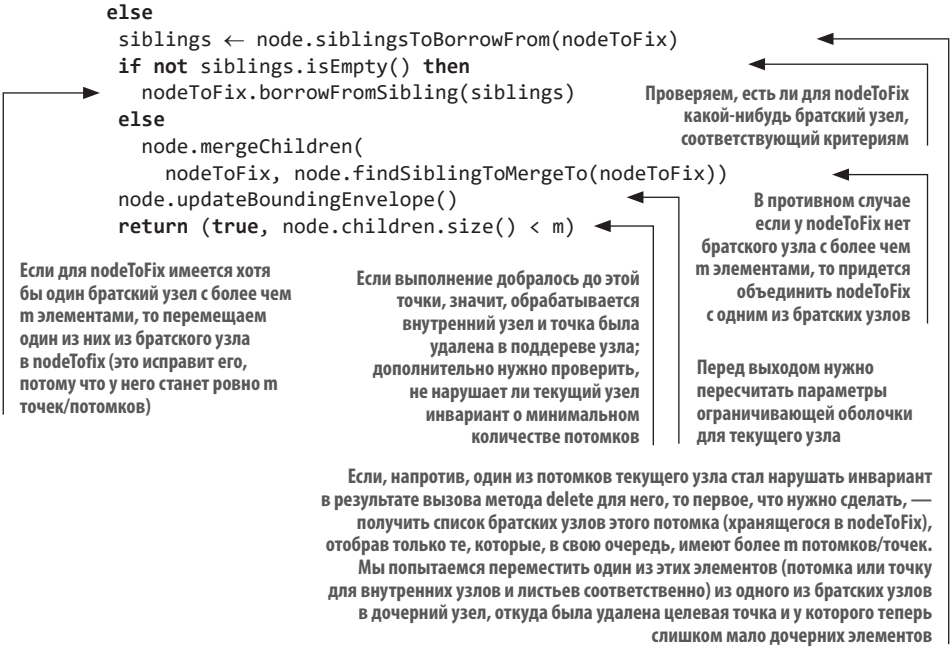


Рис. 10.19. Удаление точки. В этом примере показаны по порядку случаи 2, *в*, 2, *б* и 1, описанные в этом разделе

Но хватит примеров! Пришла пора написать тело метода `delete` (листинг 10.15).

Листинг 10.15. Метод delete





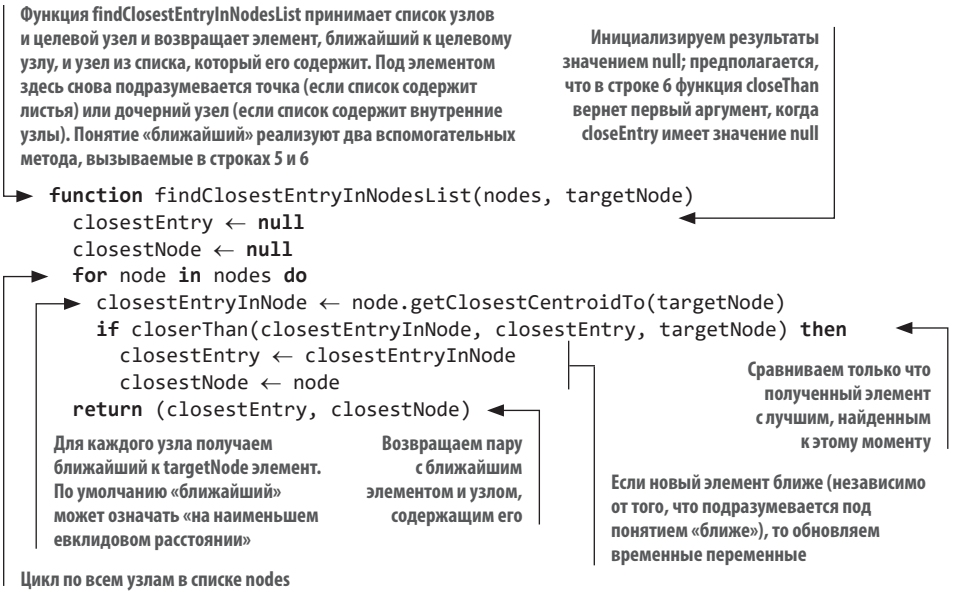
Этот метод так же сложен, как и метод `insert` (возможно, даже сложнее!). По этой причине, подобно методу `insert`, я разбил метод `delete`, использовав несколько вспомогательных функций, чтобы сделать его компактнее и чище. Однако на этот раз подробно рассматривать их все мы не будем.

Методы, включающие поиск чего-то «ближайшего» к узлу, такие как `findSiblingToMergeTo` в листинге 10.15, являются эвристиками, зависящими от используемого понятия «ближе». Как упоминалось в описании работы метода `delete`, у нас на выбор есть несколько вариантов: от кратчайшего расстояния (которое тоже легко реализовать) до наименьшей площади перекрытия.

Для экономии места мне не остается ничего иного, как предоставить возможность читателю самому поработать над ними (включая выбор функции близости). Внимательно прочитав этот и предыдущий разделы, вы сможете самостоятельно реализовать версии, использующие критерий близости на основе евклидова расстояния.

Итак, чтобы завершить описание метода `delete`, обратимся к `findClosestEntryInNodesList`. В листинге 10.16 показан псевдокод этого метода, реализующий еще один линейный поиск в списке узлов с целью найти ближайший элемент, содержащийся в любом из узлов, перечисленных в заданном списке. Обратите внимание, что метод возвращает также родительский узел, потому что он понадобится вызывающей стороне.

Листинг 10.16. Метод findClosestEntryInNodesList



Ниже, в листинге 10.17, показан метод `borrowFromSibling`, перемещающий элемент (точку для листьев или дочерний узел для внутренних узлов) в узел, который в настоящий момент нарушает инвариант минимального числа точек/потомков. При этом метод извлекает элемент из одного из братских узлов. Очевидно, что ему нужно выбрать братский узел, имеющий больше m элементов, чтобы не переместить проблему в другое место (мы же не хотим, чтобы сам братский узел нарушил инвариант, если после перемещения у него окажется на один элемент меньше заданного количества!). Здесь предполагается, что все элементы в переданном списке `siblings` являются непустыми узлами и содержат не менее $m + 1$ элементов. При реализации этого метода на языке, поддерживающем утверждения (assertion), можно добавить утверждение, проверяющее эти условия[1].

Если эти условия выполняются, то производится поиск лучшего элемента для «заимствования». Обычно это означает ближайший элемент к целевому узлу, но, как уже упоминалось, можно использовать и другие критерии. Отыскав его, остается только переместить ближайший элемент и соответствующим образом обновить параметры ограничивающих оболочек узлов.

---

[1] Предполагается, что метод реализуется как приватный. Утверждения могут быть отключены (и часто отключаются в версии для промышленной эксплуатации), поэтому они никогда не должны использоваться для проверки входных значений. Проверка аргументов в приватных методах не лучшее решение, и ее не следует применять для проверки значений, передаваемых пользователем. В идеале утверждения желательно использовать только для инвариантов.

Листинг 10.17. Метод SsNode::borrowFromSibling

Метод borrowFromSibling определен в классе SsNode. Он принимает непустой список узлов, братских для текущего узла, и перемещает элемент, ближайший к текущему узлу, из одного из братских узлов в текущий

Отыскивается в списке братских узлов ближайший к текущему узлу элемент. Здесь определение «ближайший» должно быть задано при разработке структуры данных. Вспомогательная функция вернет как ближайший элемент для перемещения, так и братский узел, содержащий этот элемент

```
function borrowFromSibling(siblings)
  (closestEntry, closestSibling) ←
    findClosestEntryInNodesList(siblings, this)
  closestSibling.deleteEntry(closestEntry)
  closestSibling.updateBoundingEnvelope()
  this.addEntry(closestEntry)
  this.updateBoundingEnvelope()
```

Удаляется выбранный элемент из братского узла, где он содержится в данный момент, и обновляются параметры его ограничивающей оболочки

Добавляем выбранный элемент в текущий узел и обновляем параметры его ограничивающей оболочки

Если условия не выполняются и нет братского узла, извлечение потомка из которого не повлечет нарушения инварианта, то это может означать одно из двух.

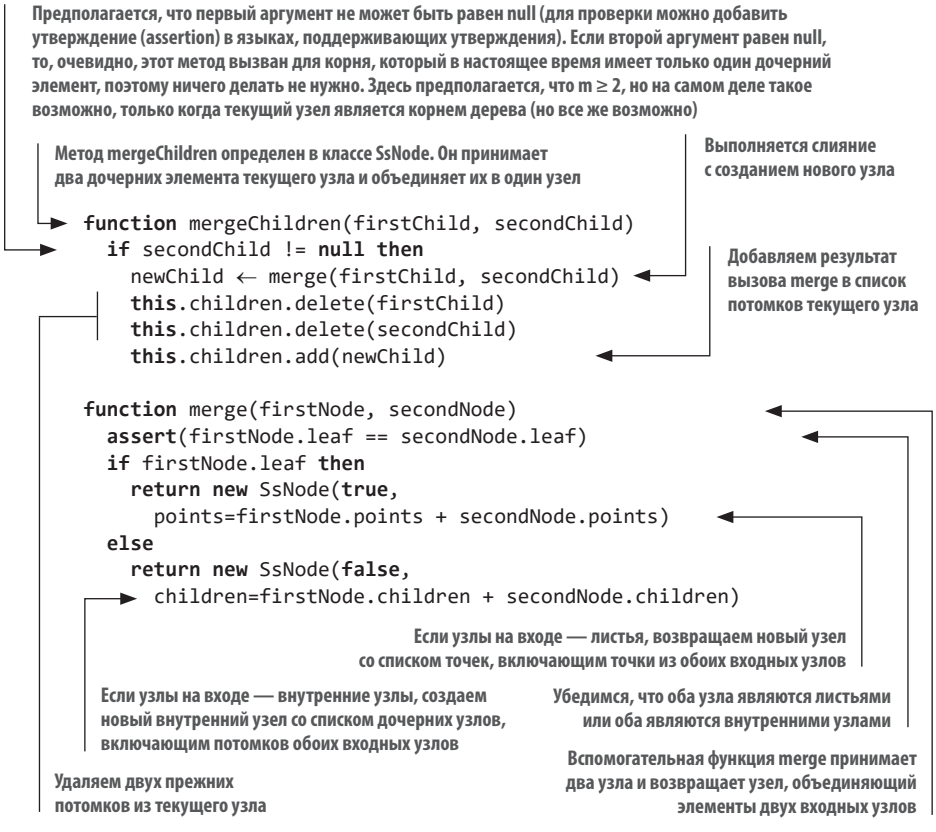
1. Нет братских узлов: если предположить, что $m \geq 2$, это может произойти только в корне, имеющем единственный дочерний элемент. В этом случае ничего делать не нужно.
2. Есть братские узлы, но все они имеют ровно m записей. В этом случае, поскольку $m \leq M/2$, если объединить текущий узел, нарушающий инвариант, с любым из его братских узлов, то получится новый узел с $2m - 1 < M$ элементами — другими словами, допустимый узел, не нарушающий никаких инвариантов.

В листинге 10.18 показано, как обработать обе ситуации: метод `mergeChildren` сравнивает второй аргумент с `null`, чтобы понять, находится он в первом или во втором случае, и если нужно выполнить слияние, то очищает также родительский узел (текущий узел, для которого вызван метод `mergeChildren`).

Это был последний фрагмент псевдокода, которого не хватило для завершения реализации метода `delete`. Однако, прежде чем закончить раздел, я хочу предложить вам еще раз взглянуть на рис. 10.19. Конечным результатом является правильное SS-дерево, не нарушающее ни один из его инвариантов. Но, будем честными, результат не впечатляет, верно? Теперь получилась одна огромная сфера S_3, которая занимает почти все пространство в ограничивающей оболочке своего родителя и имеет значительную площадь перекрытия не только со своим братским узлом, но и с другой ветвью своего родителя.

Выше такая ситуация упоминалась как риск обработки слияний в случае 2, *б*, в описании `delete`. К сожалению, таков типичный побочный эффект слияния узлов и перемещения узлов/точек между братскими узлами. Особенно ограниченным является выбор элемента для перемещения, и для слияния/перемещения можно выбрать только дальний элемент, а, как показано на рис. 10.19, в долгосрочной перспективе после множества удалений это может нарушить балансировку дерева.

Листинг 10.18. Метод SsNode::mergeChildren



Нужно предпринять что-то еще, чтобы сохранить дерево сбалансированным, а его производительность приемлемой. В разделе 10.5 мы рассмотрим одно из возможных решений: деревья SS$^+$.

10.4. ПОИСК ПО СХОДСТВУ

Прежде чем обсуждать способы улучшения балансировки SS-деревьев, я предлагаю завершить обсуждение их методов. Уже было показано, как конструировать такие деревья, но для каких целей их можно использовать? Вы наверняка уже догадались, что одним из основных применений этой структуры данных является поиск ближайшего соседа. Еще одно важное применение SS-деревьев, так же как и *k*-мерных деревьев, — поиск по диапазону. Оба применения, поиск ближайшего соседа и поиск по диапазону, относятся к категории поиска по сходству, то есть поиска в больших многомерных пространствах, когда единственным критерием является сходство между парой объектов.

Как и *k*-мерные деревья, обсуждавшиеся в главе 9, SS-деревья тоже можно (даже более просто) расширить для поддержки приближенного поиска по сходству. В разделе 9.4 отмечалось, что приближенный поиск является возможным решением проблем с производительностью *k*-мерного дерева. Подробный разговор об этих методах пойдет в подразделе 10.4.2.

10.4.1. Поиск ближайшего соседа

Алгоритм поиска ближайшего соседа аналогичен алгоритму, использующему *k*-мерные деревья. Дерево имеет совсем другую структуру, но главное отличие в логике алгоритма — формула, с помощью которой проверяется, пересекает ли ветвь область искомой точки (сферу с центром в искомой точке и с радиусом, равным расстоянию до текущего предполагаемого ближайшего соседа). То есть, если ветвь находится достаточно близко к целевой точке, ее, ветвь, нужно пройти. Кроме того, в отличие от *k*-мерных деревьев, где проверяются и обновляются расстояния в каждом узле, SS-деревья (и R-деревья) содержат точки только в своих листьях, поэтому начальное расстояние будет обновляться только после достижения первого листа.

Для большей эффективности поиска ветви должны проверяться в правильном порядке. Этот порядок неочевиден, но хорошей отправной точкой может стать сортировка узлов по их минимальному расстоянию от искомой точки. Конечно, нет никаких гарантий, что узел, который *потенциально* имеет ближайшую точку (то есть его ограничивающая оболочка находится ближе всего к целевой точке), *на самом деле* будет иметь точку, расположенную достаточно близко к искомой цели. И поэтому нельзя с уверенностью полагать, что эта эвристика даст наилучший возможный порядок обхода. Однако в среднем такой порядок лучше, чем случайный.

Чтобы напомнить, почему это важно, отошлю вас к материалам подраздела 9.3.5, в котором упоминалось, что более точное определение расстояния до ближайшего соседа помогает сократить количество проверяемых ветвей и тем самым способствует повышению производительности поиска. На самом деле если известно, что есть некоторая точка на расстоянии *D* от искомой точки, то можно исключить из поиска все ветви, чьи ограничивающие оболочки находятся от искомой точки на расстоянии большем, чем *D*.

В листинге 10.19 показан код метода `nearestNeighbor`, а на рис. 10.20 и 10.21 — примеры вызова метода для нашего дерева. Как видите, код сам по себе компактный: ему достаточно пройтись по всем ветвям, пересекающим сферу с центром в искомой точке и радиусом, равным расстоянию до текущего ближайшего соседа, и последовательно обновить найденное наилучшее значение.

Эта простота и ясность не являются неожиданными. Мы немало потрудились, проектируя и создавая структуру данных, и теперь можем наслаждаться ее преимуществами!

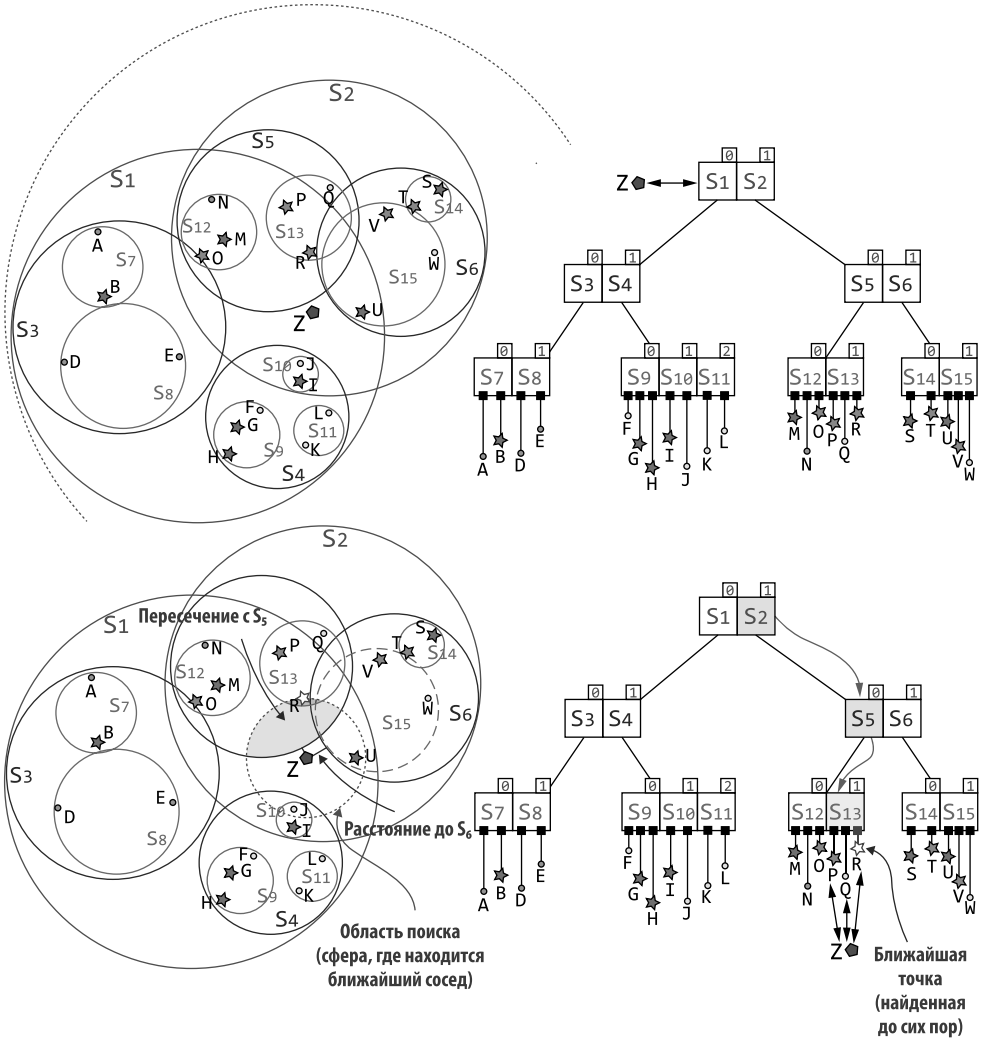


Рис. 10.20. Поиск ближайшего соседа. На этом рисунке показаны первые шаги, выполняемые методом, начиная с корня дерева. *Вверху:* первоначально областью поиска является сфера с центром в точке Z и с бесконечным радиусом (хотя здесь для единообразия она показана в виде окружности, включающей все дерево). *Внизу:* алгоритм обходит дерево, выбирая сначала самые близкие ветви. Граница S_5 ближе, чем граница S_6, поэтому сначала проверяется первая (хотя, как нетрудно заметить, лучше было бы сделать противоположный выбор, но алгоритм не может этого знать). Обычно расстояние вычисляется для ограничивающей оболочки узла, но, поскольку Z пересекает и обе сферы, S_1 и S_2, сначала выбирается та, чей центроид находится ближе. Алгоритм выполняет поиск в глубину, проходя весь путь до листа перед возвратом. Здесь показан путь к первому листу: оказавшись там, можно обновить область поиска, которая теперь является сферой с центром в точке Z и радиусом, равным расстоянию до R, ближайшей точки в S_{13}. Эта точка сохраняется как наилучший предполагаемый ближайший сосед

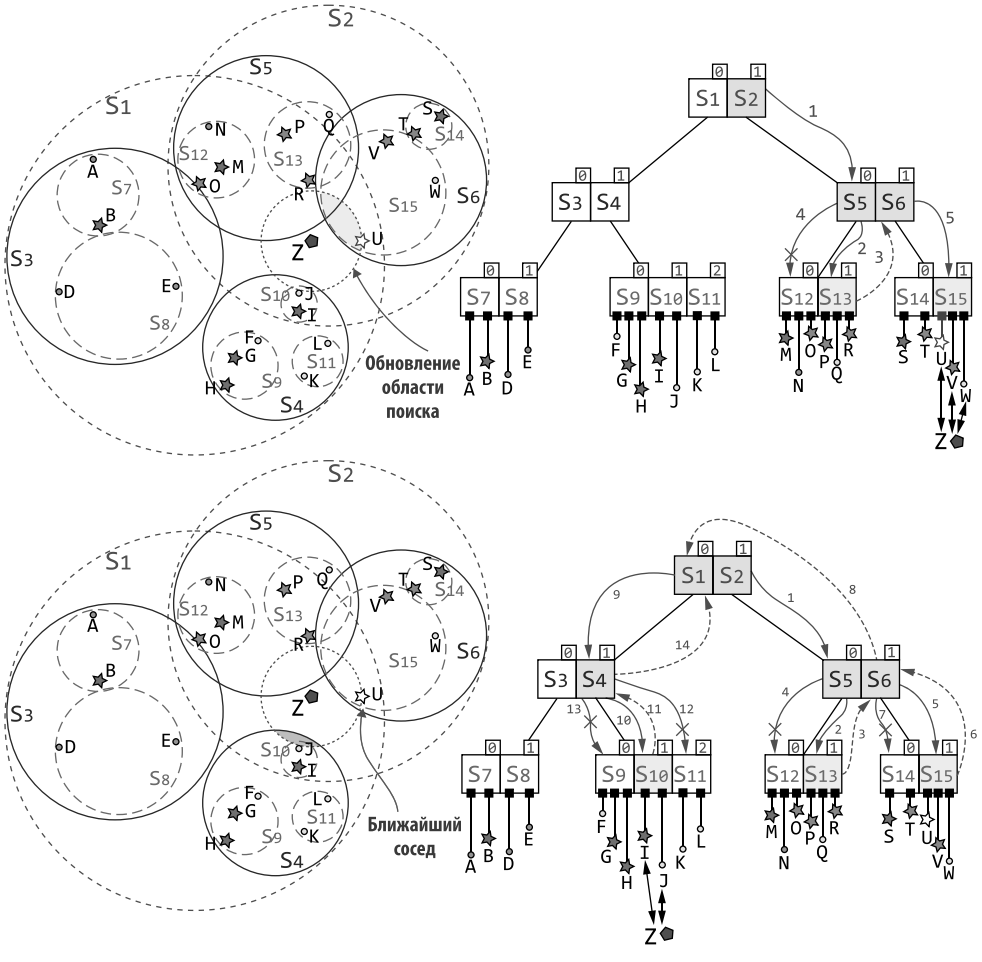
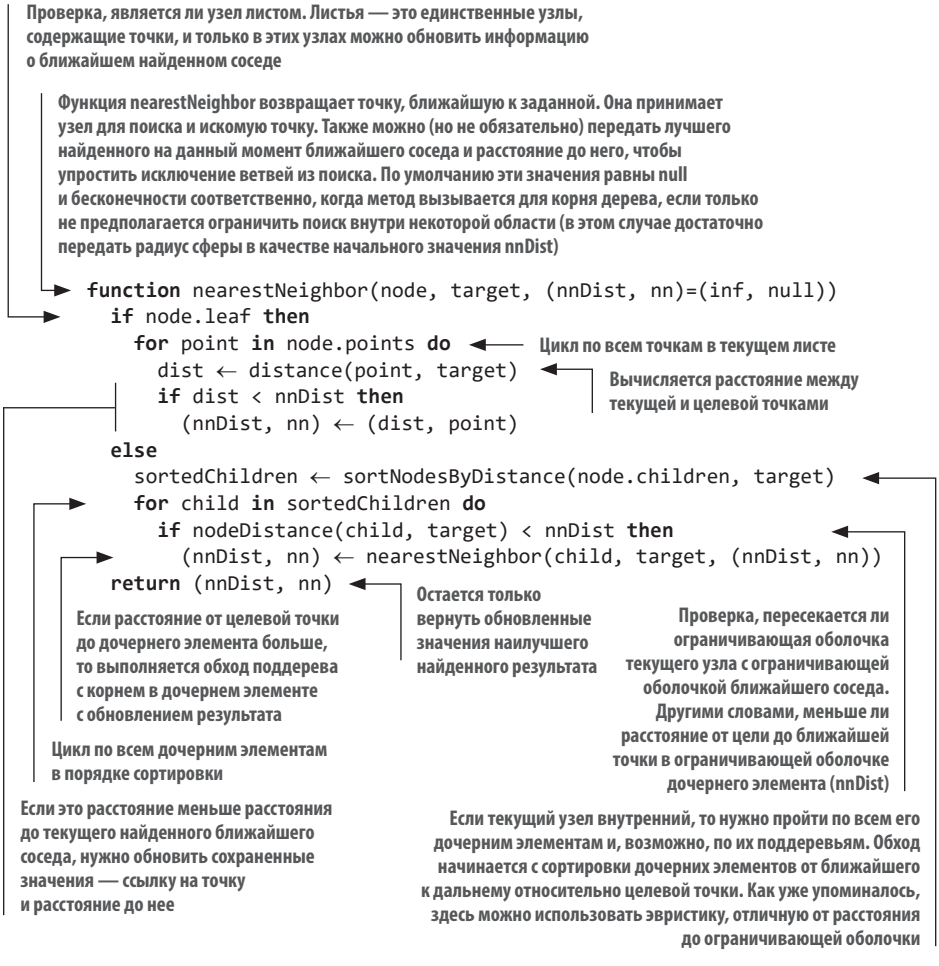


Рис. 10.21. Поиск ближайшего соседа. Краткий обзор следующих шагов в обходе. Стрелки пронумерованы, чтобы показать порядок рекурсивных вызовов. *Вверху:* после посещения S_{13} и обнаружения R как наилучшего предполагаемого ближайшего соседа, S_{12} исключается из поиска, потому что находится за пределами обновленной области поиска. Затем алгоритм возвращается назад и переходит к узлу S_6, братскому для S_6, у которого есть непустое пересечение с областью поиска. *Внизу:* обход ветви до конца. Также нужно обойти ветвь S_1, потому что Z лежит внутри нее. На самом деле область поиска пересекает другой лист S_{10}, поэтому нужно пройти ветвь, ведущую к нему. Как видите, точка J рядом с ближайшей соседкой Z, поэтому вполне вероятно, что мы найдем истинного ближайшего соседа в одном из последующих вызовов

Из вспомогательных методов, используемых в листинге 10.19, важно уделить некоторое время функции `nodeDistance`. Если обратиться к рис. 10.22, то можно увидеть, почему минимальное расстояние между узлом и ограничивающей оболочкой равно

расстоянию между центроидами минус радиус оболочки: это формула расстояния между точкой и сферой, взятая из геометрии.

Листинг 10.19. Метод nearestNeighbor SS-дерева



Алгоритм поиска ближайшего соседа легко расширить, чтобы он возвращал n ближайших соседей. Подобно тому как мы делали это в главе 9, используя k-мерные деревья, нужно просто использовать очередь с ограниченными приоритетами, содержащую не более n элементов, и самую дальнюю из этих n точек при определении расстояния для исключения ветвей из поиска (пока найдено меньше n точек, расстояние отсечения будет равно бесконечности).

Чтобы так же поддерживать поиск в сферических областях, можно добавить в функцию поиска пороговый аргумент, который становится начальным расстоянием отсечения (вместо бесконечности) для `nnDist`. Поскольку эти расширения реализуются

просто, как описывается в главе 9, я оставляю эту задачу читателям (подсказка: загляните в реализацию, находящуюся в репозитории книги на сайте GitHub[1]).

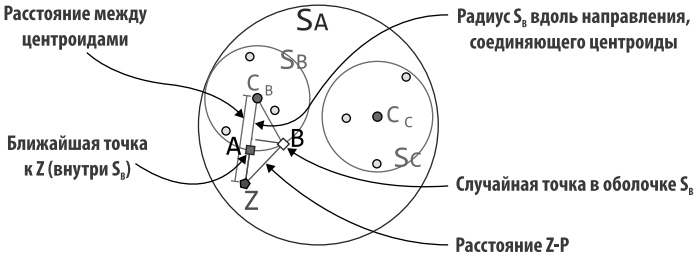


Рис. 10.22. Минимальное расстояние до ограничивающей оболочки. Рассмотрим треугольник ZBC_B. Согласно неравенству треугольника $|C_BB| + |BZ| > |ZC_B|$, но $|ZC_B| == |ZA| + |AC_B|$ и $|AC_B| == |C_BB|$ (оба являются радиусами), поэтому в конечном счете $|C_BB| + |BZ| > |ZA| + |AC_B| \Rightarrow |BZ| > |ZA|$. Следовательно, минимальное расстояние — это длина отрезка от Z до сферы S_B вдоль направления, соединяющего ее центроид с Z

10.4.2. Поиск области

Поиск области похож на таковой, описанный для *k*-мерных деревьев. Единственное отличие — помимо структурных изменений, связанных с тем, что точки хранятся только в листьях, нужно вычислять пересечение между каждым узлом и областью поиска.

В листинге 10.20 показана общая реализация этого метода для SS-деревьев. В ней предполагается, что заданная область включает метод проверки ее пересечения с гиперсферой (ограничивающей оболочкой узла). Подробные разъяснения и примеры поиска для наиболее распространенных типов областей и их алгебраический смысл вы найдете в подразделе 9.3.6.

Листинг 10.20. Метод pointsWithinRegion SS-дерева

Инициализируем возвращаемое значение пустым списком

Функция pointsWithinRegion принимает узел, начиная с которого следует выполнить поиск, а также область поиска. Возвращает список точек, хранящихся в поддереве с корнем в заданном узле и лежащих в области поиска (другими словами, пересечение между областью и ограничивающей оболочкой узла)

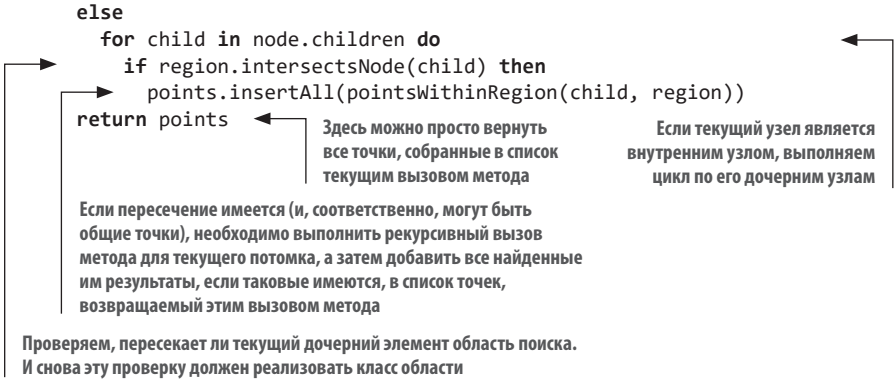
Проверяем, является ли узел листом

Цикл по всем точкам в текущем листе

Если текущая точка находится в области поиска, то добавляем ее в список результатов. Ответственность за выбор правильного метода проверки попадания точки в область лежит на классе области (поэтому области разной формы могут по-разному реализовать этот метод)

```
function pointsWithinRegion(node, region)
    points ← []
    if node.leaf then
        for point in node.points do
            if region.intersectsPoint(point) then
                points.insert(point)
```

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#ss-tree.



10.4.3. Приближенный поиск по сходству

Как уже упоминалось, поиск по сходству в k-мерных деревьях, а также R-и SS-деревьях страдает так называемым *проклятием размерности*: производительность методов этих структур данных падает в экспоненциальной прогрессии с ростом числа измерений пространства поиска. k-мерные деревья страдают и от дополнительных проблем разреженности, которые становятся все более актуальными с ростом числа измерений.

Использование R-и SS-деревьев может улучшить баланс древовидной структуры и привести к созданию более производительных деревьев, но на этом наш арсенал возможностей не исчерпывается, и можно сделать кое-что еще, чтобы улучшить производительность методов поиска по сходству.

Методы приближенного поиска в действительности несут в себе компромисс между точностью и производительностью. Есть несколько различных (а иногда и взаимодополняющих) стратегий, которые можно использовать для ускорения приближенного поиска.

- *Уменьшение размерности объектов* — используя такие алгоритмы, как PCA (Principal Component Analysis — метод главных компонентов) или дискретное преобразование Фурье, можно спроецировать объект из набора данных в другое пространство с более низкой размерностью. Идея состоит в том, что это сокращенное пространство содержит только информацию, необходимую для различения разных точек в наборе данных. С динамическими наборами данных этот метод явно менее эффективен.
- *Уменьшение количества проверяемых ветвей* — в предыдущем разделе использовалась довольно консервативная стратегия исключения ветвей из поиска, согласно которой ветвь подлежит обходу, если есть хоть какой-то шанс (даже малейший), что в ней можно найти точку ближе, чем текущий ближайший сосед. Используя более агрессивную стратегию исключения, можно уменьшить количество просматриваемых ветвей (и в конечном счете точек), если согласиться с тем, что результаты могут быть не самыми точными из возможных.

- *Использование стратегии досрочного прекращения поиска* — в этом случае поиск останавливается, когда текущий результат оценивается как достаточно хороший. Критерием, помогающим решить, что значит «достаточно хороший», может быть порог (например, когда найден ближайший сосед, находящийся ближе некоторого расстояния) или условие остановки, связанное с вероятностью обнаружения лучшего совпадения (например, если ветви посещаются в порядке от ближних к дальним относительно искомой точки, то вероятность найти ближайшего соседа уменьшается с количеством проверенных листьев).

Ниже сосредоточимся на второй стратегии — стратегии уменьшения количества проверяемых ветвей. В частности, рассмотрим метод, который при заданном параметре ε (ε называется *ошибкой аппроксимации*), где $0 \le \varepsilon \le 0{,}5$, гарантирует, что при приближенном поиске n ближайших соседей полученный n-й ближайший сосед будет соответствовать истинному n-му ближайшему соседу в пределах коэффициента $(1 + \varepsilon)$.

Для объяснения алгоритма ограничимся[1] случаем, когда $n == 1$, то есть поиском единственного ближайшего соседа. Предположим, что метод приближенного поиска ближайшего соседа, вызванный для точки P, возвращает точку Q, тогда как истинным ближайшим соседом точки P является другая точка $N \neq Q$. Тогда, если расстояние между P и истинным ближайшим соседом N равно d, расстояние между P и Q не будет превышать $(1 + \varepsilon) \times d$.

Гарантировать это условие легко. При оценке возможности исключения ветви из поиска расстояние между целевой точкой и ограничивающей оболочкой ветви сравнивается не с расстоянием до текущего ближайшего соседа, а с расстоянием до текущего ближайшего соседа, умноженным на $1 / (1 + \varepsilon)$.

Если обозначить искомую точку как Z, текущего ближайшего соседа — как C, а ближайшую точку в ветви, исключенной из поиска, — как A (рис. 10.21), то фактически получим:

$$d(Z, A) \ge \frac{1}{1+\varepsilon} d(Z, C) \Rightarrow \frac{d(Z, C)}{d(Z, A)} \le 1+\varepsilon.$$

Таким образом, если расстояние до ближайшей точки в ветви больше, чем расстояние до текущего ближайшего соседа, умноженное на обратную величину $(1 + \varepsilon)$, то возможный ближайший сосед в этой ветви гарантированно находится не дальше, чем на расстоянии, равном расстоянию до текущего ближайшего соседа, умноженному на эпсилон-фактор.

Конечно, в наборе данных может быть несколько точек, находящихся в пределах расстояния до истинного ближайшего соседа, умноженного на коэффициент $(1 + \varepsilon)$, поэтому нет никаких гарантий, что мы не получим вторую или третью и так далее ближайшие точки.

---

[1] Его легко обобщить до случая n-ближайших соседей, рассматривая расстояние до n ближайших соседей.

Однако вероятность, что эти точки существуют, пропорциональна размеру кольцевой области с радиусами `nnDist` и `nnDist / (1 + ε)`, поэтому чем меньше ε, тем ниже вероятность пропустить более близкие точки.

Более точную оценку вероятности дает площадь пересечения упомянутого кольца с ограничивающей оболочкой узла, исключаемого из поиска. Эту идею иллюстрирует рис. 10.23, где показана разница между SS-, *k*-мерными деревьями и R-деревьями: вероятность максимальна, когда внутренний радиус только касается области, а сфера имеет меньшее пересечение по отношению к любому прямоугольнику.

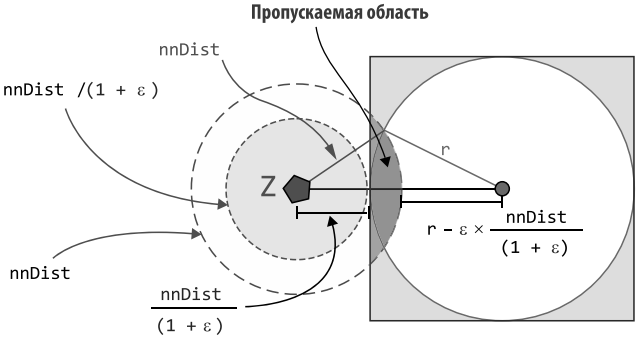


Рис. 10.23. Вероятность пропустить j точек при использовании ошибки аппроксимации пропорциональна пересечению области, исключенной из поиска (кольца с радиусом (ε / 1 + ε)) × nnDist) и ограничивающей оболочки узла, пересекающей эту область. Максимальная вероятность соответствует области, касающейся внутренней сферы. На рисунке показано, насколько меньше пересечение для сферических ограничивающих оболочек, чем для прямоугольных

Установив ε == 0, вернемся к алгоритму точного поиска как частному случаю, потому что `nnDist/(1+epsilon)` сокращается до `nnDist`.

При ε > 0 обход ветвей может выглядеть, как показано на рис. 10.24, где демонстрируется пример, немного отличающийся от примера на рис. 10.20 и 10.21, чтобы показать, как приближенный поиск ближайшего соседа может пропустить истинного ближайшего соседа.

Во многих областях приближенного поиска более чем достаточно. Возьмем приведенный выше пример поиска в наборе изображений: можно ли быть уверенными, что точное совпадение лучше приблизительного? Если бы мы работали с географическими координатами, скажем, на карте, то разница на множитель ε могла бы иметь неприятные последствия (выбор более длинного и дорогостоящего маршрута и, возможно, небезопасного). Но когда задача состоит в том, чтобы найти платья, максимально похожие на просматриваемое, нельзя гарантировать точность даже принятой нами метрики. Возможно, одна пара векторов признаков немного ближе, чем другая, но для человеческого глаза изображения из второй пары будут выглядеть более похожими!

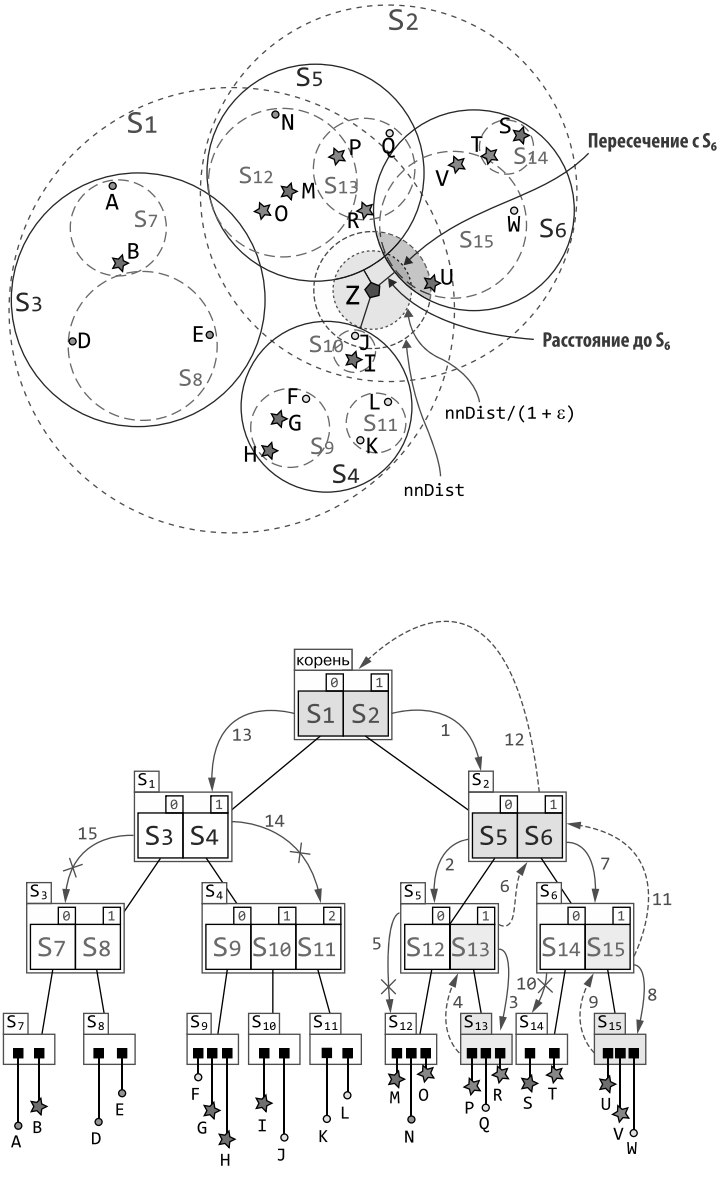


Рис. 10.24. Приближенный поиск ближайшего соседа. Пример подобен (почти идентичен) тому, что изображен на рис. 10.20 и 10.21, для более удобного сравнения. Здесь дерево показано очень подробно, чтобы более понятным образом отобразить путь, которым следует алгоритм. Узел S_4 содержит точку J, истинного ближайшего соседа. Этот узел находится от искомой точки Z чуть дальше, чем S_5 и S_6, и вне аппроксимированной области поиска (значение ε подобрано специально, чтобы вызвать это условие, в этом примере ε ~= 0,25). Стрелки пронумерованы в порядке обхода

Итак, пока ошибка аппроксимации ε не слишком велика, есть вероятность, что результат поиска «наиболее похожего изображения» будет не хуже, а, может быть, и лучше, чем точный результат.

Желающие смогут найти множество литературных источников по теме приближенного поиска по сходству и глубже погрузиться в понятия, которые здесь рассмотрели лишь поверхностно. В качестве отправной точки я предлагаю замечательную работу Джузеппе Амато (Giuseppe Amato)[1].

10.5. ДЕРЕВЬЯ SS+[2]

До сих пор мы использовали исходную структуру SS-дерева, описанную в статье Уайта и Джейна[3]. Цель разработки SS-деревьев — уменьшить перекрытия узлов и количество листьев, которые нужно обойти при поиске, по сравнению с R- и *k*-мерными деревьями.

10.5.1. SS-деревья лучше?

Основное преимущество этой новой структуры данных, по сравнению с *k*-мерными деревьями, в том, что все листья находятся на одной высоте, настолько она самобалансирующаяся. Кроме того, использование ограничивающих оболочек вместо разделений, параллельных одной из осей, ослабляет проклятие размерности, потому что одномерное разделение позволяет разделить пространство поиска только в одном направлении за раз.

R-деревья также используют ограничивающие оболочки, но другой формы: гиперпрямоугольники вместо гиперсфер. Для хранения гиперсфер требуется меньше памяти и точное расстояние до них вычисляется быстрее, зато гиперпрямоугольники могут расти асимметрично в разных направлениях: это позволяет охватывать меньший объем, тогда как гиперсферы, будучи симметричными во всех направления, обычно охватывают большие объемы пространства, не содержащие каких-либо точек. И действительно, если сравнить рис. 10.4 и 10.8, то можно заметить, что прямоугольные ограничивающие оболочки точнее соответствуют данным, чем сферические оболочки SS-дерева.

С другой стороны, можно доказать, что разложение по сферическим областям минимизирует число листьев, просматриваемых во время поиска[4].

---

1 *Amato G.* Approximate similarity search in metric spaces. Diss. Technical University of Dortmund, Germany, 2002.

2 Этот раздел содержит преимущественно теоретический материал.

3 *White D. A., Jain R.* Similarity indexing with the SS-tree // Proceedings of the Twelfth International Conference on Data Engineering. IEEE, 1996.

4 *Cleary J. G.* Analysis of an algorithm for finding nearest neighbors in Euclidean space. ACM Transactions on Mathematical Software (TOMS), 1979. — Vol. 5.2 — P. 183–192.

Если сравнить рост объемов сфер и кубов, определяемых следующими формулами:

$$V_{\text{куб}} = r^k;\ V_{\text{сфера}} = \frac{r^k \pi^{k/2}}{(k/2)!},$$

то можно заметить, что для k-мерных пространств при различных значениях k сферы растут медленнее, чем кубы, это также показано на рис. 10.25.

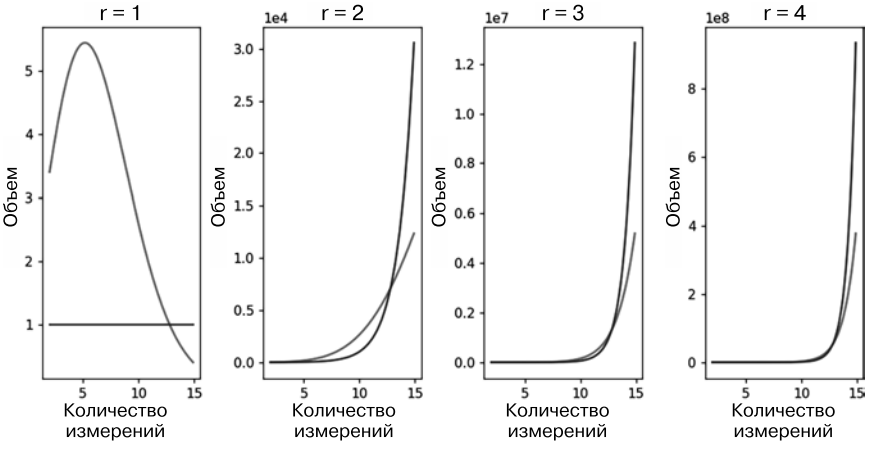


Рис. 10.25. Объем сфер (более светлая линия) и кубов (более темная линия) для различных радиусов в зависимости от количества измерений пространства поиска

И если группа точек равномерно распределена по всем направлениям и имеет форму сферического кластера, то ограничивающая оболочка в форме гиперсферы будет тратить впустую наименьший объем, как можно видеть на примере для двумерного пространства на рис. 10.23, где окружность радиуса r вписана в квадрат со стороной $2r$. Если точки распределены в форме круглого кластера, то все области между кругом и описывающим его квадратом (выделены на рис. 10.23) потенциально не содержат точек и, следовательно, расходуются впустую.

Эксперименты подтверждают, что SS-деревья, использующие сферические ограничивающие оболочки, лучше подходят для наборов данных, равномерно распределенных по всем направлениям, тогда как прямоугольные оболочки лучше подходят для наборов данных с асимметричным распределением.

Ни R-деревья, ни SS-деревья не могут предложить логарифмические верхние границы времени выполнения своих методов в наихудшем случае. В худшем случае (маловероятном, но возможном) придется проверить все листья дерева, которых не более n/m. Это означает, что для выполнения каждой из основных операций с этими структурами данных может потребоваться время, линейно зависящее от размера набора данных. В табл. 10.1 приводится сравнение времени работы методов SS- и k-мерных деревьев.

Таблица 10.1. Операции, поддерживаемые k-мерными деревьями и стоимость их выполения на сбалансированном k-мерном дереве с n элементами

Операция	k-мерное дерево	R-дерево	SS-дерево
`Search`	O(log(*n*))	O(*n*)	O(*n*)
`Insert`	O(log(*n*))	O(*n*)	O(*n*)
`Remove`	$O(n^{1-1/k})^*$	O(*n*)	O(*n*)
`nearestNeighbor`	$O(2^k + \log(n))^*$	O(*n*)	O(*n*)
`pointsInRegion`	O(*n*)	O(*n*)	O(*n*)

* Амортизированное для *k*-мерного дерева, содержащего *k*-мерные точки.

10.5.2. Смягчение недостатков гиперсфер

Теперь попробуем ответить на вопрос: можно ли как-то смягчить недостатки использования сферических ограничивающих оболочек, чтобы воспользоваться их преимуществами при работе с симметричными наборами данных и минимизировать недостатки при работе с асимметричными наборами?

При работе с асимметричными наборами данных вместо сфер можно использовать эллипсоиды, чтобы кластеры могли расти независимо в каждом направлении. Однако это усложнит поиск, потому что придется вычислять радиус вдоль направления, соединяющего центроид с искомой точкой, которая в общем случае не будет лежать ни на одной из осей.

Другой подход к уменьшению объема неиспользуемой области — попытаться уменьшить объем ограничивающих сфер. До сих пор всегда использовались сферы с центром, который является центроидом группы точек, и с радиусом, равным расстоянию от центроида до самой дальней точки, чтобы сфера охватывала все точки в кластере. Такая сфера, однако, не самая маленькая из возможных и охватывающих все точки. На рис. 10.26 показан пример разницы между сферами с минимальным радиусом и минимальным охватом.

Однако вычисление этой наименьшей объемлющей сферы в пространствах с более высокими измерениями невозможно, потому что сложность алгоритма вычисления точных значений центра (и радиуса) гиперсферы растет экспоненциально в зависимости от количества измерений.

Впрочем, есть возможность вычислить аппроксимацию наименьшей объемлющей сферы, начав с центроида кластера в качестве начального приближения. Если говорить в общих чертах, то алгоритм аппроксимации пытается в каждой итерации переместить центр ближе к самой дальней точке в наборе данных. После каждой итерации максимальное расстояние, на которое может перемещаться центр, уменьшается, ограничиваясь интервалом предыдущего обновления, что обеспечивает сходимость.

Мы не будем углубляться в этот алгоритм, но желающие могут больше узнать о нем, начав, например, со статьи Фишера и его коллег[1].

А теперь перейдем к другому способу улучшения балансировки дерева: уменьшению перекрытия узлов.

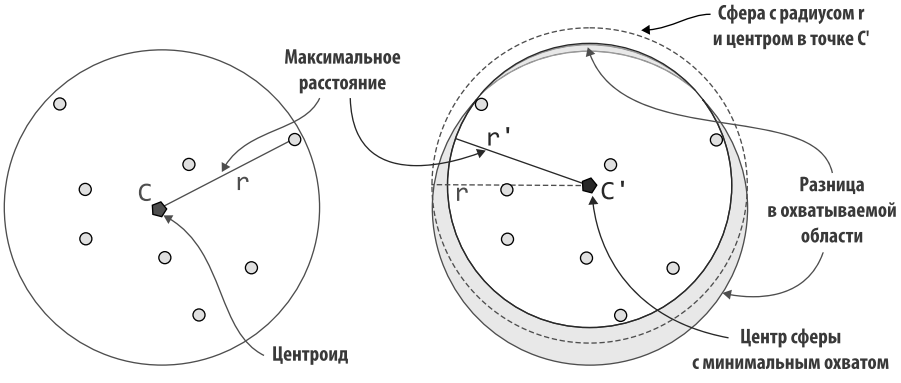


Рис. 10.26. Разница между сферами с минимальным радиусом и центром в центроиде кластера *(слева)* и сферой с минимальным охватом *(справа)* для одного и того же набора точек

10.5.3. Улучшение эвристики разделения

В разделе 10.3 было показано, что разделение и слияние узлов, а также «заимствование» точек/потомков у братских узлов может привести к появлению асимметричных кластеров с ограничивающими оболочками большего размера, чем необходимо, которые имеют увеличенную площадь перекрытия с другими узлами.

Чтобы противодействовать этому эффекту, Курниавати (Kurniawati) с коллегами в своей работе[2] по деревьям SS$^+$ представили новую эвристику разделения, которая не просто делит точки по направлению максимальной дисперсии, но пытается найти две группы, объединяющие ближайшие точки.

Для этого используется вариант алгоритма кластеризации k-средних с двумя ограничениями:

- количество кластеров фиксировано и равно 2;
- максимальное количество точек на кластер привязано к M.

Более подробно о кластеризации и методе k-средних рассказывается в главе 12, поэтому желающие увидеть детали реализации могут обратиться сразу к разделу 12.2.

---

1 *Fischer K., Gärtner B., Kutz M.* Fast smallest-enclosing-ball computation in high dimensions // European Symposium on Algorithms. — Berlin; Heidelberg: Springer, 2003.

2 SS$^+$ tree: an improved index structure for similarity searches in a high-dimensional feature space // Storage and Retrieval for Image and Video Databases *V*. — Vol. 3022. — International Society for Optics and Photonics, 1997.

Время выполнения не более *j* итераций алгоритмом кластеризации методом *k*-средних на наборе данных с *n* точками составляет O(*jkn*)[1], где *k* — количество центроидов.

Поскольку для эвристики разделения имеем *k* == 2, а количество точек в узле, подлежащем разделению, равно *M* + 1, время выполнения становится O(*jdM*). Сравните его с O(*dM*), которое мы имели для первоначальной эвристики разделения, описанной в разделе 10.3. Таким образом, управляя количеством итераций *j*, можно выбирать между качеством результата и производительностью.

На рис. 10.27 показано, насколько эффективной может быть эта эвристика, и становится ясно, почему увеличение времени выполнения стоит того.

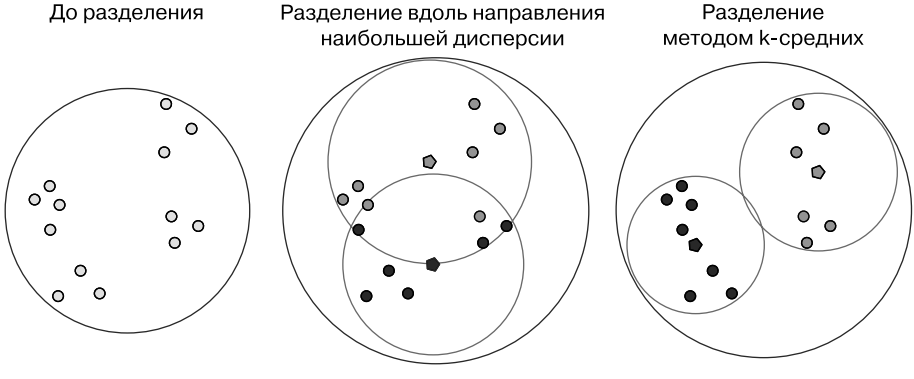


Рис. 10.27. Пример различных вариантов разделения с использованием эвристики в первоначальной версии SS-дерева *(в центре)* и эвристики на основе метода k-средних *(справа)*. Исходная эвристика выбрала направление максимальной дисперсии вдоль оси Y. В этом примере предполагается, что M == 12 и m == 6

За прошедшие годы были разработаны другие, более сложные алгоритмы кластеризации (это будет показано в главе 12, на примере алгоритмов DBSCAN и OPTICS), тем не менее метод *k*-средних по-прежнему идеально подходит для SS-деревьев, потому что он естественным образом создает сферические кластеры с центром в точке центра масс точек в кластере.

10.5.4. Уменьшение площади перекрытия

Как упоминалось в начале предыдущего раздела, эвристика разделения на основе метода *k*-средних — мощный инструмент, помогающий уменьшать площадь перекрытия узлов, поддерживать сбалансированность дерева и выполнять быстрый поиск.

---

[1] Для большей точности необходимо также учесть, что вычисление каждого расстояния требует O(*d*) шагов, поэтому, если *d* может меняться, время выполнения становится равным O(*djkn*). Учитывая то, как ведут себя алгоритмы SS-деревьев с увеличением размерности пространства, новость о линейной зависимости, безусловно, является хорошей, ободряющей. Можно позволить себе опустить этот член (в соответствии с соглашениями, учитывая фиксированную размерность) без искажения результата.

Тем не менее удаление точек с последующим слиянием или перемещением точки/потомка между братскими узлами может разбалансировать узел. Более того, иногда перекрытие, ставшее результатом нескольких операций над деревом, может включать более чем два узла или даже несколько уровней.

Наконец, эвристика разделения методом *k*-средних не использует минимизацию площади перекрытия как критерий и из-за внутренних особенностей алгоритма может давать результаты, в которых узел с большей дисперсией полностью охватывает меньший узел.

Чтобы проиллюстрировать эти ситуации, в верхней половине рис. 10.28 показаны несколько узлов и их родителей со значительным перекрытием.

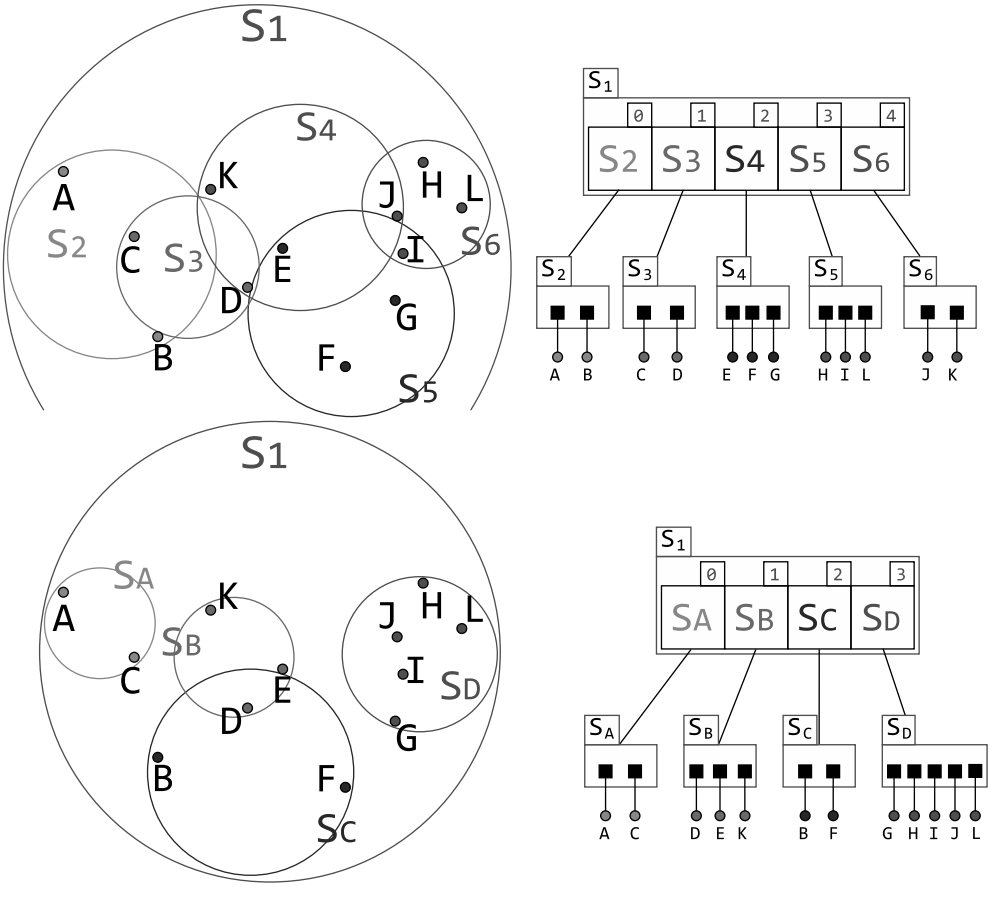


Рис. 10.28. Пример использования эвристики разделения по внукам, способной уменьшить площадь перекрытия узлов. SS-дерево в примере имеет параметры m == 2 и M >= 5. Эвристика разделения методом k-средних выполняется в точках A — L с k == 5. Обратите внимание, как метод k-средних может (а иногда и будет) давать меньше кластеров, чем исходное количество центроидов. В этом случае было получено только четыре кластера

Для решения этой проблемы в деревьях SS+ вводятся два новых элемента.

1. Проверка для обнаружения таких ситуаций.
2. Новая эвристика, применяющая метод *k*-средних ко всем внукам узла *N* (независимо от того, являются они точками или другими узлами; в последнем случае их центроиды будут использоваться для кластеризации) и заменяющая созданными кластерами дочерние элементы узла *N*.

Для проверки перекрытия прежде всего нужна формула вычисления объема пересечения двух сфер. К сожалению, точное вычисление пересечения двух гиперсфер в *k*-мерном случае требует не только значительной работы и хороших математических навыков для вывода формул, но и надежных вычислительных ресурсов. Дело в том, что в результате получается формула, включающая интеграл, вычисление которого стоит достаточно дорого, и заставляет усомниться в собственной полезности в эвристике.

Альтернативное решение — проверить, полностью ли входит одна из двух ограничивающих оболочек в другую. Для этого нужно оценить, попадает ли центр одной сферы в пределы другой, и находится ли центр меньшей сферы от центра большей сферы на расстоянии, меньшем чем $R-r$, где R — радиус большей сферы, а r — радиус меньшей сферы.

Варианты этой проверки могут задавать порог отношения фактического расстояния между двумя центроидами к $R-r$ и использовать его для приближенной оценки объема пересечения по мере приближения этого отношения к 1.

Если условие проверки выполняется, то далее применяется эвристика реорганизации. Чтобы вникнуть в ее детали, необходимо хорошее понимание метода *k*-средних, поэтому я сразу порекомендую читателям обратиться к главе 12, где приводится описание этого алгоритма кластеризации. А сейчас рассмотрим лишь небольшой пример, иллюстрирующий работу эвристики.

В примере эвристика вызывается для узла S_1, а кластеризация выполняется на его внуках, точках $A-L$. Как уже упоминалось, внуками могут быть центроиды других внутренних узлов — алгоритм от этого не изменится.

Результат показан в нижней части на рис. 10.28. Возможно, вас удивит, что в узле S_1 осталось всего четыре дочерних элемента. Дело в том, что, даже если метод *k*-средних применяется с *k*, числом начальных кластеров, равным пяти (по количеству дочерних элементов в S_1), этот алгоритм кластеризации может вернуть меньше чем *k* кластеров. Если на втором шаге (назначения точек) одному из центроидов не будет назначено ни одной точки, этот центроид просто удаляется и алгоритм продолжает работу с количеством кластеров на один меньше.

И проверка, и эвристика реорганизации задействуют значительные ресурсы. Последняя, в частности, требует $O(jMk)$ сравнений/присваиваний, где j — это максимальное количество итераций в методе *k*-средних. Поэтому в оригинальной статье рекомендовалось проверять ситуацию перекрытия после разделения узлов, но применять реорганизацию как можно реже.

Проверку также можно выполнять при перемещении элементов между братскими узлами, но при объединении двух узлов она становится менее интуитивной. В этом случае всегда можно анализировать все объединяемые пары братских узлов или — чтобы ограничить затраты — просто выбрать несколько пар.

Чтобы ограничить количество применений эвристики реорганизации и избежать ее повторного запуска на узлах, которые недавно были повторно кластеризованы, можно ввести порог, определяющий минимальное количество точек, добавляемых/удаляемых из поддерева с корнем в каждом узле, и заниматься реорганизацией потомков узла только при превышении этого порога. Эти методы способствуют уменьшению дисперсии дерева, создавая более компактные узлы.

Завершить обсуждение этих вариантов я хотел бы советом: начинайте с реализации простых SS-деревьев (к настоящему моменту вы вполне готовы создавать свою версию), затем проведите профилирование в своем приложении (как мы это сделали для кучи в главе 2), и только если SS-деревья окажутся узким местом и вы решите, что улучшение их производительности сократит время выполнения вашего приложения как минимум на 5–10 %, то попробуйте использовать одну или несколько эвристик деревьев SS^+, представленных в этом разделе.

РЕЗЮМЕ

- Для преодоления проблем, свойственных k-мерным деревьями, были созданы альтернативные структуры данных, такие как R- и SS-деревья.
- Выбор наиболее подходящей структуры данных зависит от особенностей набора данных и его использования: размерности набора данных, распределения (формы) данных, от того, является ли набор данных статическим или динамическим, а также от того, насколько часто в приложении требуется выполнять поиск.
- R- и SS-деревья не гарантируют уменьшения времени выполнения в наихудшем случае, но на практике во многих ситуациях они работают лучше, чем k-мерные деревья, особенно для многомерных данных.
- Деревья SS^+ улучшают структуру этих деревьев за счет использования эвристики, уменьшающей количество перекрывающихся узлов.
- Используя приближенный поиск по сходству, можно пожертвовать качеством результатов поиска в пользу производительности.
- Во многих областях точные результаты поиска неважны, потому что есть возможность допустить определенную погрешность или нет строгой меры подобия, гарантирующей, что точный результат будет лучшим выбором.
- Примером такой области является поиск по сходству в наборе изображений.

11 Применение поиска ближайшего соседа на практике

В этой главе

- ✓ Адаптация абстрактных алгоритмов к сложности реальных систем.
- ✓ Решение задачи «ближайшего склада» с использованием поиска ближайшего соседа.
- ✓ Добавление фильтрации к поиску ближайшего соседа для поддержки бизнес-логики и гибкого снабжения.
- ✓ Работа в сетях, допускающих сбои и развертывание в реальном мире.
- ✓ Применение поиска ближайшего соседа в других областях, таких как физика и компьютерная графика.

Пришло время воспользоваться знаниями, полученными к этому моменту, и приступить к решению задач, обсуждавшихся в нескольких предыдущих главах. Как всегда, после изучения теории я попытаюсь представить вашему вниманию «практическую» точку зрения, и в этой главе мы вместе реализуем решение задачи «поиска самого близкого склада», учитывающее многие нюансы, с которыми можно столкнуться в реальности. Но я призываю не возлагать слишком больших надежд — мы не сможем рассмотреть все возможные ситуации, и эта глава не претендует на исчерпывающее описание полного спектра проблем, с которыми можно столкнуться. Кроме того, перед вами не руководство по реализации приложений электронной коммерции. Только практика, только поиск решений методом проб и ошибок могут научить вас, как действовать.

В этой главе, помимо нескольких примеров описания действительно существующих проблем, вы найдете практический пример процесса, ведущего от анализа проблем к решению «на бумаге» и далее к рабочему приложению, способному справиться со сложными задачами.

11.1. ПРИЛОЖЕНИЕ: ПОИСК БЛИЖАЙШЕГО СКЛАДА

Теперь, получив представление о возможных способах реализации *k*-мерных деревьев (см. главу 9) и SS-деревьев (см. главу 10), можно сосредоточиться на задачах, решаемых с помощью этих структур данных, и вернуться к нашему первоначальному примеру: разработке приложения для поиска склада/магазина, ближайшего к покупателю, в режиме реального времени. На рис. 11.1 продемонстрирован еще один вариант карты, впервые представленной на рис. 8.1. На этой карте показано несколько (реальных) городов и воображаемых магазинов вокруг них. Согласно задаче необходимо найти ближайший склад или магазин, откуда можно доставить определенный товар покупателю после того, как тот оформит заказ на вашем сайте. По этой причине показаны только виды товаров, продаваемых в каждом магазине, но не названия этих магазинов.

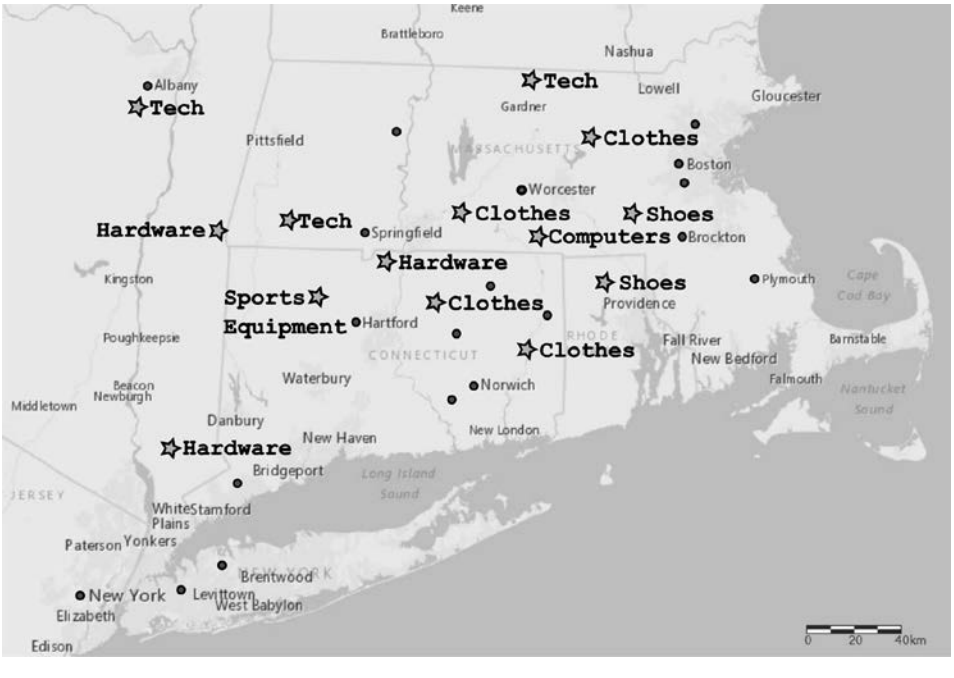


Рис. 11.1. Карта[1] с городами и гипотетическими складами (в этой главе они часто будут называться также магазинами) и покупателями (точки). Для магазинов на карте для простоты указываются только продаваемые ими товары

[1] Источники: Esri, DeLorme; HERE; MapmyIndia.

Учитывая работу, проделанную в предыдущих главах, здесь мы могли бы обойтись вообще без кода. *k*-мерное дерево, SS-дерево или другая подобная структура данных способны позаботиться о выполнении всех необходимых нам операций, нужно лишь создать контейнер с полным списком складов, а затем для каждого заказа выполнить поиск ближайшего склада в этом контейнере.

Если, однако, запасы товара на складе меняются динамически и могут заканчиваться в одном месте или пополняться в другом, то текущая модель работать не будет и в реализацию ее узлов и метода поиска нужно внести небольшое изменение. Поговорим об этих деталях далее в этой главе. Но прежде посмотрим, как реализовать базовое решение нашей задачи электронной коммерции.

11.1.1. Набросок решения

Сначала сделаем некоторые допущения, чтобы упростить сценарий. Предположим, что во всех магазинах все товары всегда есть в наличии. На рис. 11.2 показан возможный процесс обработки заказов в этом сценарии; конечно, он далек от реальной жизни, но упрощение задачи обычно помогает набросать первое решение, которое можно развивать, вводя одно за другим ограничения, встречающиеся в реальных ситуациях, а также рассуждая об их влиянии на приложение и способы их преодоления:

- запасы товаров в магазинах динамически меняются, поэтому нужно проверить, действительно ли ближайший к покупателю магазин продает требуемый товар и имеется ли он на его складе;
- стоимость доставки и другие факторы могут привести к тому, что предпочтительнее окажется везти товар из магазина, более далекого от покупателя;
- если потребуется выполнять HTTP-запросы в процессе поиска, то нужно проявлять особую осторожность, чтобы избежать проблем с ошибками в сети, и, возможно, более тщательно выбирать алгоритм, чтобы справиться с такими проблемами, как недостаточная отказоустойчивость системы и тайм-ауты.

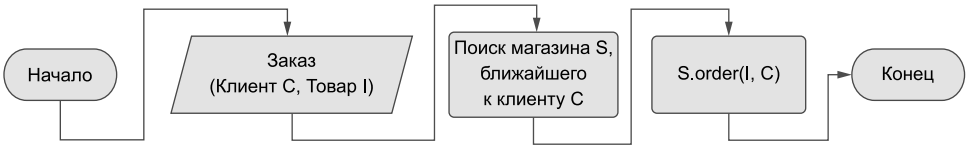


Рис. 11.2. Упрощенный процесс оформления заказа на сайте электронной коммерции. После получения заказа нужно найти ближайший магазин продавца (предполагается, что на складе этого магазина запрошенный товар всегда есть в наличии), а затем разместить заказ у него. Поскольку предполагается, что все звенья цепи работают безотказно, процесс получился линейным (и практически нереалистичным)

Проблема заключается в поддержании постоянно обновляемого списка магазинов и выборе из них ближайшего к покупателю, когда тот размещает заказ на веб-сайте.

В листинге 11.1 определен вспомогательный тип, представляющий магазины, а в листинге 11.2 показаны вышеназванные тривиальные операции.

Листинг 11.1. Класс Shop

```
class Shop
  #type string
  shopName
  #type Array<Item>
  items
  #type tuple(k)
  point

  function Shop(shopName, location, items)
  function order(item, customer)
```

Конструктор класса

Осуществляет фактическую покупку. (Детали этого метода не получится обобщить, и они не имеют отношения к настоящему обсуждению. Просто представьте, что он обновляет запасы и запускает процесс фактической отправки товара покупателю)

Этот класс инкапсулирует информацию о розничном продавце. Ниже в главе, когда мы приступим к разработке окончательного решения, эта информация станет особенно полезной. В поле `point` хранится местоположение магазина, чтобы обеспечить согласованность с тем, что было показано в предыдущих главах.

Листинг 11.2. Методы addShop и BuyItem

Метод addShop принимает корень дерева, имя магазина, его местоположение (координаты точки на плоскости) и, возможно, список товаров

Создаем новый экземпляр класса Shop

```
function addShop(treeRoot, shopName, shopLocation, items=[])
  shop ← new Shop(shopName, shopLocation, items)
  if treeRoot == null then
    return new Node(shop, null, null, 0)
  else
    return insert(treeRoot, shop, treeNode.level + 1)

function buyItem(treeRoot, customerLocation, item)
  closestShop ← nearestNeighbor(treeRoot, customerLocation)
  closestShop.order(item, customerLocation)
```

Если дерево пустое...

...то просто создаем новый корень

Иначе добавляем магазин в дерево (для этого метод insert создаст новый экземпляр Node)

Находим ближайший к покупателю магазин и сохраняем его во временной переменной closestShop

Передаем заказ этому магазину

Метод, вызываемый, когда клиент покупает товар

Из этих фрагментов кода видно, что потребуется внести некоторые коррективы в объекты узлов и в API методов, которые были определены в предыдущих главах.

Например, класс `Node` должен содержать ссылку на магазин, как показано в листинге 11.3, а не на точку, описывающую местоположение, и такие методы, как `insert`, тоже должны принимать экземпляр `Shop`, а не просто точку, поэтому новая сигнатура будет выглядеть примерно так:

```
function insert(node, newShop, level=0)[1]
```

[1] Я не буду приводить здесь полную реализацию, которая отличается от показанной в главах 9 и 10 лишь заменой каждого вхождения newPoint на newShop.point и использованием newShop вместо newPoint в качестве аргумента конструктора Node.

Листинг 11.3. Переопределение класса Node

```
class Node
  #type Shop
  shop
  #type Node
  left
  #type Node
  right
  #type integer
  level

  function Node(shop, left, right, level)
```

Код приложения выглядит просто, но это обманчивая простота: вся сложность скрыта в функции `order`, которая выполняет все операции, необходимые для размещения заказа в реальном магазине. В нем, вероятно, вам придется взаимодействовать с собственными веб-приложениями магазинов[1], поэтому прежде всего вы должны осознать, что нужно спроектировать общий API, через который будут выполняться операции с сервисами всех магазинов[2].

11.1.2. Неожиданные проблемы

Итак, наконец определили простой метод, который отыщет ближайшего продавца и сообщит ему о покупке, чтобы тот мог отправить товар пользователю.

Обратите внимание на простоту кода в листинге 11.2. Простота в ИТ, как и в жизни, часто бывает полезна. Простой код обычно гибкий, его легко поддерживать. Однако на бумаге все может выглядеть прекрасно, а стоит развернуть приложение и открыть его для реального трафика, как обнаруживаются проблемы, о которых не задумывались и даже не имели представления. Например, в реальном мире приходится иметь дело с динамически изменяющимися запасами товаров на складах магазинов и, что еще хуже, с состоянием гонки[3].

Первая проблема, которую мы упустили из виду, — не все продавцы имеют в наличии все товары. Поэтому недостаточно найти ближайший к покупателю магазин: нужно найти ближайший магазин, на складе которого имеется заказанный товар. Когда пользователь покупает несколько разных товаров и их нужно доставить

---

1 Если, конечно, всем магазинам не предоставляется единая ИТ-инфраструктура, что вполне возможно, хотя и заведомо сложнее и технически, и финансово.

2 Это, пожалуй, самая простая часть, потому что каждый магазин может написать адаптер, обеспечивающий взаимодействие между внутренним программным обеспечением магазина и вашим интерфейсом.

3 Состояние гонки — это ситуация, когда система пытается выполнить две и более операции одновременно, при этом результат этих операций зависит от их выполнения в определенном порядке. Другими словами, состояние гонки возникает, когда операции A и B выполняются одновременно, но правильный результат получается, только если, например, A выполнится раньше B, а при завершении их в обратном порядке возникает ошибка.

вместе (чтобы сэкономить на доставке или просто уменьшить отток пользователей), было бы желательно отфильтровать магазины, в которых есть все заказанные товары, если такое возможно. Но тогда возникает другой вопрос. Что, если пользователь покупает товары A и B и есть только один магазин, готовый отправить оба товара, но он находится на расстоянии 100 миль от покупателя, и есть два других магазина, готовых отправить только один товар, но они находятся на расстоянии 10 миль от покупателя? Какое решение выбрать? А как насчет выбора между ближайшим магазином, доставляющим товар дольше или запрашивающим более высокую цену за доставку, и другим, который находится дальше, но доставляет товар за меньшие деньги? Что будет лучше для покупателя?

Еще больше проблем возникает при определении архитектуры системы (рис. 11.3). До сих пор считалось, что данные о магазинах размещаются локально, но это не обязательно так. У каждого продавца может быть своя система, с которой нужно взаимодействовать.

- Например, они могут продавать товары в физическом магазине. В этом случае информация о запасах на складах может быстро устаревать. Точно так же наша информация может оказаться неактуальной, когда запасы товаров в магазине регулярно пополняются.
- В листинге 11.2 и на рис. 11.3 предполагается, что вызов `shop.order` завершится успехом, но, поскольку это, вероятно, будет удаленный вызов через HTTP, он может завершиться ошибкой по множеству причин, не связанных с наличием товара на складе магазина: запрос может быть прерван по тайм-ауту, приложение магазина может выйти из строя и стать недоступным и т. д. Если не проверять ответ, то наше приложение никогда не узнает, был ли заказ успешно размещен. А если проверять, то что делать в случаях, когда ответ не был получен?

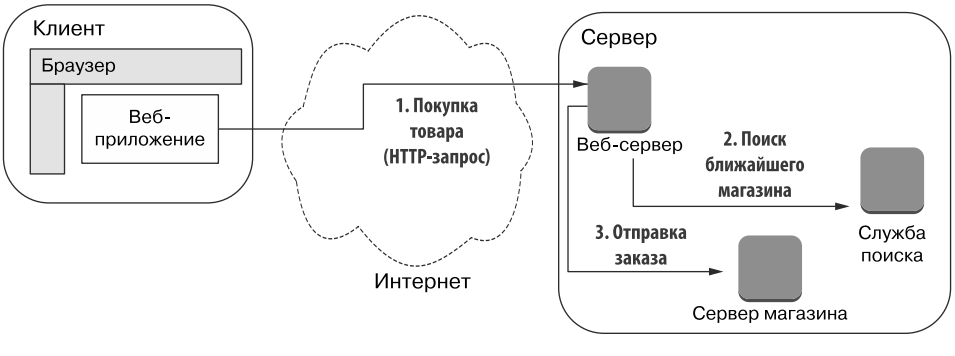


Рис. 11.3. Упрощенная архитектура приложения электронной коммерции, описанного в главе 8 и в разделе 11.1

Все это чрезвычайно сложные вопросы, и я постараюсь дать ответы на них в следующих разделах.

11.2. ЦЕНТРАЛИЗОВАННОЕ ПРИЛОЖЕНИЕ

Пока отложим в сторону все, связанное с архитектурой, и предположим, что обрабатываем все заказы самостоятельно (на той же виртуальной машине, на которой работает веб-приложение). Сосредоточимся на первом наборе вопросов. Один из них звучит так: как выбрать магазин или комбинацию магазинов, которые лучше подходят для обслуживания клиента. Ни *k*-мерные деревья, ни SS-деревья, ни любая другая структура данных не могут (и не должны) иметь готовый ответ. Решение связано с бизнесом и может меняться от компании к компании и, возможно, с течением времени или изменившимися обстоятельствами даже внутри одной компании.

Мы можем лишь реализовать в наших контейнерах механизм фильтрации точек по некоторым условиям в поиске ближайшего соседа, передавать их в код и тем самым позволить тому, кто использует контейнер, рассуждать о бизнес-правилах, настраивать их и передавать методу поиска в виде аргументов.

11.2.1. Фильтрация точек

Предусматривая обработчики для настройки бизнес-логики, мы предоставляем *шаблонный* метод, который можно эффективно настраивать в зависимости от потребностей бизнеса. В листинге 11.4 показана реализация поиска ближайшего соседа в *k*-мерном дереве, поддерживающая фильтрацию точек.

Здесь предусматривается передача дополнительного аргумента (по сравнению с методом в листинге 9.9) с предикатом, который проверяет пригодность текущего узла, прежде чем принять его на роль ближайшего соседа.

Единственная разница с базовой версией кода, помимо стандартного шаблона передачи этого нового аргумента, находится в строке 5, где предикат фактически применяется к текущему узлу.

Эта версия метода позволяет фильтровать точки и решать первую из проблем, упомянутых в предыдущем разделе, помогая убедиться, что выбран магазин, в котором действительно есть товары, заказанные покупателем.

Можно было бы, например, переопределить узлы, как показано в листинге 11.3, и передать в `fNN` предикат `hasItemX()` (листинг 11.5). Или определить более общую версию `hasItem()`, принимающую два аргумента, товар и магазин, и использующую каррирование[1] для создания предиката с одним аргументом. Предикат должен

[1] Каррирование — это прием из функционального программирования, позволяющий преобразовать вызов функции с *n* аргументами в последовательность из *n* вызовов функций с одним аргументом. Он дает возможность, например, определить общую функцию, такую как add(*a*,*b*), которая складывает числа *a* и *b*, а затем добавить новые функции, фиксирующие значение первого аргумента: add5 = add(5). Аналогично с помощью этого приема можно вызвать оригинальную функцию, чтобы сложить 4 и 3: add(4)(3). Не все языки программирования поддерживают каррирование, но многие предлагают обходные решения для его достижения.

проверять наличие одного фиксированного товара в магазине и передавать его в вызов `fNN`, как показано в листинге 11.6.

Листинг 11.4. Метод filteredNearestNeighbor

Если текущий узел удовлетворяет заданному предикату и его расстояние до цели меньше, чем расстояние текущего найденного ближайшего соседа, то обновляем значения, сохранив ссылку на ближайшего соседа и расстояние до него

Если узел пустой, это означает, что методу передано пустое дерево, в котором не может быть ближайшего соседа. Однако возможно, что при возврате будет встречена другая ветвь, где обнаружится такой сосед

В противном случае перед нами стоят три задачи: проверить, ближе ли текущий узел, чем ранее найденный ближайший сосед, обойти ветвь, расположенную на той же стороне разделения, где находится целевая точка, и проверить, можно ли исключить другую ветвь (или тоже нужно пройти ее)

Находим точку, ближайшую к заданной цели, удовлетворяющую предикату. Методу также передаются ссылка на ближайшего соседа, найденного до сих пор, и расстояние до него, чтобы облегчить исключение ветвей. По умолчанию эти значения равны null и inf, которые эти параметры принимают при вызове для корня дерева

```
function fNN(node, location, predicate, (nnDist, nn)=(inf, null))
  if node == null then
    return (nnDist, nn)
  else
    dist ← distance(node.shop.point, location)
    if predicate(node) and dist < nnDist then
      (nnDist, nn) ← (dist, node.shop)
    if compare(location, node) < 0 then
      closeBranch ← node.left
      farBranch ← node.right
    else
      closeBranch ← node.right
      farBranch ← node.left
    (nnDist, nn) ← fNN(closeBranch, location, predicate, (nnDist, nn))
    if splitDistance(location, node) < nnDist then
      (nnDist, nn) ← fNN(farBranch, location, predicate, (nnDist, nn))
    return (nnDist, nn)
```

Проверяем, находится ли целевая точка в левой ветви от линии разделения. Если да, то левая ветвь является ближайшей к целевой точке, иначе — самой дальней

Возвращаем ближайшую точку, найденную к данному моменту

Обходим дальнюю ветвь и обновляем текущие хранимые значения: ссылку на ближайшего соседа и расстояние до него

Обходим ближайшую ветвь в поисках ближайшего соседа. Сделать это и обновить хранимые значения со ссылкой на ближайшего соседа и расстоянием до него необходимо, чтобы выполнить исключение более эффективно

С помощью одной из вспомогательных функций, представленных в листинге 9.2, вычисляем расстояние между линией разделения, проходящей через текущий узел, и целевой точкой. Если это расстояние меньше, чем расстояние до текущего ближайшего соседа, то самая дальняя ветвь может содержать точки, расположенные ближе, чем текущий ближайший сосед (см. рис. 9.21)

Вычисляем расстояние между точкой текущего узла и целевым местоположением

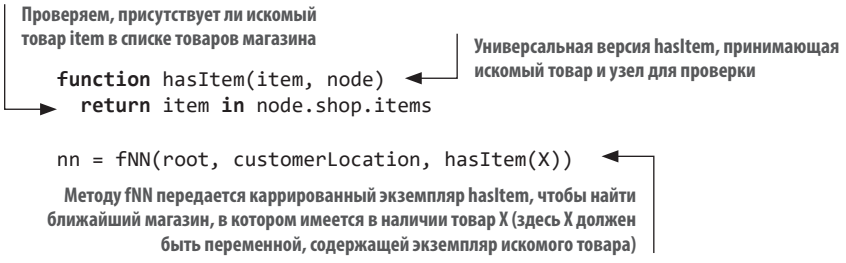
Листинг 11.5. Метод hasItemX

Определение функции, принимающей узел и проверяющей наличие в нем определенного (фиксированного) товара X

```
function hasItemX(node)
  return X in node.shop.items
```

Просто проверяем наличие товара в списке товаров магазина

Листинг 11.6. Каррированная версия hasItemX



Чтобы найти ближайший магазин, который может доставить виноград «пино нуар», можно вызвать метод `fNN` примерно так:

```
fNN(treeRoot, customerPosition, hasItem("Pinot noir"))
```

Приведенный механизм позволяет отфильтровать магазины, не имеющие нужных товаров в наличии, но он недостаточно мощный и не позволяет сделать выбор между более близким магазином с высокой стоимостью доставки и более далеким магазином, доставка из которого стоит в два раза меньше. И вообще он не справляется со сложными условиями, которые могут понадобиться для сравнения разных решений.

11.2.2. Сложные решения

Если действительно нужна возможность не просто фильтровать магазины по каким-то критериям, а выбирать лучшие варианты из имеющихся, то надо искать более мощный механизм.

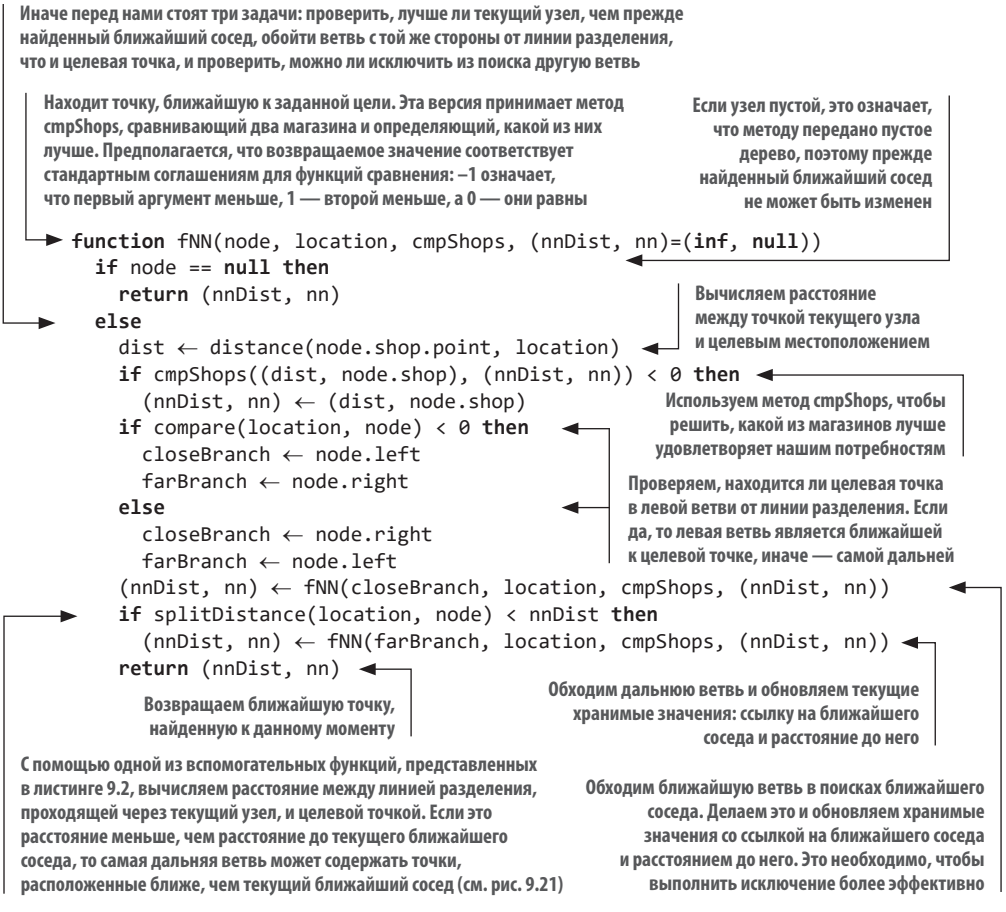
У нас есть два основных варианта:

- отыскать *n* ближайших соседей, чтобы получить список из *n* магазинов, удовлетворяющих критериям, а затем отобрать из него тот магазин, который фактически является лучшим выбором;
- или можно заменить предикат, передаваемый методу поиска ближайшего соседа, как показано в листинге 11.7, этот метод должен вместо предиката с одним аргументом использовать функцию с двумя аргументами — текущим узлом и лучшим решением, найденным на данный момент, — и возвращать лучшее решение, исходя из этих условий.

И конечно, можно использовать комбинацию этих двух вариантов.

При реализации первого варианта не требуется вносить никаких изменений в контейнер. Для выбора лучшего кандидата из списка в соответствии с бизнес-правилами можно использовать сортировку, очередь с приоритетами или любой другой алгоритм выбора, который вам нравится. Более того, у этого механизма есть даже преимущество: он позволяет попробовать решение, когда различные товары заказываются в разных магазинах, и проверить — не лучше ли оно, чем решение, когда все товары заказываются в одном месте.

Листинг 11.7. Другая версия метода filteredNearestNeighbor



Однако, когда в `fNN` передается функция сравнения, эта гибкость теряется и становится возможным рассматривать только решения, в которых все товары поставляются одним и тем же магазином.

Если это нормально, например, когда есть гарантии, что всегда можно найти такой магазин (или случай отсутствия такого магазина обрабатывается отдельно), то механизм «функция сравнения» дает дополнительное преимущество, требуя меньше дополнительной памяти, и в целом работает быстрее.

Как уже упоминалось, идея состоит в том, чтобы инкапсулировать всю бизнес-логику в функцию двух аргументов, которая будет сравнивать узел, рассматриваемый в данный момент в процессе поиска ближайшего соседа, с лучшим решением, найденным до сих пор. Также предполагается, что этот предикат выполнит любую необходимую фильтрацию, например проверит наличие в текущем магазине всех товаров на складе. Для реализации такого подхода требуется внести лишь минимальные изменения

в метод поиска, как показано в листинге 11.7. Достаточно объединить два шага (фильтрацию магазинов и сравнение текущего решения с лучшим из найденных) и вместо простой проверки расстояний во время поиска использовать функцию, переданную методу `fNN` в аргументе, функцию `cmpShops`[1], — чтобы определить, что *ближе*.

Итак, теперь бизнес-логика находится в этом методе, и он решает, какие магазины фильтровать и как выбрать наилучший вариант.

Есть несколько пограничных случаев, которые следует в этой функции учесть:

- если текущий узел отфильтрован, то всегда выбирается текущий найденный ближайший сосед;
- если лучшее решение, найденное до сих пор, пустое (то есть `nn == null`), то всегда следует выбирать текущий узел только при условии, что он не отфильтрован (см. предыдущий пункт).

В листинге 11.8 представлена возможная реализация этого метода сравнения, включающего фильтрацию магазинов, в которых не все товары есть в наличии, и проверку расстояний с помощью неуказанного эвристического метода, выбирающего лучший магазин.

Листинг 11.8. Возможная реализация метода сравнения

Метод сравнения, решающий, какой магазин предпочтительнее. Принимает два магазина (из текущего узла и ближайшего соседа, найденного на данный момент) и их расстояния

Если получена пустая ссылка на магазин shop или в нем нет всех товаров, перечисленных в списке покупок (boughtItems), то возвращается 1, чтобы сообщить вызывающей стороне, что текущий ближайший сосед является лучшим решением

```
function compareShops(boughtItems, (dist, shop), (nnDist, nnShop))
  if shop==null or not (item in shop.items ∀ item in boughtItems) then
    return 1
  else if nnShop == null then
    return -1
  else if dist <= nnDist and heuristic(shop, nnShop) then
    return -1
  else
    return 1
```

Если до сих пор решение не принято, то просто считаем лучшим найденного до сих пор ближайшего соседа

```
fNN(root, customerPosition, compareShops(["Cheddar", "Gravy", "Salad"]))
```

Иначе, если прежде не был найден лучший магазин, то возвращается –1: текущий узел лучше, чем ничего

Пример вызова метода поиска ближайшего соседа (с фильтрацией), чтобы найти наиболее подходящий магазин, в котором продаются все три товара, перечисленные в списке

Если текущий узел не хуже найденного до сих пор, то можно сравнить два магазина с помощью эвристики и посмотреть, какой из них лучше. Эта эвристика может инкапсулировать любую бизнес-логику: например, проверять стоимость доставки, стоимость товаров или любое другое условие, характерное для предметной области

[1] Сокращенная форма имени compareShops; это вынужденное сокращение, сделанное для того, чтобы уместить код листинга по ширине книжной страницы.

На рис. 11.4 показана логика работы метода, оформленная в виде блок-схемы.

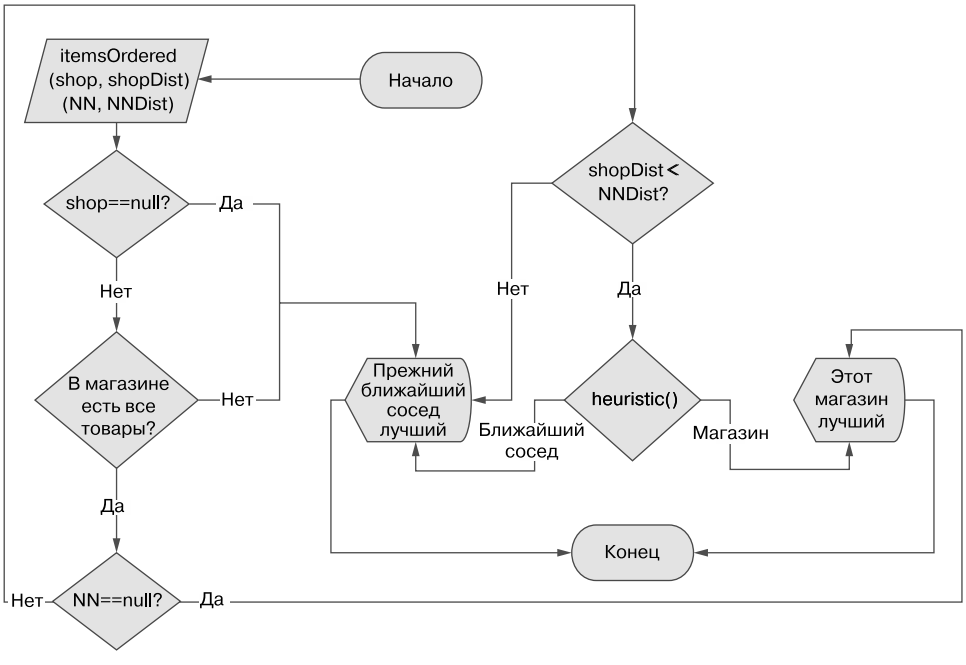


Рис. 11.4. Блок-схема, описывающая логику функции сравнения из листинга 11.8

Стоит еще раз уточнить: выбор эвристики никак не связан ни с одной из рассмотренных нами структур данных и, наоборот, представленные алгоритмы не зависят от этой функции. Просто помните, что существует эвристический метод, который инкапсулирует логику предметной области, поэтому он может меняться от приложения к приложению. В зависимости от реализации класса Shop поведение метода поиска можно настроить для решения любой конкретной задачи.

11.3. ПЕРЕХОД К РАСПРЕДЕЛЕННОМУ ПРИЛОЖЕНИЮ

Пока все складывается более или менее хорошо: мы решили задачу «ближайшего склада», исходя из предположения, что под нашим контролем находятся все части приложения и что системы, регистрирующие заказ и запускающие процесс отправки товаров клиентам, всегда доступны и никогда не подводят.

Ах, если бы реальный мир был похож на идеальный! К сожалению, не только системы (приложения, компьютеры, сети и т. д.) выходят из строя, но также велика вероятность, что некоторые компоненты, необходимые веб-приложению

электронной коммерции, подобному описанному, будут находиться под контролем других людей. Это весьма характерно для распределенных приложений, объединяющих разные службы, которые работают на разных машинах (возможно, расположенных далеко друг от друга) и общаются посредством передачи данных по сети. На рис. 11.5 показана упрощенная архитектура, основанная на рис. 11.3, отражающая более реалистичную ситуацию, когда серверы продавцов находятся на отдельных машинах, в другом адресном пространстве и взаимодействие с ними возможно только по сети (через HTTP или любой другой протокол связи, такой как *IPFS*[1]).

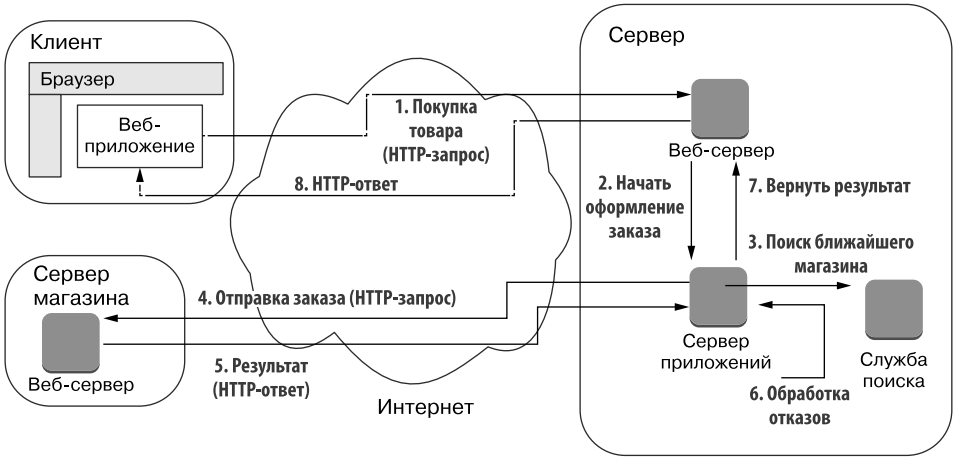


Рис. 11.5. Более реалистичная организация приложения электронной коммерции с учетом сложностей распределенной архитектуры и добавления отдельного сервера приложений для обработки сбоев HTTP-запросов к серверам магазинов

Совершенно понятно, что это меняет правила игры. Можно жестко контролировать происходящее на своей машине (виртуальной или физической), но как только в процесс вводятся удаленные вызовы, тут же появляются дополнительные точки отказа и возникает необходимость правильно обрабатывать задержки. Например, синхронно вызывая метод в локальном приложении, мы осознаем, что он может потерпеть сбой, и также понимаем, надеюсь, что сможем выяснить причину этого сбоя. Мы знаем, что вычисления могут занять некоторое время (или, возможно, даже зациклиться навсегда), но все-таки уверены, что сразу после вызова метод приступил к выполнению своей работы.

Обработка заказа с учетом всех этих факторов становится заметно сложнее, как показано на рис. 11.6.

[1] IPFS — это одноранговый протокол гипермедиа: https://ipfs.io. Он стоит того, чтобы поближе познакомиться с ним.

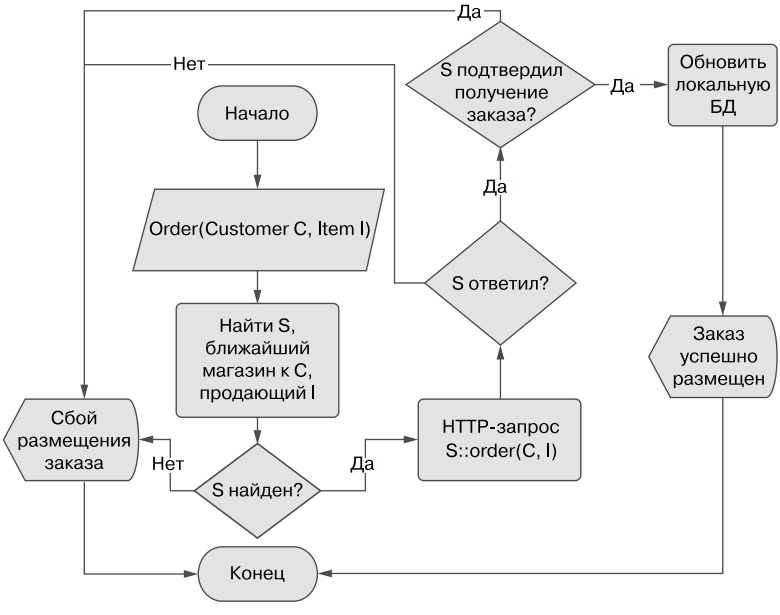


Рис. 11.6. Процесс обработки заказа, учитывающий некоторые возможные источники сбоя. С учетом этих источников выполнение перестает быть линейным, а логика обработки заказа усложняется

11.3.1. Проблемы, связанные со взаимодействиями через HTTP

В распределенных системах связь по сети становится дополнительным источником неопределенности. Мы знаем, что отправили HTTP-запрос, но, не получив ответа, не можем знать, дошло ли сообщение, или, может быть, сеть отключилась, или сервер завис на какое-то время[1].

Поэтому важно решить, как поступать с этой неопределенностью. Можно подождать ответа (синхронный обмен); отправить запрос и в ожидании ответа заняться чем-то другим; или поступить по принципу «выстрелил и забыл», то есть отправить запрос и не ждать ответа от удаленного сервера.

Описанная связь является частью рабочего процесса взаимодействия с пользователем, когда пользователи ждут ответа, и обычно их терпение небезгранично. Кто в здравом уме будет ждать 10 мин (да даже две!), прежде чем попытаться перезагрузить страницу?

И действительно, страницы электронной коммерции, от которых пользователь ожидает увидеть оперативное обновление информации в течение разумного времени,

[1] Получив ответ, мы можем проверить HTTP-код и узнать, не возникло ли какой-нибудь ошибки в сети, при условии, конечно, что вызываемая сторона точно реализует правила обработки протокола HTTP и в случае ошибок отправляет обратно соответствующие сообщения, объясняющие их причины.

обычно имеют короткие тайм-ауты на получение ответов и возвращают код ошибки 5XX через несколько секунд, обычно меньше десяти.

Это создает дополнительные проблемы, потому что, если выбрать более длительное время ожидания ответа на запрос, отправленный серверу магазина, высока вероятность, что обслуживание HTTP-запроса от покупателя завершится ошибкой, после чего в ответ на наш запрос магазин пришлет успешный ответ[1] и возникнет несоответствие. Из-за него мы можем даже заставить клиента купить один и тот же товар дважды. Этот случай показан на рис. 11.7, где представлена временна́я диаграмма происходящего.

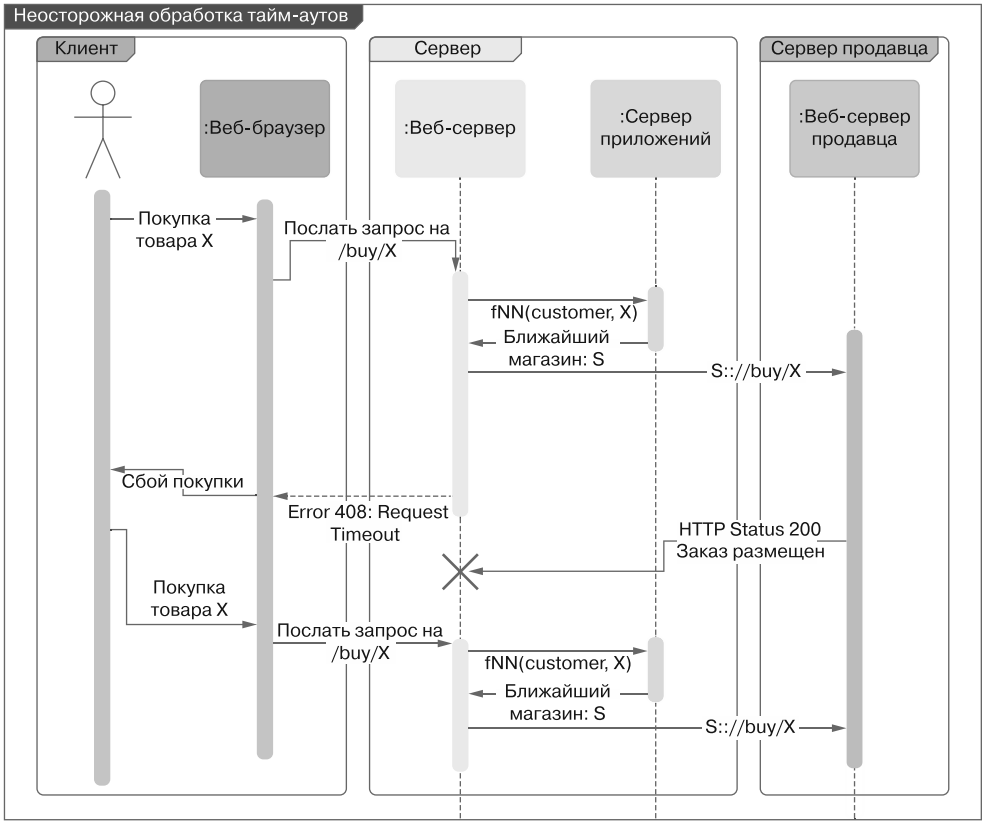


Рис. 11.7. Временная диаграмма, показывающая, как неправильная обработка тайм-аутов при обращениях к внешним службам может привести к неприятным последствиям. Время ожидания веб-сервера приложения истекает до получения ответа от внешней службы, поэтому пользователь видит, что заказ не принят, при этом служба магазина зарегистрировала его и приняла к исполнению. Если пользователь попытается заказать товар еще раз, то это уже будет запрос на дубликат

[1] Здесь подразумевается протокол HTTP/1.1, который не предлагает возможности отменить запрос вызывающей стороной. Спецификация HTTP/2 включает в себя возможность отмены отправленных запросов.

Если для взаимодействий с серверами магазинов установлен тайм-аут, равный 8 с[1], то нужно завершить все оставшиеся операции в течение 2 с, что, вероятно, оставляет меньше секунды на поиск ближайшего соседа.

Проще говоря, выполняя переход на распределенные архитектуры, необходимо учитывать гораздо большее количество факторов, которые не связаны с простым использованием алгоритмов. В такой ситуации выбор эффективного алгоритма поиска становится еще более важным. Неправильный выбор может иметь ужасные последствия для веб-приложения.

Это означает, что нужно тщательно подходить к выбору алгоритма для реализации.

- Разные структуры данных имеют разную производительность среднего и худшего случаев. Можно выбрать алгоритм с лучшей производительностью в среднем, чтобы обслуживать как можно больше запросов за кратчайшее время, или с лучшей производительностью в худшем случае, чтобы иметь уверенность, что все запросы будут выполнены в течение заданного времени (даже если в среднем он будет работать медленнее).
- Если набор данных для поиска постоянно увеличивается после определенной точки, то он может стать слишком большим и поиск ближайшего соседа может не быть выполнен за доступное время. В таком случае нужно подумать о других способах масштабирования приложения, например о географическом сегментировании данных; или, если это не имеет смысла для вашего бизнеса, задействовать приближенный алгоритм, использующий случайный сегмент с распараллеливанием, а затем выбирающий лучшее из решений, полученных для сегментов.
- При использовании приближенного алгоритма обычно приходится идти на компромисс между производительностью и точностью. А здесь следует убедиться в допустимости такого компромисса, который затрагивает качество результатов: на него приходится идти, чтобы получить ответ в течение установленного времени.

11.3.2. Синхронизация информации о товарах на складах

Если ситуация еще не кажется вам сложной, то сформулируем один добавочный вопрос, который пока не рассматривался: откуда берется информация о наличии товаров?

До сих пор предполагалось, что она хранится в узлах нашего k-мерного дерева (или SS-дерева), но это может быть не так. Если подумать, то после передачи заказа продавцу запасы продавца успевают сократиться, но информация в нашем контейнере может не отражать этого.

---

[1] К сожалению, как уже упоминалось, при использовании протокола HTTP/1.1 нет возможности задать тайм-аут, поэтому придется приспосабливаться к настройкам серверов магазинов и, в частности, к самому длительному из этих тайм-аутов. И все же для примера предположим, что тайм-аут равен 8 с — это максимальное время, после которого запрос к одному из этих серверов гарантированно будет считаться невыполненным.

Есть ряд моментов, которые необходимо учитывать: продавец может продавать товары через другие каналы (через другой сайт электронной коммерции или обычный магазин), и копия информации о запасах, с которой работаем мы, должна обновляться, когда такое происходит. А поскольку взаимодействия происходят по сети, важно быть очень осторожными с состояниями гонки, чтобы не допустить двойного размещения заказов и не потерять их.

Можно было бы придумать обходные пути для решения обеих проблем, но лучше все же использовать другой подход. На рис. 11.8 показана еще более сложная архитектура приложения, включающая базу данных (которая может быть экземпляром Memcached, SQL DB или их комбинацией) и еще одну службу, цель которой — выполняться в виде демона, не зависящего от основного приложения, и периодически опрашивать серверы продавцов для получения последней информации о наличии товаров на складах. После получения каждого ответа служба должна обновлять локальную базу данных свежими значениями.

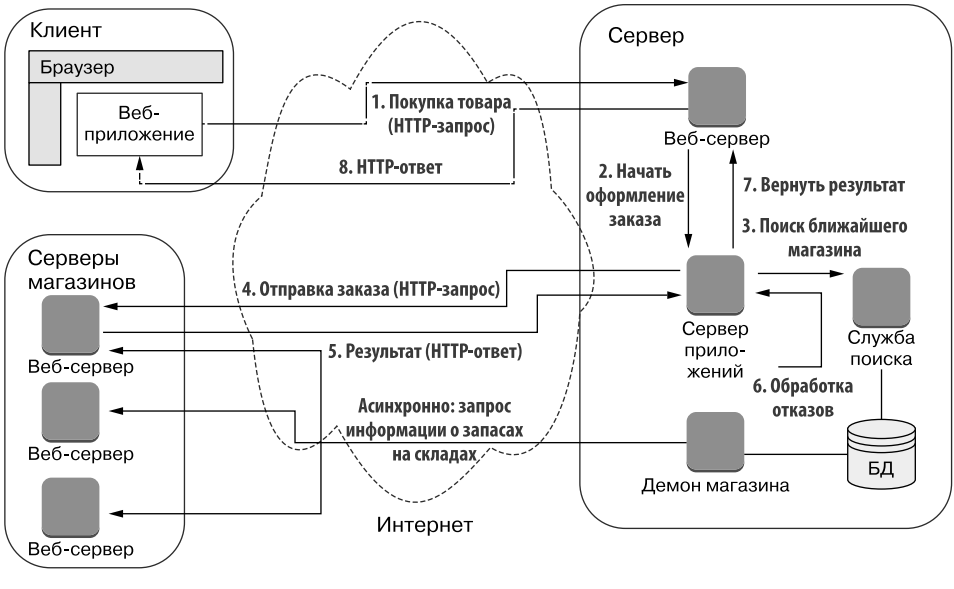


Рис. 11.8. Еще более сложная архитектура, включающая демон, который асинхронно опрашивает серверы магазинов и синхронизирует локальную БД

Это также означает, что, запуская поиск ближайшего соседа, нужно убедиться в актуальности нашей копии информации о запасах в памяти. Здесь тоже придется идти на компромисс между производительностью и точностью, потому что выполнение одного запроса к БД (даже через быстрый кэш) не будет отличаться высокой скоростью. Поэтому, вероятно, не будет лишним иметь еще один демон на машине сервера приложений, который получает обновления из базы данных только для значений, изменившихся с момента последнего обновления, просматривает список магазинов (хранящийся в некоторой общей памяти) и обновляет эти значения.

11.3.3. Извлеченные уроки

Мы с вами довольно далеко продвинулись в развитии приложения электронной коммерции: начали с обсуждения грубой схемы централизованного приложения и закончили мельчайшими деталями распределенной системы.

Это обсуждение нельзя назвать исчерпывающим, да и не стояла перед нами такая цель. Тем не менее я надеюсь, что работа над его созданием была полезна и помогла вам получить представление о том, как можно организовать процесс проектирования, ведущий от алгоритма к законченному, готовому к промышленной эксплуатации приложению. Надеюсь, что приведенные в обсуждении указания на возможные проблемы, с которыми можно столкнуться при разработке веб-приложения, были также полезны, как и советы, на что следует обратить внимание, повышая уровень образования в этой области.

Теперь пришло время двигаться дальше и познакомиться с еще несколькими задачами в совершенно другом контексте. Их тоже можно решить с помощью поиска ближайшего соседа.

11.4. ДРУГИЕ ПРИЛОЖЕНИЯ

Карты не единственная область применения поиска ближайшего соседа. Более того, о них никто и не думал, когда были изобретены *k*-мерные деревья. Конечно, *k*-мерные деревья прекрасно зарекомендовали себя в этой области, но вообще такие контейнеры предназначались для решения задач не в двух- и трехмерных пространствах, а в пространствах с намного бóльшим числом измерений.

Чтобы дать вам представление об огромном количестве областей, где могут применяться подобные алгоритмы, кратко рассмотрим несколько примеров.

11.4.1. Уменьшение количества цветов

Задача проста: есть растровое изображение RGB, использующее определенную цветовую палитру, например обычные 16 миллионов цветов RGB. Согласно этой палитре каждый пиксель изображения имеет три связанных с ним канала — красный, зеленый и синий. И каждому из этих каналов соответствует значение интенсивности цвета от 0 до 255. На рис. 11.9 показан пример кодирования растрового изображения. Скажем, если пиксель должен быть полностью красным, в соответствие ему будет поставлен кортеж (R = 255, G = 0, B = 0)[1], темно-синему пикселю — что-нибудь вроде (0, 0, 146), желтому — (255, 255, 0), черному — (0, 0, 0), а белому — (255, 255, 255).

[1] Это означает, что красный канал будет иметь максимальную интенсивность, а два других — минимальную.

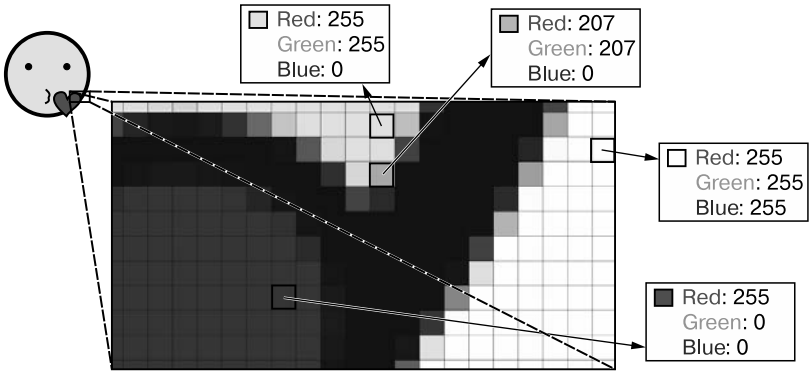


Рис. 11.9. Растровое изображение в формате RGB. Оно состоит из крошечных точек — пикселей, цвет которых определяется комбинацией трех основных цветов: красного (red), зеленого (green) и синего (blue), каждый из которых может иметь интенсивность от 0 до 255

Для хранения каждого пикселя такого изображения требуется 3 байта[1]. Для хранения изображения размером 1280 × 720 пикселей[2] потребуется 2 764 800 байт или более 2 Мбайт; а для хранения изображения размером 1920 × 1080 пикселей[3] потребуется почти 6 Мбайт.

Для экономии места можно либо использовать сжатый формат, что может привести к потере информации, либо преобразовать данные в другое пространство[4], либо уменьшить количество цветов, используемых для каждого пикселя. Предположим, что изображения сохраняются для определенной цели, например изображения дорожных знаков для передачи модели машинного обучения.

Если фон не имеет значения, то для отображения остальных элементов дорожных знаков используется ограниченный набор цветов, поэтому можно попробовать уменьшить общее количество цветов и выбрать, например, 256-цветную палитру[5]. Это поможет сократить необходимый объем памяти в 3 раза, то есть 4 Гбайт на каждую тысячу сохраненных изображений (а если изображения хранятся или обрабатываются в облаке, то это решение, вероятно, поможет сэкономить массу времени и денег).

---

1 Если хранить в несжатом виде. Такие форматы, как JPG или WEBM, хранят сжатую версию изображения, занимающую значительно меньше памяти — как минимум на порядок, пусть и с небольшой потерей качества.

2 Изображение с разрешением 720p, также известное как HD (High Density — высокого разрешения). В настоящее время такое разрешение считается низкокачественным.

3 Разрешение 1080p, или Full HD, тоже довольно далеко от разрешения 4K, ставшего стандартным в наше время.

4 Алгоритм JPG преобразует изображение из пиксельного пространства в частотное.

5 Этот прием едва ли поможет обучить надежную модель, но я привел пример только ради иллюстрации.

Итак, проблема заключается в следующем: как преобразовать каждое изображение из одной цветовой палитры в другую, сохранив высочайшую точность?

Для этого нужно выбрать 256 цветов и «втиснуть» каждый из 16 миллионов исходных цветов в эти 256 конечных цветов. Ключ к решению — в выборе конечной цветовой палитры. Конечно, сделать это можно множеством способов. Например, можно выбрать одни и те же 256 цветов для всех изображений путем равномерного масштабирования исходной 16-миллионной палитры, но также можно выбрать для каждого изображения наилучшую палитру, чтобы уменьшить потерю информации.

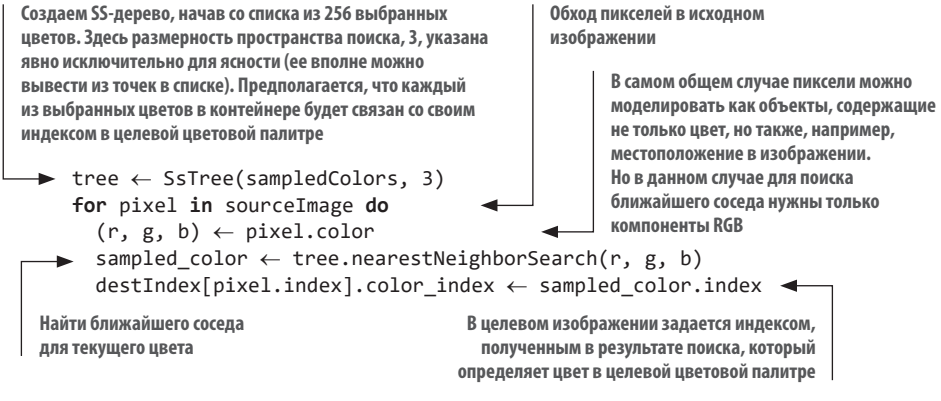
На самом деле вопрос о выборе палитры был задан мне на собеседовании одним из замечательных инженеров, с которым я потом имел удовольствие работать. Но я оставлю поиск ответа на него вам, уважаемый читатель, как самостоятельное упражнение, чтобы не испортить интригу.

Ну а, отыскав лучший вариант использования 256 цветов, как преобразовать все пиксели из одной палитры в другую?

Здесь в игру вступает поиск ближайшего соседа: нужно создать k-мерное дерево или SS-дерево, содержащее каждый из 256 целевых цветов; размерность пространства поиска при этом будет равна 3, как вы наверняка понимаете.

Затем для каждого пикселя в исходном изображении нужно отыскать его ближайшего соседа в дереве и сохранить индекс цвета в целевой палитре, ближайшего к исходному цвету пикселя. В листинге 11.9 показан псевдокод для этих операций с использованием SS-дерева.

Листинг 11.9. Уменьшение количества цветов в изображении с использованием алгоритма поиска ближайшего соседа



Видите, как просто решать такие сложные задачи, имея правильную структуру данных! Это потому, что вся сложность заключена в реализации SS-дерева (k-мерного дерева или эквивалентной структуры), и именно в этом состоит главная цель книги — помочь вам распознавать ситуации, в которых можно использовать эти структуры данных. Только приобретение такого умения сделает из вас разработчика, способного писать более быстрый и надежный код.

11.4.2. Взаимодействие частиц

Ученым, изучающим физику элементарных частиц, приходится моделировать системы, в которых большое количество атомов, молекул или субатомных частиц взаимодействует в замкнутой среде. Например, вы можете смоделировать поведение газа при изменении температуры или попадании в него лазерного луча и т. д.

На рис. 11.10 упрощенно изображено, как может выглядеть такая имитация. Учитывая, что при средней комнатной температуре 25 °C в кубическом метре воздуха содержится примерно 10^{22} молекул, нетрудно представить, что даже при небольших объемах и значительном разрежении симуляция должна обрабатывать миллиарды миллиардов частиц на каждом этапе симуляции.

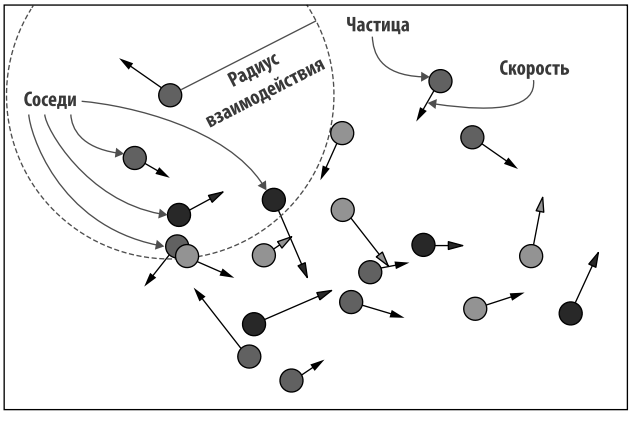


Рис. 11.10. Упрощенное представление взаимодействий частиц. В замкнутой среде взаимодействуют несколько частиц разных типов (например, частицы газа в замкнутом объеме). Каждая частица движется с определенной скоростью, и для каждой вычисляются взаимодействия только с ближайшими соседями, чтобы сэкономить вычислительные ресурсы

Все эти симуляции так или иначе вычисляют взаимодействия между частицами, но при таком их количестве проверить, как каждая частица взаимодействует с любой другой частицей, просто невозможно. Всякий раз нужно будет проверить около 10^{40} пар, а это число слишком велико для любого традиционного компьютера[1].

Более того, это даже не всегда имеет смысл: у электрической силы и гравитации ограниченный диапазон действия, поэтому вне определенного радиуса величина взаимодействий между двумя частицами будет пренебрежимо мала.

---

[1] В таких ситуациях на помощь могут прийти квантовые компьютеры: для ознакомления я рекомендую книгу Learn Quantum Computing with Python and Q# Сары С. Кайзер (Sarah C. Kaiser) и Кристофера Э. Гранада (Christopher E. Granade), выпущенную издательством Manning Publications в 2021 году (*Кайзер С., Гранад К.* Изучаем квантовые вычисления на Python и Q#).

Видите, куда я клоню? Это идеальный вариант для использования поиска *n* ближайших соседей, когда модель аппроксимируют, исходя из предположения, что на каждую частицу влияют только *n* ближайших к ней частиц (и, настраивая *n*, можно пожертвовать точностью ради скорости). Или, как вариант, можно проверить только взаимодействия между частицами, находящимися внутри радиуса действия четырех фундаментальных сил (или их подмножества в зависимости от типа частиц).

В листинге 11.10 показана возможная реализация такой модели, использующей SS-дерево для поиска ближайших соседей и фильтрации для каждой частицы и окружающих ее частиц, взаимодействие с которыми требуется вычислить. Загвоздка в том, что используемое *k*-мерное дерево (или эквивалентная ему структура данных) необходимо обновлять после каждого шага (потому что положение каждой частицы меняется), но даже в этом случае ускорение, которое можно получить, впечатляет.

Листинг 11.10. Взаимодействие частиц с поиском по диапазону

Сначала нужно инициализировать заданные n частиц: в зависимости от симуляции это может быть случайная инициализация или настройка определенной конфигурации

Функция, реализующая симуляцию, принимает количество частиц и предикат, который возвращает true после завершения симуляции (может быть основан на проверке количества выполненных итераций или других условий, например на переходе системы в стабильное состояние)

На каждом шаге нужно инициализировать SS-дерево (или аналогичную структуру данных) текущей конфигурацией системы. Поскольку положение частиц меняется на каждом шаге, необходимо каждый раз обновлять дерево или создавать новое. Предполагается, что моделируется трехмерная среда, но есть особые случаи, когда размеры кортежей могут быть другими

```
function simulation(numParticles, simulationComplete)
  particles ← initParticles(numParticles)
  while (not simulationComplete)
    tree ← SsTree(particles, 3)
    forces = {0̄ for particle in particles}
    for particle in particles do
      neighbors ← tree.pointsInSphere(particle.position, radius)
      for neighbor in neighbors do
        forces[particle] += computeInteraction(particle, neighbor)
    for particle in particles do
      update(particle, forces[particle])
```

Цикл до завершения симуляции

Цикл по частицам

Для каждого из выбранных «соседей» вычисляем силу, возникающую в результате взаимодействия двух частиц

В этой конфигурации сила взаимодействия между частицами A и B вычисляется дважды: один раз, когда particle==A и neighbor==B, и второй раз, когда particle==B и neighbor==A. В этом нет большой беды, хотя двойной просчет несколько снижает эффективность моделирования. Внеся небольшие изменения, можно следить за обновлениями пар и вычислять взаимодействия только один раз

Инициализируем массив, содержащий результирующую силу, которая воздействует на каждую частицу. Каждый элемент массива — это трехмерный вектор (согласно предположению, сделанному выше), содержащий векторную сумму всех сил, действующих на частицу. Первоначально элементам присваивается 0̄, нулевой вектор

После вычисления всех сил еще раз обходим все частицы и обновляем их координаты и скорости. Также необходимо учесть взаимодействие с границами замкнутой среды, например инвертировать скорость в момент, когда частица ударяется о стенку, предполагая упругое столкновение (или более сложное взаимодействие в случае неупругого столкновения)

Для каждой частицы отыскиваем ее соседей, то есть другие частицы, наиболее сильно влияющие на текущую. В этом примере используется поиск по сферическому диапазону с определенным радиусом, за пределами которого влияние других частиц пренебрежимо мало. Это значение, очевидно, зависит от задачи. Как вариант, можно также вычислять взаимодействие только с m ближайшими частицами для некоторого значения m

11.4.3. Оптимизация многомерных запросов к БД

Как было показано в главе 9, *k*-мерные деревья поддерживают многомерные запросы по диапазону (Multidimensional Range Queries, MDRQ), выполняя поиск по интервалам в двух или более измерениях многомерного пространства поиска. (Например, как предлагалось в главе 9, поиск всех сотрудников в возрасте от 30 до 60 лет, зарабатывающих от 40 до 100 тысяч долларов в год.)

Эти запросы широко используются в бизнес-приложениях, и многие базы данных поддерживают оптимизированные методы для их выполнения. Вы не найдете их в MySQL, но PostgreSQL, начиная с версии 9, поддерживает многомерные поисковые индексы по диапазонам, и Oracle реализует их в Extensible Indexing.

При индексировании таблиц с одним ключевым полем (и несколькими неключевыми полями) можно использовать двоичное дерево поиска, чтобы обеспечить быстрый (логарифмический) поиск на основе индексированного поля.

k-мерные деревья обеспечивают естественное расширение этого варианта использования, когда нужно индексировать таблицы с составными ключами. При обходе такого дерева циклически перебираются все поля составного ключа. Кроме того, *k*-мерные деревья предоставляют методы для поиска точного совпадения, наилучшего совпадения (ближайшего соседа) и по диапазону; также может поддерживаться возможность поиска частичного совпадения.

ПОИСК ЧАСТИЧНОГО СОВПАДЕНИЯ В K-МЕРНЫХ ДЕРЕВЬЯХ

Мы не включили методы поиска частичного совпадения в API наших контейнеров, но реализовать их несложно. Для этого нужно выполнить обычный обход с поиском точных соответствий для полей в запросе, но при этом следовать по обеим ветвям на уровнях, соответствующих полям ключей, которые не фильтруются в запросе.

Например, если *k*-мерное дерево используется для представления геопространственных данных и требуется найти все точки на линии, параллельной оси *X*, такие, что *x* == *C*, то нужно обойти обе ветви в узлах на нечетных уровнях (где происходит разделение по координате *y*), а на четных уровнях (соответствующих разделению по координате *x*) — только ветвь, содержащую *C*.

На рис. 11.11 и 11.12 показано, как поиск частичного совпадения по диапазону работает в *k*-мерном дереве.

В качестве альтернативы также можно использовать существующий метод `pointsInRectangle`, передавая ему диапазон от минимально возможного до максимально возможного значения тех полей, на которые не накладывается никаких ограничений. Так, в этом примере мы установили бы следующие критерии:

```
{x: {min:C, max:C}, y:{min=-inf, max=inf}}
```

На рис. 11.11 и 11.12 показан пример поиска частичного совпадения в *k*-мерном дереве.

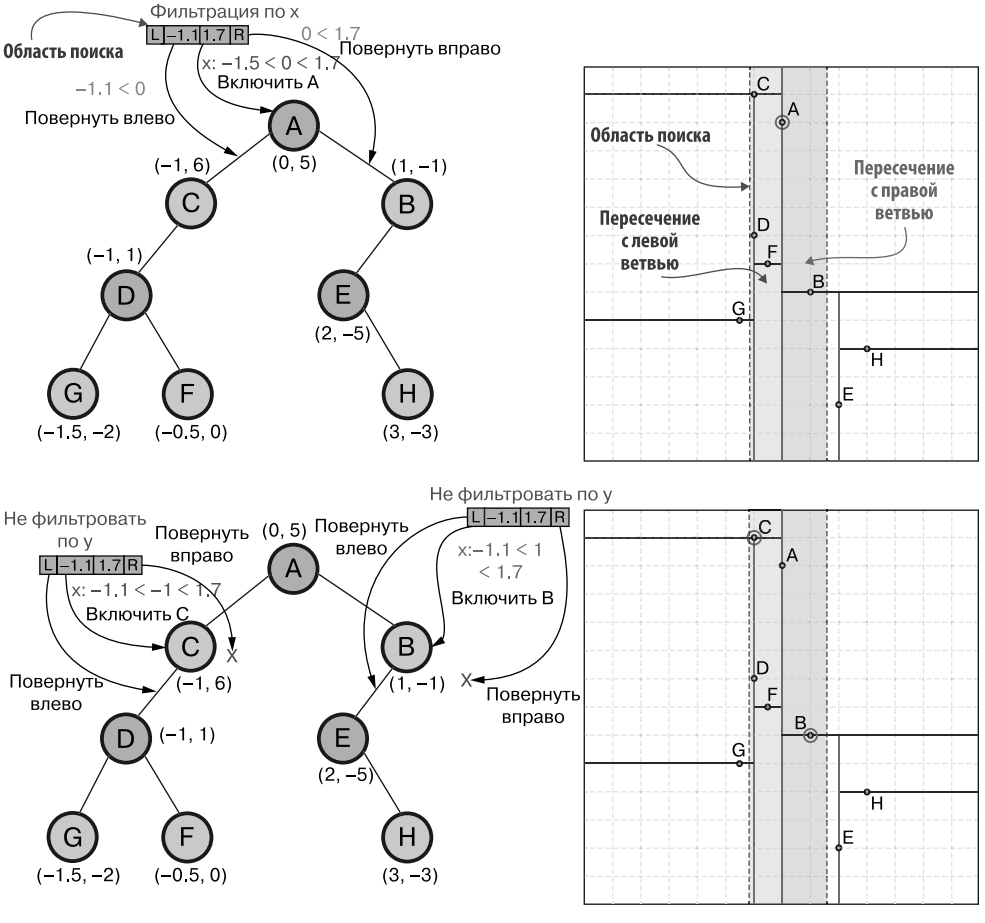


Рис. 11.11. Пример поиска частичного совпадения в k-мерном дереве. Здесь показано, как протекает поиск частичного совпадения по диапазону в двумерном дереве с полями x и y, когда критерий поиска определяет только значение x. Задача состоит в том, чтобы выбрать все точки, значение x которых находится в диапазоне от –1,1 до 1,7, независимо от значения y. На первом шаге применяется фильтрация к координате x корня, так как первое разделение происходит по координате x. Второй шаг, на узлах C и B, не выполняет никакой фильтрации, потому что разделение в этих узлах происходит по координате y, а поиск частичного совпадения выполняется только в диапазоне значений x. Координаты x узлов C и B проверяются, чтобы определить, следует ли включить их в результаты

Многие SQL-запросы можно напрямую преобразовать в вызовы методов структуры данных. Рассмотрим на нескольких примерах, как это сделать.

Прежде всего определим контекст. Представьте, что у нас есть таблица SQL с тремя полями: name (имя), birthdate (дата рождения) и salary (размер зарплаты). Первых двух вполне достаточно для первичного ключа[1], но мы также хотим создать индекс по размеру зарплаты, потому что по некоторым причинам пользователи выполняют много запросов по зарплате. Таким образом, наш индекс будет использовать трехмерное дерево с теми же полями.

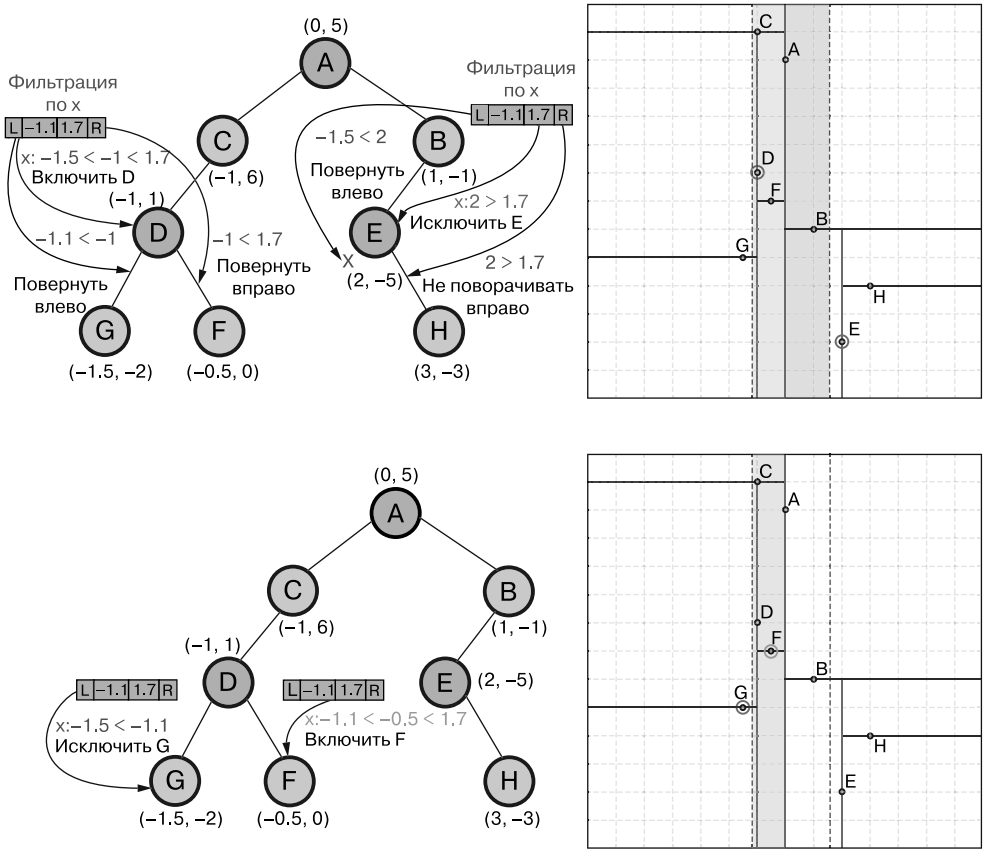


Рис. 11.12. Продолжение примера рис. 11.11 с поиском частичного совпадения в k-мерном дереве. Третий шаг вверху — снова фильтрация: в узлах D и E разделение снова происходит по координате x, поэтому мы можем исключить из поиска ненужные ветви. На последнем шаге, поскольку достигнут уровень листа, проверяется только необходимость добавления в результаты узлов G и F (будет добавлен только F)

[1] Конечно, здесь не учитывается случай, когда данных может быть настолько много, что могут встретиться два человека с одинаковыми именами и датами рождения. Предположим для простоты примера, что такое невозможно.

В табл. 11.1 показано несколько примеров фрагментов SQL и соответствующих им вызовов методов *k*-мерного дерева.

Таблица 11.1. Запросы SQL и соответствующие им вызовы методов k-мерного дерева (при условии, что для реализации множественной индексации таблицы используется k-мерное дерево, описанное в этом разделе)

Операция	**Поиск точного совпадения**
SQL[1]	`SELECT * FROM people WHERE name="Bruce Wayne" AND birthdate="1939/03/30" AND salary=150M`
k-мерное дерево	`tree.search(("Bruce Wayne", "1939/03/30", 150M))`
Операция	**Поиск точного совпадения**
SQL	`SELECT * FROM people WHERE name="Bruce" AND birthdate>"1950" AND birthdate<"1980" AND salary>=1500 AND salary<=10000`
k-мерное дерево[2]	`tree.pointsInRectangle({name:{min:"Bruce", max: inf}, birthdate:{min:"1950", max:"1980"}, salary:{min: 1500, max:10000}})`
Операция	**Поиск точного совпадения**
SQL	`SELECT * FROM people WHERE birthdate>"1950" AND birthdate<"1980" AND salary>=1500 AND salary<=10000`
k-мерное дерево	`tree.partialRangeQuery({birthdate:{min:"1950", max:"1980"}, salary:{min: 1500, max:10000}})`
k-мерное дерево[3]	`tree.pointsInRectangle({name:{min:-inf, max: inf}, birthdate:{min:"1950", max:"1980"}, salary:{min: 1500, max:10000}})`

11.4.4. Кластеризация

Наконец, мы подошли к одному из самых важных применений поиска ближайшего соседа: кластеризации. Это применение настолько важно, что ему посвящена целая следующая глава, где объясняются два алгоритма кластеризации, которые используют поиск ближайшего соседа: DBSCAN и OPTICS.

Полное описание понятия кластеризации будет, как я уже сказал, дано ниже, а пока лишь отмечу, что кластеризация — это метод обучения без учителя, когда модель машинного обучения получает набор немаркированных точек и выводит возможный

---

[1] Оператор select * обычно не приветствуется, и на то есть веские причины. Мы же будем использовать его здесь, чтобы упростить код, но вообще в запросах желательно указывать только те поля, которые действительно нужны.

[2] В примере реализации в листинге 9.12 передавался кортеж с объектами, представляющими минимальные и максимальные значения.

[3] В качестве альтернативы явной реализации поиска частичного совпадения можно использовать метод pointsInRectangle, особым образом выбирая диапазоны для полей, не имеющих ограничений.

вариант группировки этих точек в значимые категории. Например, можно разработать алгоритм кластеризации, который на основании набора данных о людях (возраст, образование, финансовое положение и т. д.) смог бы сгруппировать их в категории по сходству. Но алгоритм не сможет сказать, что это за категории. В этом и заключается работа специалиста по обработке и анализу данных, чтобы изучить результаты применения алгоритма и увидеть, например, что одна категория соответствует подросткам из среднего класса, другая, скорее всего, объединяет беби-бумеров с высшим образованием и т. д. Алгоритмы такого типа часто используются для таргетированной интернет-рекламы.

Кластеризация также используется как предварительный шаг перед применением других более сложных алгоритмов, потому что она дает недорогой способ разбить большие наборы данных на согласованные группы. Это и многое другое вы увидите в следующей главе.

РЕЗЮМЕ

- Поиск ближайшего соседа можно использовать для сопоставления физических ресурсов по географическому положению, например для поиска магазина, ближайшего к покупателю.
- При переходе от теории к практике необходимо учитывать множество факторов и корректировать алгоритм поиска ближайшего соседа, чтобы он принимал во внимание бизнес-логику, например позволяя фильтровать ресурсы или взвешивать результаты в соответствии с некоторыми бизнес-правилами.
- Также необходимо учитывать ограничения физических вычислительных систем, включая объем памяти, доступность процессора и пропускную способность сети.
- Распределенные веб-приложения подвержены дополнительным проблемам, о которых следует помнить при проектировании систем. Недостаточно придумать хороший алгоритм, нужно выбрать/спроектировать такой алгоритм, который сможет работать в реальной системе.
- Поиск ближайшего соседа имеет широкое применение во многих областях — от моделирования в физике элементарных частиц до машинного обучения.

12 Кластеризация

В этой главе

- ✓ Различные виды кластеризации.
- ✓ Разделяющая кластеризация.
- ✓ Понимание и реализация метода *k*-средних.
- ✓ Кластеризация на основе плотности.
- ✓ Понимание и реализация алгоритма DBSCAN.
- ✓ OPTICS: алгоритм иерархической кластеризации, улучшенная версия DBSCAN.
- ✓ Оценка результатов кластеризации.

В предыдущих главах мы описали и реализовали три структуры данных, обеспечивающие эффективный поиск ближайшего соседа, рассмотрели примеры их практического применения. Описывая их использование, я упомянул, что кластеризация — одна из основных областей, где эффективный поиск ближайшего соседа имеет особое значение. Мы отложили обсуждение этой темы, и вот теперь наконец пришло время добавить глазури на торт и воспользоваться плодами нашего тяжелого труда. В этой главе сначала кратко познакомимся с понятием кластеризации и узнаем, что это такое и как она связана с машинным обучением

и искусственным интеллектом, перечислим разные виды кластеризации, основанные на различных подходах, а затем подробно обсудим три совершенно разных алгоритма. Здесь вы познакомитесь с теоретическими основами кластеризации, с алгоритмами, которые можно реализовать или просто применить для разбиения наборов данных на более мелкие однородные группы, а также получите более глубокое понимание поиска ближайшего соседа и многомерного индексирования.

Но прежде я опишу задачу, для решения которой потребуется использовать кластеризацию. В предыдущих главах во второй части этой книги мы начали работу над примером сайта электронной коммерции, «основав» его еще в те дни, когда Интернет только стал входить в моду. Теперь пришло время перенести нашу компанию в 2010-е и нанять команду специалистов по данным. На самом деле, чтобы торговля процветала, необходимо провести сегментацию клиентов, чтобы понять их поведение и классифицировать, опираясь на известные нам сведения о них[1]: их покупательские привычки и финансовое положение, демографические данные, возраст, уровень образования и место проживания — все эти факторы так или иначе влияют на вкусы и платежеспособность людей.

Сегментация клиентов заключается в разделении клиентов на однородные группы со сходной покупательной способностью, историей покупок или ожидаемым поведением. Кластеризация — один из этапов этого процесса, на котором из необработанных немаркированных данных формируются группы-кластеры. Алгоритмы кластеризации не описывают группы; они просто разбивают на них всю совокупность клиентов, а затем специалисты по данным должны провести дальнейший анализ различных морфотипов, чтобы понять, что объединяет людей в отдельных кластерах. После получения этих знаний их можно использовать для проведения целевых рекламных кампаний внутри каждой из групп (или некоторых из них, имеющих решающее значение для благосостояния компании). Например, на веб-сайте компании, занимающейся распространением потокового видео (такой как Netflix), специалисты по данным могут определить группы пользователей, более склонных смотреть комедии или интересующихся боевиками и т. д.

В реальной жизни клиенты характеризуются сотнями особенностей, которые учитываются при маркетинговой сегментации. В этой главе для наглядности и простоты мы используем упрощенный пример, содержащий всего две характеристики: годовой доход и среднемесячные траты на нашем сайте электронной коммерции. Вернемся к этому примеру позже, а пока познакомимся с дополнительным контекстом и инструментами, необходимыми для понимания и выполнения кластерного анализа.

---

[1] Именно по этой причине компании пытаются заполучить как можно больше информации о вас: данные имеют первостепенное значение в нашу эпоху.

12.1. ВВЕДЕНИЕ В КЛАСТЕРИЗАЦИЮ

В последние годы, особенно во второй половине первого десятилетия этого века, отдельная ветвь исследований в области искусственного интеллекта (ИИ) получила настолько мощный импульс к развитию, что теперь она рассматривается в СМИ и общественном мнении как синоним ИИ. Я говорю, конечно же, о *машинном обучении*, которое, в свою очередь, в последнее время (примерно с 2015 года) все чаще отождествляется с *глубоким обучением*.

На самом деле глубокое обучение — лишь часть машинного обучения и основано на моделях, построенных с помощью *глубоких* (читай: многослойных) нейронных сетей, а машинное обучение, в свою очередь, является всего лишь одной из ветвей искусственного интеллекта.

В частности, машинное обучение — это отрасль, ориентированная на разработку математических моделей, описывающих систему после изучения ее характеристик на основе данных.

Машинное и глубокое обучение может достигать впечатляющих, а иногда и невероятных результатов. Пример тому — модели, генерирующие фотореалистичные изображения лиц, пейзажей и даже фильмов, такие как генеративно-состязательные сети[1]. Но оно не может и не ставит перед собой цели создать «интеллектуального» агента — что-то более близкое к романтической идее искусственного сознания, которую нам подарили культовые фильмы, такие как «Короткое замыкание» или «Военные игры». Генерация такого сознания — скорее цель *универсального искусственного интеллекта*.

12.1.1. Типы обучения

Классификация моделей машинного обучения основана на типе обучения. В частности, на том, как реализована обратная связь с моделью во время обучения.

- Обучение с учителем. Эти модели обучаются на маркированных данных: то есть каждая запись в обучающем наборе имеет метку (для алгоритмов *классификации*) или значение (для алгоритмов *регрессии*), характеризующую эту запись. В процессе обучения модель настраивает свои параметры, чтобы повысить точность определения класса или значения для новых записей. Примерами обучения с учителем могут служить модели обнаружения объектов (классификация) или прогнозирования цен на товары (регрессия).

---

[1] Генеративно-состязательные сети (Generative Adversarial Network, GAN) — это особый тип глубоких нейронных сетей, в которых две конкурирующие модели обучаются генерировать искусственный контент (на основе обучающего набора) и отличать искусственный контент от реального. Их совместная эволюция повышает реалистичность создаваемого контента.

- Обучение с подкреплением. Вместо анализа явных меток, связанных с данными, модель выполняет какую-то задачу и только в конце получает обратную связь о результате (с оценкой успешности). Это одна из самых интересных областей исследований на момент написания данной книги. В качестве примеров можно привести теорию игр (когда модель учится играть в шахматы или го) и многие области робототехники (например, обучение роботизированной руки жонглированию предметами).
- Обучение без учителя. Эта категория отличается от первых двух тем, что алгоритмы не получают никакой обратной связи с данными и их цель состоит в том, чтобы осмыслить данные путем экстраполяции их внутренней (часто скрытой) структуры. Кластеризация является основной формой обучения без учителя.

Очевидно, что в оставшейся части главы мы сосредоточимся на обучении без учителя. Алгоритмы кластеризации, по сути, получают немаркированный набор данных и пытаются собрать как можно больше информации о его структуре, группируя схожие точки данных и отделяя разнородные.

На первый взгляд такой подход выглядит менее понятным, чем обучение с учителем или обучение с подкреплением, но у кластеризации есть несколько естественных вариантов применения, и лучший способ описать кластеризацию — это привести примеры некоторых из них.

1. *Сегментация рынка*. Алгоритм получает данные о покупках и выделяет группы клиентов со схожим покупательским поведением. Поскольку клиенты в каждой группе имеют похожие черты поведения, высока вероятность, что маркетинговая стратегия будет одинаково работать (или не работать) для всей группы (при условии, что сегментация выполнена правильно). Алгоритм не будет определять смысловые метки для групп; он не скажет, объединяет ли группа «студентов до 25 лет» или «писателей среднего возраста, увлекающихся комиксами». Он просто соберет вместе похожих людей, а затем аналитики смогут дополнительно изучить группы, чтобы лучше понять их состав.
2. *Поиск выбросов*. Алгоритм пытается выявить данные, выделяющиеся на общем фоне. В зависимости от контекста это может быть шум в сигнале, новый вид цветов в тропическом лесу или даже новый поведенческий шаблон клиентов.
3. *Предварительная обработка*. Поиск кластеров в данных и обработка каждого кластера отдельно (и, возможно, параллельно). Очевидно, что этот подход может обеспечить значительное ускорение, но также имеет и другой побочный эффект. Поскольку при делении огромного набора данных на более мелкие части объем пространства уменьшается, каждая отдельная часть может уместиться в памяти или обрабатываться на одной машине, а весь набор данных — нет. Иногда даже можно использовать быстрый алгоритм кластеризации (например, *кластеризацию купола* (canopy clustering)) в качестве первого шага перед применением более медленного алгоритма кластеризации.

12.1.2. Типы кластеризации

Кластеризация — это NP-трудная (NP-hard) задача[1], поэтому решить ее точно сложно с вычислительной точки зрения (невозможно для современных реальных наборов данных). Более того, сложно даже определить объективные метрики для оценки качества решения! Эту идею поясняет рис. 12.1. Наша интуиция подсказывает, что два кольца должны находиться в разных кластерах, но трудно придумать метрическую функцию, которая объективно выполнит такое деление (например, функция минимального расстояния в этом случае не работает). Для многомерных наборов данных ситуация становится еще сложнее (здесь даже наша интуиция не может помочь проверить результат, потому что трудно представить и интерпретировать что-либо за пределами трехмерного пространства).

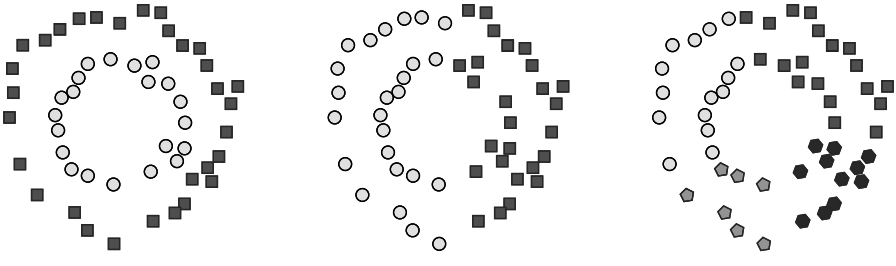


Рис. 12.1. Набор данных, который сложно кластеризовать. *Слева* показана идеальная кластеризация, соответствующая нашей интуиции. Рядом изображены неоптимальные результаты, полученные по таким метрикам, как близость или распространение сходства

По этим причинам все алгоритмы кластеризации являются эвристиками, которые более или менее быстро сходятся к локально оптимальному решению.

К категории кластеризации мы отнесли несколько алгоритмов разделения данных, использующих совершенно разные подходы. Все эти подходы могут применяться к задачам, описанным в предыдущем разделе, почти взаимозаменяемо, хотя очевидно, что каждый имеет сильные и слабые стороны, поэтому мы должны выбирать наиболее подходящие алгоритмы, исходя из наших требований.

Первое важное различие проводится между жесткой и мягкой кластеризацией, как показано на рис. 12.2:

- при жесткой кластеризации каждая точка связывается с одним и только одним кластером (или не более одного, если алгоритм кластеризации также может обнаруживать шум);

---

[1] Класс NP-трудных задач включает задачи, по крайней мере не уступающие в сложности самым сложным задачам NP. Как было показано в главе 2, класс задач NP включает задачи, решаемые за полиномиальное время на недетерминированной машине, и, в частности, задачи, которые относятся к классу NP, но не к классу P, не могут быть решены за полиномиальное время на детерминированной машине. На сегодняшний день выявление наличия какой-либо задачи в NP-P является одной из самых больших проблем в информатике.

- при мягкой кластеризации каждой точке P и каждой группе G присваивается вероятность того, что P принадлежит G.

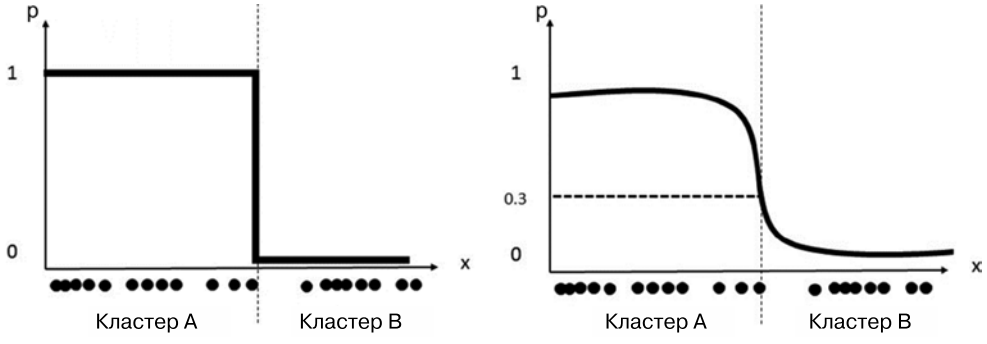


Рис. 12.2. Различие между жесткой и мягкой кластеризацией можно объяснить с точки зрения используемой ими функции определения принадлежности. На выходе жесткая кластеризация возвращает 0 или 1 для каждой точки и каждого кластера, вследствие чего 1 может быть получена только для одной из комбинаций «точка — кластер». Функция принадлежности для мягкой кластеризации возвращает вероятность от 0 до 1 (любое значение между ними), и для каждой комбинации «точка — кластер» она может быть отличной от нуля

Другой основной критерий классификации алгоритмов кластеризации отличает *разделяющую кластеризацию* от *иерархической*:

- разделяющая кластеризация, также известная как *плоская кластеризация*, делит входной набор данных на разделы, поэтому ни один кластер не является подмножеством другого и не пересекается ни с каким другим кластером;
- иерархическая кластеризация создает иерархию кластеров, которые затем можно интерпретировать и «нарезать» по любой заданной точке в зависимости от параметров, установленных для алгоритма.

Очевидно, что эти два критерия ортогональны. Например, алгоритмы кластеризации могут быть жесткими разделяющими (такими как алгоритм k-средних) или мягкими иерархическими (такими как OPTICS).

Другие критерии, часто используемые для классификации, делят алгоритмы на центроидные и плотностные, а также рандомизированные и детерминированные.

В следующем разделе мы сначала рассмотрим алгоритм разделяющей кластеризации — k-средних, прародителя всех алгоритмов кластеризации. Затем перейдем к другому типу плоской кластеризации, алгоритму DBSCAN, и, наконец, исследуем иерархическую кластеризацию и познакомимся с алгоритмом OPTICS, который относится к категории плотностных алгоритмов. В табл. 12.1 перечислены основные характеристики трех алгоритмов, представленных в этой главе.

Не волнуйтесь, в следующих разделах я подробно опишу каждое из этих свойств и приведу примеры, чтобы различия между алгоритмами стали более ясными.

Таблица 12.1. Характеристики алгоритмов кластеризации, представленных в этой главе

Категория	k-средних	DBSCAN	OPTICS
Класс	Жесткий	Жесткий	Мягкий
Структура	Плоский	Плоский	Иерархический
Стратегия	Центроидный	Плотностный	Плотностный
Детерминизм	Рандомизированный	Детерминированный	Детерминированный
Обнаружение выбросов	Нет	Да	Да

12.2. K-СРЕДНИХ

Начнем наше обсуждение с классического алгоритма, история успеха которого берет начало в 1950-х годах. Это разделяющий алгоритм *k*-средних, он объединяет данные в заранее определенном количестве сферических кластеров.

На рис. 12.3 показано, как работает метод *k*-средних. Работу алгоритма можно разбить на три основных шага.

1. *Инициализация* — создаются *k* случайных *центроидов*, случайных точек, которые могут принадлежать или не принадлежать набору данных и будут играть роль центров сферических *кластеров*.
2. *Классификация* — для каждой точки в наборе данных вычисляется расстояние до каждого центроида, и эта точка назначается ближайшему центроиду.
3. *Повторное центрирование* — точки, назначенные центроиду, образуют кластер. Для каждого кластера вычисляется его центр масс (как описано в разделе 10.3), а затем центр кластера перемещается в вычисленный центр масс.
4. Шаги *классификации* и *повторного центрирования* повторяются, пока не возникнет ситуация, когда на шаге 2 не найдена ни одна точка для отнесения к другому кластеру или не будет достигнуто максимальное количество итераций.

Шаги 2–4 этого алгоритма представляют детерминированную эвристику, которая вычисляет точные расстояния между точками и центроидами, а затем обновляет центроиды кластеров, перемещая их во вновь вычисленные центры масс. Однако алгоритм квалифицируется как рандомизированный алгоритм Монте-Карло из-за первого шага, где используется случайная инициализация. Как оказывается, этот шаг имеет решающее значение для алгоритма. На самом деле первоначальный выбор центроидов сильно влияет на окончательный результат. Неправильный выбор может замедлить сходимость, что, в свою очередь, при ограниченном максимальном количестве итераций приведет к преждевременной остановке и плохому результату. А поскольку алгоритм удаляет центроиды, с которыми не связана ни одна точка, очень плохой начальный выбор с несколькими центроидами, расположенными близко друг к другу, может привести к нежелательному сокращению числа центроидов на ранних стадиях эвристики (рис. 12.4).

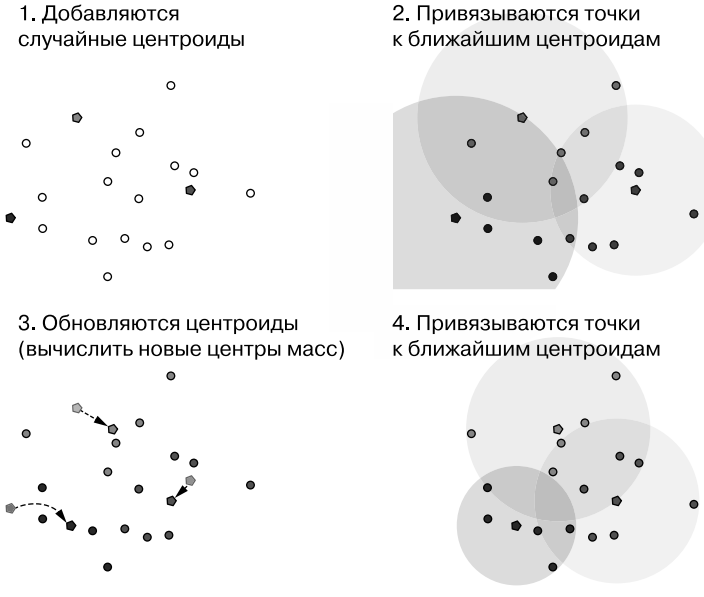


Рис. 12.3. Пример работы алгоритма k-средних с k == 3. Точки, входящие в набор данных изначально, слегка затенены. Шаг 1: случайным образом создаются k центроидов (показаны пятиугольниками). Каждому центроиду назначается свой цвет. Шаг 2: для каждой точки измеряется расстояние до всех центроидов и ей назначается ближайший центроид. В конце каждый центроид C будет определять кластер, сферу с центром в C, радиус которой равен расстоянию до самой дальней точки, назначенной этому центроиду C. Кластеры выделены тем же цветом, что и их центроиды. Шаг 3: обновление центроидов — для каждого получившегося кластера вычисляется центр масс. Шаг 4: повторяются шаги 2 и 3 j раз. Некоторые точки будут менять кластер

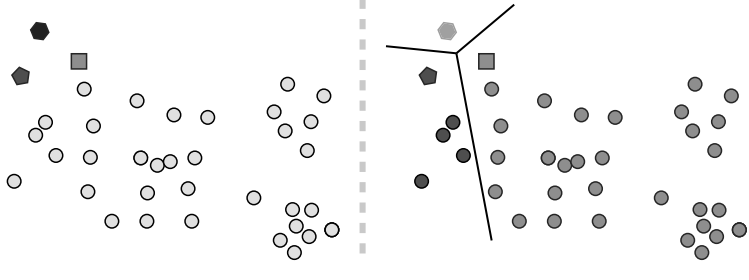
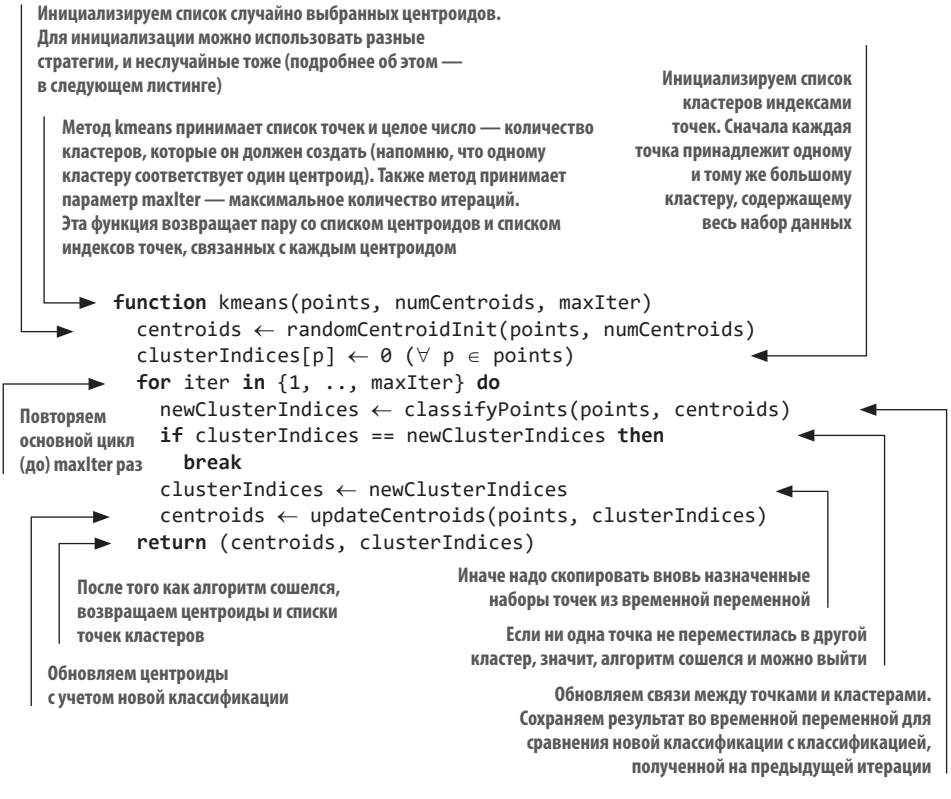


Рис. 12.4. Очень неудачный выбор начальных центроидов (показаны многоугольниками). Все центроиды собраны в одном углу набора данных. Черная линия показывает приблизительные границы между областями, определяемыми каждым центроидом (в идеале расстояние между центроидами и разделяющими линиями одинаково). Поскольку центроид, представленный шестиугольником, находится дальше, чем два других, ему не будет назначено ни одной точки, поэтому алгоритм удалит его из списка центроидов. Выбор двух других центроидов тоже выглядит несбалансированным, хотя следующие шаги повторного центрирования будут (медленно) перебалансировать ситуацию, перемещая квадрат к центру кластера справа

Чтобы смягчить эту проблему, на практике алгоритм *k*-средних всегда используется вкупе со стратегией случайного перезапуска: алгоритм запускается несколько раз на одном и том же наборе данных, каждый раз с другой случайно выбранной комбинацией центроидов, а затем результаты сравниваются и выбирается лучший (подробнее об этом рассказывается в последнем разделе главы).

В листинге 12.1 показан код основного метода, выполняющего кластеризацию методом *k*-средних. Основные шаги выделены в отдельные функции, чтобы получить более чистый и удобный для сопровождения код. Каждую из них мы рассмотрим на следующих страницах. Желающие могут также ознакомиться с реализациями этого метода в репозитории книги на GitHub[1].

Листинг 12.1. Кластеризация методом k-средних

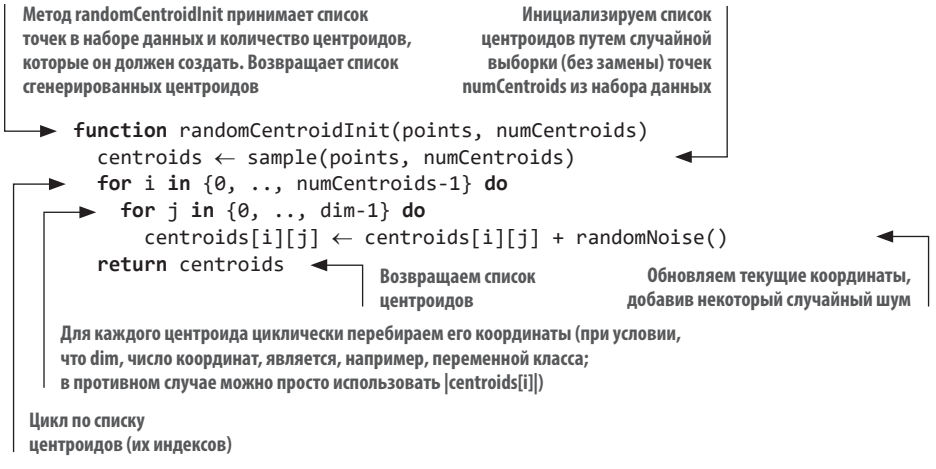


Алгоритм можно рассматривать как эвристику поиска, сходящуюся к (локальному) оптимуму, с немного более сложными функциями вычисления оценки и шага градиента. В строке 6 находится условие, проверяющее, сошелся ли алгоритм. Если классификация точек не изменилась на последнем шаге, то центроиды останутся на тех же местах, что и на предыдущем шаге, поэтому дальнейшие итерации будут

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#k-means.

выполняться вхолостую. Поскольку вычисление этих функций на каждом шаге довольно затратно, а сходимость не гарантируется, в алгоритм добавлено еще одно условие остановки, ограничивающее максимальное число итераций. Как уже упоминалось, мы абстрагировали логику функций обновления и оценки в отдельные методы. Но прежде, чем рассматривать их, интересно проверить случайный шаг инициализации, который часто недооценивают. В листинге 12.2 показана предлагаемая реализация этой функции.

Листинг 12.2. Метод randomCentroidInit



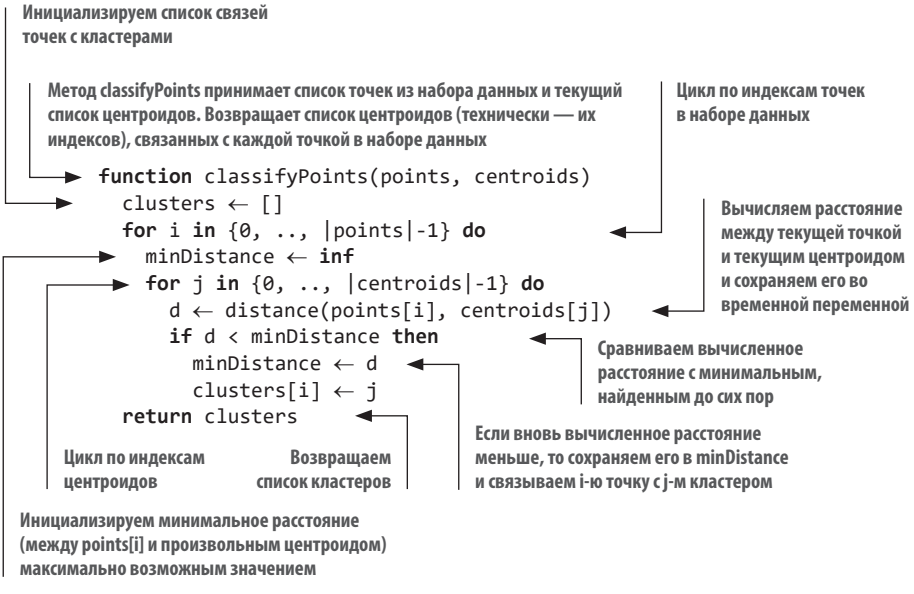
Несмотря на наличие нескольких жизнеспособных альтернатив (например, случайный выбор каждой координаты в некоторых границах или использование фактических точек, имеющихся в наборе данных), решение, основанное на выборе *k* случайных точек из набора данных с дополнительными случайными искажениями их координат, дает несколько преимуществ.

- Не нужно беспокоиться о границах. Если бы координаты каждого центроида генерировались совершенно случайно, то пришлось бы сначала просмотреть набор данных, чтобы найти приемлемый диапазон для каждой координаты.
- Даже если учитывать границы набора данных, сгенерированные центроиды могут оказаться в разреженных или пустых областях и поэтому впоследствии могут быть удалены. В то же время, если равномерно выбирать существующие точки из набора данных, то сгенерированные центроиды будут располагаться близко к точкам в данных и...
- ...Увеличится вероятность, что они окажутся в областях с большей плотностью точек.
- Дополнительное случайное возмущение помогает уменьшить риск, что все центроиды окажутся сосредоточенными в более плотных областях.

Код в листинге 12.2 интуитивно понятен, но есть пара интересных моментов, на которые стоит обратить внимание.

В строке 2 используется общая функция выборки, которая извлекает *n* элементов из набора без повторения; детали этого метода здесь неважны, поэтому не будем о них беспокоиться. Большинство языков программирования предоставляют такую функцию в своих стандартных библиотеках[1], так что вам вряд ли придется писать ее самостоятельно[2].

Листинг 12.3. Метод classifyPoints



Теперь можно перейти к этапу классификации, реализация которого представлена в листинге 12.3. Метод `classifyPoints` просто перебирает все пары «точка — центроид», чтобы отыскать ближайший центроид к каждой точке. Если вы дочитали книгу до этого места, то при словах «просто перебирает» у вас уже должны побежать мурашки по коже и возникнуть мысль: а нельзя ли использовать что-то получше? Мы вернемся к этому вопросу несколькими разделами ниже.

Теперь рассмотрим последний вспомогательный метод, который нужно реализовать для завершения алгоритма *k*-средних, — метод, обновляющий центроиды (листинг 12.4).

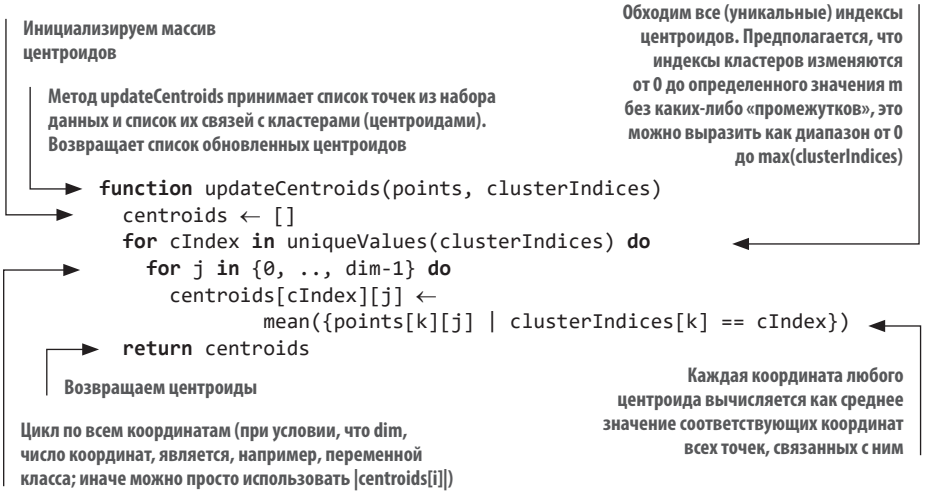
Обновляя центроиды, как уже упоминалось, метод просто вычисляет центр масс для каждого кластера. Это означает, что нужно сгруппировать все точки по центроиду, с которым они связаны, а затем для каждой группы вычислить среднее значение координат точек.

---

[1] Например, в Python можете импортировать функцию sample из модуля random, а в JavaScript — метод sample из стандартной библиотеки.

[2] Но если это действительно понадобится, то я рекомендую прочитать главу 11 из книги Practical Probabilistic Programming Стюарта Рассела (Stuart Russell), изданной Manning Publications в 2016 году, где приводится подробное описание такой функции.

Листинг 12.4. Метод updateCentroids



Эта реализация не предпринимает никаких действий для удаления центроидов, которым не назначена ни одна точка, — они просто игнорируются. Проблема с массивом центроидов тоже никак не решена; массив просто инициализируется пустым, при этом предполагается, что он будет динамически изменяться при добавлении в него новых элементов. В реальной реализации, конечно, обе проблемы должны быть приняты во внимание с учетом особенностей поддержки массивов в конкретном языке программирования.

Желающие посмотреть, как выглядят действительные реализации алгоритма *k*-средних, найдут версию на Python в репозитории книги на GitHub. По адресу https://github.com/andreaferretti/kmeans также можно найти хороший ресурс (не имеющий отношения к книге), который, подобно сайту Rosetta Stone, содержит реализации этого алгоритма на многих языках программирования (там вы найдете версии на большинстве основных языков).

Теперь, познакомившись с реализацией алгоритма *k*-средних, посмотрим еще раз, как он работает.

12.2.1. Недостатки алгоритма k-средних

На рис. 12.5 показан результат применения кластеризации методом *k*-средних к искусственному набору данных. Набор данных был специально разработан, чтобы представить идеальную ситуацию, в которой метод *k*-средних показывает блестящий результат. Кластеры легко (и линейно[1]) разделимы, и все имеют приблизительно сферическую форму.

[1] В *d*-мерном пространстве два множества S_1 и S_2 линейно разделимы, если существует хотя бы одна $(d - 1)$-мерная гиперплоскость, разделяющая пространство так, что все точки S_1 лежат по одну ее сторону, а все точки S_2 — по другую.

Начнем с положительного замечания о методе *k*-средних. На рис. 12.5 показано, как ему удается правильно идентифицировать кластеры с разной плотностью. Например, два кластера внизу слева имеют большее среднее расстояние между точками (и, следовательно, меньшую плотность), чем кластер вверху справа. На первый взгляд, в этой ситуации нет ничего особенного, но не все алгоритмы кластеризации способны так же хорошо работать с гетерогенными распределениями. В следующем разделе вы узнаете, почему DBSCAN испытывает проблемы в таких случаях.

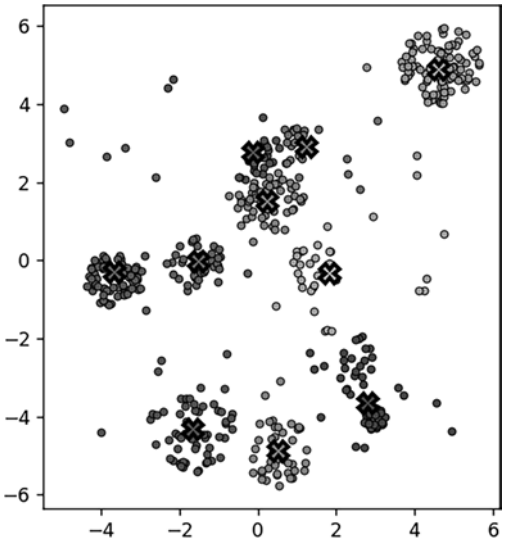


Рис. 12.5. Типичный результат кластеризации, полученный методом k-средних. Точки из набора данных изображены кружками, а центроиды — крестиками

Но на этом хорошие новости заканчиваются. На рис. 12.5 можно также видеть, что есть несколько точек, находящихся далеко от всех сферических скоплений. В набор данных также было добавлено немного шума, чтобы показать один из важнейших недостатков метода *k*-средних: он не способен обнаруживать выбросы и фактически, как показано на рис. 12.5, добавляет выбросы в ближайшие кластеры. Поскольку центроиды вычисляются как центры масс кластеров, а функция среднего чувствительна к выбросам, то, если не отфильтровать выбросы перед запуском кластеризации методом *k*-средних, центроиды кластеров будут «притягиваться» к выбросам, покидая лучшие позиции, которые они могли бы занимать. Это явление можно наблюдать в нескольких кластерах на рис. 12.5.

Однако проблема с выбросами не самая большая в методе *k*-средних. Как уже упоминалось, этот алгоритм может создавать только сферические кластеры, но, к сожалению, в реальных наборах данных не все кластеры сферические! На рис. 12.6 показаны три примера, когда кластеризация методом *k*-средних не способна распознать наилучший вариант разделения на кластеры.

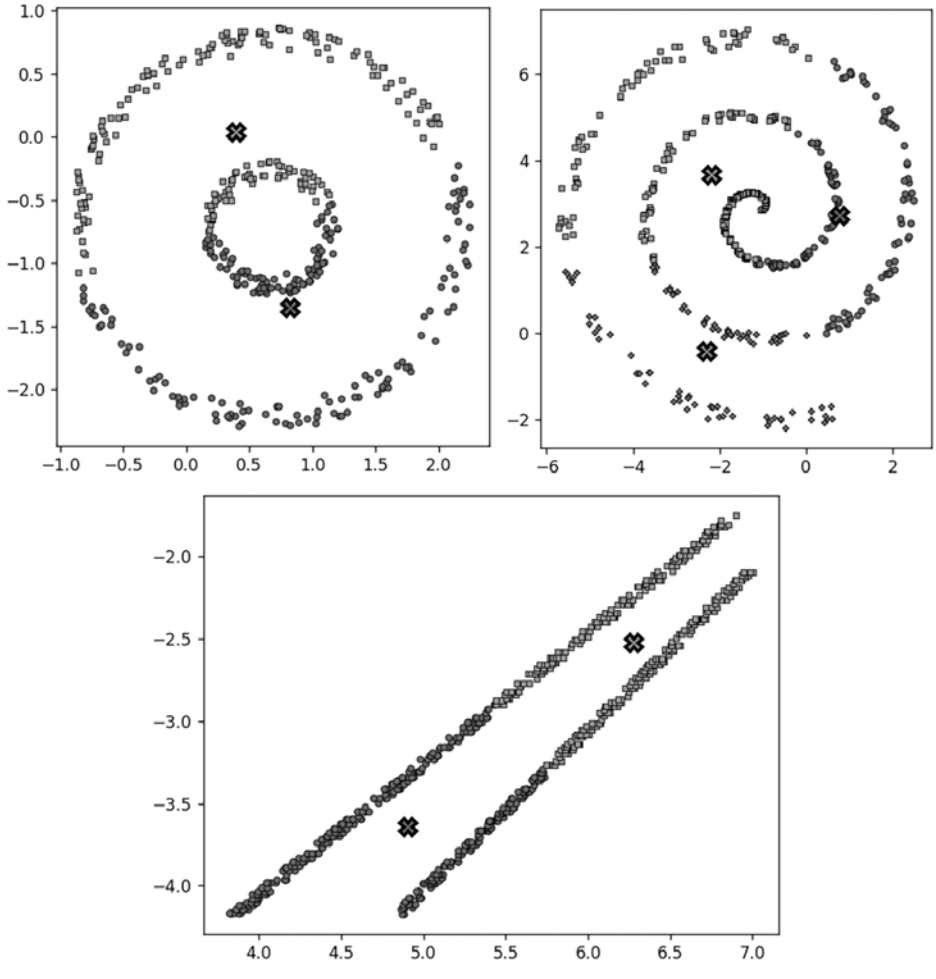


Рис. 12.6. Три примера, когда кластеризация методом k-средних дает далеко не оптимальный результат. *Слева:* два концентрических кольца (два центроида). *Справа:* спираль (три центроида: оптимальным решением здесь был бы один кластер). *Внизу:* два линейных кластера, расположенных близко друг к другу

Кластеры, неразделимые линейно, нельзя аппроксимировать сферическими кластерами, поэтому метод k-средних не способен разделять невыпуклые кластеры, например в форме двух концентрических колец. Более того, во всех ситуациях, когда форма кластеров не является сферической и точки невозможно правильно разделить путем определения ближайшего центроида, метод k-средних не сможет найти хорошее решение.

Другая проблема метода k-средних заключается в том, что количество кластеров является гиперпараметром, то есть алгоритм не может автоматически определить

правильное количество кластеров[1] и просто принимает количество центроидов как аргумент. Это означает, что в отсутствие некоторого понимания, полученного из знания предметной области и позволяющего правильно определить количество категорий для группировки данных, вам придется запустить алгоритм несколько раз, пробуя разные значения количества центроидов и сравнивая результаты с использованием какой-либо метрики (визуальное сравнение возможно только для наборов двумерных и трехмерных данных), и уже по полученным результатам выбрать наиболее подходящее количество кластеров. Мы поговорим об этом в разделе 12.5.

Эти недостатки, безусловно, ограничивают применимость метода *k*-средних, но, с другой стороны, есть области, в которых можно предположить и даже доказать, что данные хорошо моделируются с использованием сферических кластеров. Однако даже в лучшем случае или в промежуточных ситуациях, когда сферический кластер является просто хорошим приближением, имеет место еще одна проблема, ограничивающая применимость этого алгоритма: проклятие размерности.

12.2.2. Проклятие размерности снова в действии

Вы уже столкнулись с проклятием размерности в главе 9, когда знакомились с *k*-мерными деревьями — структурой данных, которая хорошо работает в пространствах низкой и средней размерности, но плохо — в пространствах высокой размерности.

Проявление этой проблемы в алгоритме *k*-средних не совпадение. Этот алгоритм фактически является эвристикой поиска, минимизирующей евклидово расстояние от точек до центроидов. Однако в пространствах с большим числом измерений:

- отношение объема к площади поверхности растет экспоненциально;
- большинство точек равномерно распределенного набора данных расположены дальше от центра масс и ближе к поверхности гиперсферы кластера;
- если данные равномерно распределены в гиперкубе (область, в которой значения признаков равномерно распределены в фиксированном диапазоне), то в пространствах с большим числом измерений большинство точек будет находиться ближе к граням гиперкуба;
- аппроксимация гиперкуба вписанной гиперсферой не будет учитывать бóльшую часть точек;
- чтобы включить все точки, нужна гиперсфера, описывающая гиперкуб, но, как было показано в подразделе 10.5.1, это влечет экспоненциальный рост объема пустого пространства с увеличением числа измерений;
- в *d*-мерном пространстве с $d >> 10$ при некоторых разумных предположениях о распределении данных проблема ближайшего соседа становится нечеткой[2], потому что при большом числе размерностей отношение между расстоянием до

---

[1] Как мы уже видели, метод *k*-средних может отбрасывать некоторые центроиды, но этот эффект очень ограничен и непредсказуем.

[2] *Beyer K. et al.* When is “nearest neighbor” meaningful? // International conference on database theory. — Berlin; Heidelberg: Springer, 1999.

самого ближнего и самого дальнего соседа становится почти равным 1. Например, если точки расположены на равном расстоянии друг от друга (скажем, по сетке), то $2d$ точек, ближайших к любому центроиду, будут находиться на одинаковом расстоянии от него. Это означает, что определение ближайшего соседа становится неоднозначным.

Проще говоря, для многомерных наборов данных, если только распределение кластеров не является строго сферическим, сферы, включающие все точки, становятся настолько большими, что почти наверняка будут пересекаться и иметь значительные объемы таких пересечений. Более того, для наборов данных с распределением, близким к равномерному, в случаях, когда используется много центроидов, поиск ближайшего центроида может давать неточный результат. Это, в свою очередь, приводит к более медленной сходимости, потому что некоторые точки могут переходить туда и обратно между почти одинаково близкими центроидами.

Поэтому важно помнить, что метод k-средних является хорошим вариантом только для наборов данных с низкой и средней размерностью (не более 20 измерений) и когда точно известно, что кластеры могут аппроксимироваться гиперсферами.

12.2.3. Анализ производительности метода k-средних

Однако, когда известно, что набор данных соответствует предварительным условиям применения метода k-средних, этот алгоритм может с успехом применяться и быстро давать хорошие результаты. Насколько быстро? Это сейчас и выясним.

Предположим, что есть набор данных с n точками, каждая из которых принадлежит d-мерному пространству (то есть каждая точка может быть представлена как кортеж из d действительных чисел), и требуется разделить точки на k разных кластеров.

Если внимательно изучить каждый шаг в листинге 12.1, то можно заметить, что:

- шаг случайной инициализации требует $O(n \times d)$ присваиваний (d координат для каждой из n точек);
- инициализация индексов кластеров требует $O(n)$ присваиваний;
- основной цикл повторяется m раз, где m — максимально допустимое количество итераций;
- связывание точек с центроидами требует $O(k \times n \times d)$ операций, так как для каждой точки нужно вычислить k d-мерных квадратов расстояний, а для получения каждого расстояния нужно выполнить $O(d)$ операций;
- сравнение двух классификаций точек требует $O(n)$ времени, но эту операцию можно амортизировать внутри метода, выполняющего разбиение;
- обновление центроидов требует для каждого центроида вычислить d (по одному для каждой координаты[1]) средних значений по не более чем n точкам, то есть всего

[1] На графических процессорах и процессорах, предназначенных для векторных вычислений, операции могут выполняться одновременно по всем координатам d-мерных кортежей с более высокой эффективностью.

потребуется $O(k \times n \times d)$ операций. Если предположить, что точки равномерно распределены между кластерами, то каждый кластер будет содержать не более n / k точек и, следовательно, среднее время работы в худшем случае становится равным $O(k \times (n / k) \times d) = O(n \times d)$.

Суммируя, получаем общее время работы алгоритма $O(m \times k \times n \times d)$, а для хранения классификации точек и списка центроидов потребуется $O(n + k)$ дополнительной памяти.

12.2.4. Ускорение метода k-средних с помощью k-мерных деревьев

Рассматривая код реализации алгоритма *k*-средних, я уже показал, что шаг разделения, на котором каждая точка связывается ровно с одним из центроидов, реализован как простой перебор всех комбинаций точек и центроидов. Теперь попробуем выяснить, можно ли ускорить его?

В подразделе 10.5.3 мы с вами видели, что при использовании в эвристике разделения метод *k*-средних может оптимизировать производительность SS⁺-деревьев, делая их более сбалансированными. Но верно ли обратное? Многие из вас наверняка поняли, куда я клоню: можно ли заменить поиск методом полного перебора чем-то более эффективным?

Если подумать, то на этапе разделения производится поиск ближайшего соседа среди множества центроидов, и мы уже знаем пару структур данных, способных ускорить этот поиск!

Но в этом случае вместо SS⁺-деревьев можно было бы попробовать *k*-мерные деревья, и на то есть три причины. Прежде чем прочитать их, попробуйте приостановиться и подумать, почему выбор пал на *k*-мерные деревья. Если вы не сможете вспомнить все три причины или вам не до конца понятно, почему эти причины имеют место, то прочитайте еще раз разделы 9.4 и 10.1, где эти концепции объясняются более подробно.

Итак, *k*-мерные деревья лучше подходят для поиска ближайшего центроида, чем SS⁺-деревья, по следующим причинам.

- Размер набора данных (центроидов) невелик, следовательно, значительна вероятность, что он уместится в памяти. В этом случае *k*-мерные деревья являются лучшим выбором, если также выполняются два других условия.
- Размерность области поиска невелика — от низкой до средней. Вы уже знаете, что это так (если, конечно, сделали свою домашнюю работу!), потому что метод *k*-средних тоже страдает проклятием размерности и не должен применяться к многомерным наборам данных.
- *k*-мерные деревья могут предложить теоретическую верхнюю границу для наихудшего случая, которая лучше, чем поиск полным перебором: $O(2^d + d \times \log(k))$ — для *k* *d*-мерных центроидов. В отличие от них, SS⁺-деревья не могут предложить ничего эффективнее, чем линейное время работы в худшем случае.

Кроме того, используемая структура данных должна воссоздаваться заново в каждой итерации основного цикла метода k-средних, поэтому алгоритму не придется иметь дело с динамическим набором данных, который со временем может привести к разбалансировке k-мерного дерева.

Необходимость создания нового набора данных в каждой итерации одновременно является одним из самых больших минусов рассматриваемого подхода, потому что требует платить дополнительную цену. Кроме того, для его хранения понадобится $O(k)$ дополнительной памяти. Это не изменит асимптотическую нагрузку алгоритма на память, но на практике будет иметь значение, особенно если k, число центроидов, велико.

Однако в большинстве приложений разумно ожидать, что $k << n$. Другими словами, количество центроидов будет на несколько порядков меньше количества точек.

В листинге 12.5 показано, как можно изменить псевдокод метода `classifyPoints`, чтобы использовать k-мерное дерево вместо поиска простым перебором.

Как видите, код получился намного короче, чем в листинге 12.3, потому что бóльшая часть сложностей поиска теперь инкапсулирована в классе `KdTree`.

Листинг 12.5. Метод classifyPoints, использующий k-мерные деревья

```
Инициализируем список центроидов пустым списком
Создаем экземпляр KdTree и инициализируем его списком центроидов
Метод classifyPoints принимает список точек из набора данных и текущий список центроидов. Возвращает список центроидов (технически — их индексов), связанных с каждой точкой в наборе данных

function classifyPoints(points, centroids)
  clusters ← []
  kdTree ← new KdTree(centroids)
  for i in {0, .., |points|-1} do
    clusters[i] ← kdTree.nearestNeighborIndex(point)
  return clusters

Цикл по индексам точек в наборе данных
Получаем индекс центроида, ближайшего к данной точке, обратившись к экземпляру KdTree
Возвращаем список кластеров
```

Обратите внимание: так как требуется узнать индекс центроида, ближайшего к каждой точке, предполагается, что объект `KdTree` может запоминать индексы своих точек в массиве инициализации и имеет метод, возвращающий индекс ближайшей точки, а не самой точки. Если нет, то эту проблему легко решить. Как вариант — хранить хеш-таблицу, отображающую центроиды в их индексы, и добавить дополнительный шаг для получения индекса центроида, возвращаемого экземпляром `KdTree`.

Для создания k-мерного дерева с центроидами понадобится $O(k \times \log(k))$ шагов, каждый из которых может потребовать до $O(d)$ операций (поскольку имеем дело с d-мерными точками). Соответственно, весь этап классификации потребует $O(d \times k \times \log(k) + n \times 2^d + n \times d \times \log(k)) = O(n \times 2^d + d \times (n + k) \times \log(k))$ шагов вместо $O(n \times k \times d)$ операций.

Формально можно точно определить условия, когда $O(n \times 2^d + d \times (n + k) \times \log(k)) < O(n \times k \times d)$; однако для наших целей можно неформально использовать интуицию, чтобы увидеть, что:

- $n \times 2^d < n \times k \times d \Leftrightarrow d << k$;
- $d \times (n + k) \times \log(k) < n \times k \times d \Leftrightarrow (n + k) \times \log(k) < n \times k \Leftrightarrow n < n \times k / \log(k) - k$. Легко показать, что это условие выполняется для $n > k$, но, построив график разницы между двумя сторонами, можно заметить, что для фиксированного n разница растет с ростом k, поэтому экономия более заметна, когда имеется много центроидов.

Если предположить, что кластеры равномерно распределены и каждый сдержит примерно n/k точек, то для выполнения всего алгоритма потребуется $O(m \times (n \times 2^d + d \times (n + k) \times \log(k) + n \times d)) = O(m \times (n \times 2^d + d \times (n + k) \times \log(k)))$. Таким образом, в теории и за вычетом постоянных множителей и деталей реализации для ускорения поиска ближайшего соседа кажется оправданным использование такой структуры данных, как k-мерные деревья.

Однако, как мы видели, если теоретический запас невелик, то иногда более сложная реализация дает асимптотическое превосходство в худших случаях только для больших, а иногда и очень больших объемов входных данных. Чтобы еще раз проверить, стоит ли на практике создавать и использовать k-мерное дерево для поиска в каждой итерации, я провел профилирование, результаты которого можно найти в репозитории книги[1].

Для этого я использовал реализацию на Python. Совершенно понятно, что полученные результаты верны только для данного языка и данной конкретной реализации. И все же они довольно показательны.

Для представления k-мерных деревьев я использовал реализацию SciPy[2], находящуюся в модуле `scipy.spatial`. Если вы читали предыдущие главы книги, то наверняка помните одно из упоминавшихся золотых правил. Прежде чем начинать разработку с нуля, посмотрите вокруг — возможно, уже есть что-то готовое и надежное. В этом случае реализация SciPy не только более надежна и эффективна, чем версия, которую я мог бы написать сам (потому что код SciPy тщательно протестирован и оптимизирован), но также предлагает метод, выполняющий поиск ближайшего соседа и возвращающий индекс (относительно порядка вставки) найденной точки. Это именно то, что нужно для нашего метода k-средних и избавляет от необходимости хранить дополнительные данные для определения индексов точек.

На рис. 12.7 показан результат профилирования метода `cluster_points` на Python, реализованного с поиском методом полного перебора и с помощью k-мерного дерева.

---

[1] http://mng.bz/A0x7.

[2] http://mng.bz/KM5g.

Внимательно рассмотрев четыре диаграммы на рис. 12.7, можно заметить, что линия, соответствующая реализации, которая использует *k*-мерное дерево, более стабильна при различных значениях *k*, а наклон верхней линии, соответствующей алгоритму поиска методом перебора, возрастает с увеличением значения *k*.

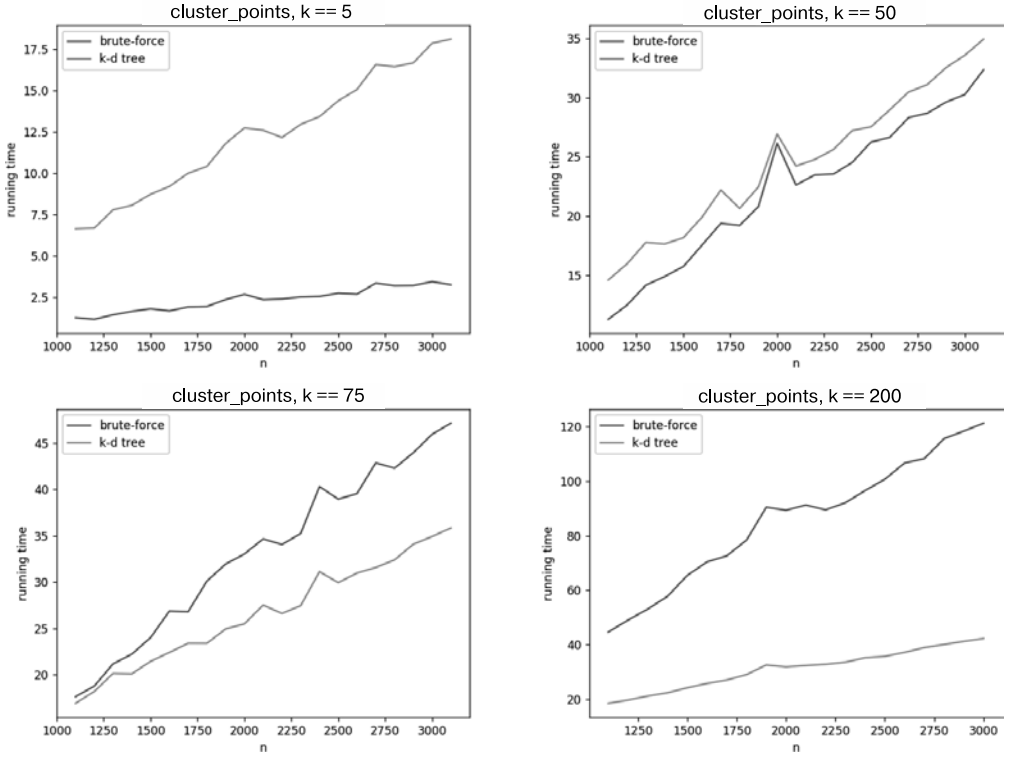


Рис. 12.7. Сравнение времени выполнения реализаций classifyPoints, использующих поиск методом перебора (темная линия) и с помощью k-мерного дерева (светлая линия). Время выполнения показано как функция от n, количества точек, для фиксированных значений k, количества центроидов. *Вверху слева:* k == 5, метод, использующий k-мерное дерево, всегда медленнее, и время его выполнения растет быстрее. *Вверху справа:* k == 50, метод с использованием k-мерного дерева все еще медленнее, но время его выполнения растет так же, как время выполнения метода полного перебора. *Внизу слева:* k == 75, оба метода выполняются за одинаковое время, но по мере роста n кривые расходятся, причем время выполнения реализации, использующей k-мерное дерево, растет медленнее. *Внизу справа:* k == 200, реализация на основе k-мерного дерева работает вдвое быстрее реализации на основе метода полного перебора даже для малых значений n и растет намного медленнее

Эта тенденция еще более очевидна, если посмотреть на данные под другим углом, оставив *n* фиксированным и построив график зависимости времени работы от *k*, как показано на рис. 12.8.

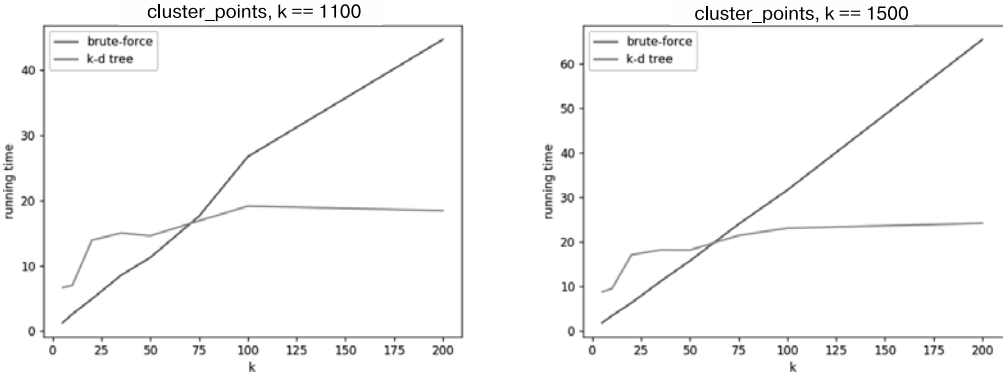


Рис. 12.8. Сравнение времени выполнения реализаций classifyPoints *(слева)* и k_means *(справа)*, использующих поиск методом перебора (темная линия) и с помощью k-мерного дерева (светлая линия). Время выполнения показано как функция от k, количества центроидов, для фиксированных значений n, размера набора данных. Диаграммы построены для n == 1500, но та же тенденция наблюдается для всех проверенных значений n > 1000. Очевидно, что метод classifyPoints занимает бóльшую часть времени выполнения этой реализации алгоритма k-средних

Теперь можно с уверенностью сказать, что по крайней мере для Python реализация подпрограммы разделения точек с использованием k-мерных деревьев имеет лучшую производительность, чем поиск методом перебора.

12.2.5. Заключительные замечания об алгоритме кластеризации методом k-средних

В заключение раздела, посвященного алгоритму кластеризации методом k-средних, подведем итоги:

- метод k-средних — это метод жесткой кластеризации на основе центроидов;
- при реализации со вспомогательным k-мерным деревом время работы алгоритма составляет $O(m \times (n \times 2^d + d \times (n + k) \times \log(k)))$;
- метод k-средних хорошо работает с наборами данных низкой и средней размерности, когда кластеры имеют сферическую форму и количество кластеров можно оценить *априори*, и показывает неплохие результаты, даже если данные в наборе распределены неоднородно;
- метод k-средних плохо работает с многомерными данными и в случаях, когда кластеры нельзя аппроксимировать гиперсферами.

Теперь, описав прародителя всех алгоритмов кластеризации, перепрыгнем на 40 лет вперед, к моменту изобретения следующего алгоритма, который будет представлен далее.

12.3. DBSCAN

Статья, описывающая алгоритм DBSCAN, была опубликована в 1996 году, в ней был представлен новый подход к решению этой задачи[1]. Название DBSCAN — это аббревиатура от Density-Based Spatial Clustering of Applications with Noise (плотностная пространственная кластеризация для приложений с шумами), и главное отличие этого подхода от метода *k*-средних ясно уже из названия. Метод *k*-средних основан на центроидах и поэтому строит кластеры как выпуклые множества вокруг точек, выбранных в качестве центроидов. Плотностный алгоритм определяет кластеры как наборы точек, находящиеся достаточно близко друг к другу, чтобы плотность точек в любой области кластера была выше определенного порога. Естественное расширение этого определения, кстати, вводит понятие шума (или *выбросов*) для точек, находящихся в областях с низкой плотностью. Чуть ниже мы формально определим обе категории, но прежде познакомимся с некоторыми общими соображениями.

Как и метод *k*-средних, алгоритм DBSCAN — это плоский алгоритм жесткой кластеризации, который связывает каждую точку не более чем с одним кластером (выбросы не связываются ни с одним кластером) со 100%-ной достоверностью, и все кластеры являются объектами одного уровня — иерархия групп точек не сохраняется.

В методе *k*-средних случайная инициализация центроидов играет настолько важную роль (правильный выбор ускоряет сходимость), что часто алгоритм запускается повторно несколько раз, чтобы можно было выбрать лучший вариант кластеризации. Ничего подобного нет в DBSCAN, где точки циклически переносятся несколько раз случайным образом. Но это и меньше влияет на конечный результат, если такое влияние вообще имеется; следовательно, этот алгоритм можно считать детерминированным[2].

Наконец, DBSCAN расширяет концепцию *односвязной кластеризации*[3] (Single-Linkage Clustering, SLC), вводя порог минимальной плотности точек, используемый при принятии решения об объединении двух точек в один кластер. Это уменьшает *эффект образования односвязной цепи* — худший побочный эффект SLC, — в результате которого независимые кластеры, соединенные тонкой линией (шумовых) точек, ошибочно классифицируются как один кластер.

---

[1] *Ester M. et al.* A density-based algorithm for discovering clusters in large spatial databases with noise // Kdd. — 1996 — Vol. 96. — No. 34. Описываемый подход тесно связан с другой статьей, опубликованной в 1972 году: *Ling R. F.* On the theory and construction of k-clusters // The Computer Journal. — 1972. — Vol. 15.4 — P. 326–332.

[2] Технически если запустить алгоритм DBSCAN дважды и он будет обходить точки в одном и том же порядке, то в обоих случаях получится один и тот же результат. Чтобы метод *k*-средних считать детерминированным, случайную инициализацию следует заменить детерминированной.

[3] Односвязная кластеризация — класс алгоритмов восходящей иерархической кластеризации, в которых на каждом шаге происходит слияние пары кластеров, оказавшихся на минимальном расстоянии (изначально каждая точка находится в своем кластере).

12.3.1. Прямая достижимость и достижимость по плотности

Чтобы понять, как работает алгоритм DBSCAN, начнем с нескольких определений. Для проверки их понимания и для справки используйте рис. 12.9.

- Точка p на рис. 12.9 называется основной, потому что на расстоянии ε от нее имеется не менее `minPoints` точек (включая саму точку p); в этом примере `minPoints==3`.
- Точка q *прямо достижима* из p, потому что находится на расстоянии ε от точки p, которая является основной точкой. Точка может быть достижима только из основной точки.
- Точка w просто *достижима* (или, что то же самое, *достижима по плотности*) из основной точки (например, p) через путь из основных точек $p = w_1 ... w_n = w$, если каждая точка w_{i+1} прямо достижима из w_i. Из предыдущего определения прямой достижимости следует, что все точки в пути, кроме w, должны быть основными точками.
- Любые две точки, достижимые друг для друга по плотности, по определению принадлежат одному и тому же кластеру.
- Если есть точка r, не достижимая ни из какой другой точки в наборе данных, то r (и все точки, подобные r) маркируются как выбросы (или, что то же самое, как шум).

Алгоритм построен на понятии основных точек: каждая из них имеет не менее определенного количества соседей, находящихся не далее определенного расстояния. Говоря иначе, основные точки — это точки в областях с плотностью не ниже минимальной.

Основные точки (такие как p, q и т. д. на рис. 12.9), достижимые (то есть смежные) по отношению друг к другу, принадлежат одному и тому же кластеру. Почему? Потому что предполагается, что области с высокой плотностью определяют кластеры (в отличие от большей части пространства с низкой плотностью). И все точки, находящиеся на расстоянии ε от центральной точки p, принадлежат тому же кластеру, что и p.

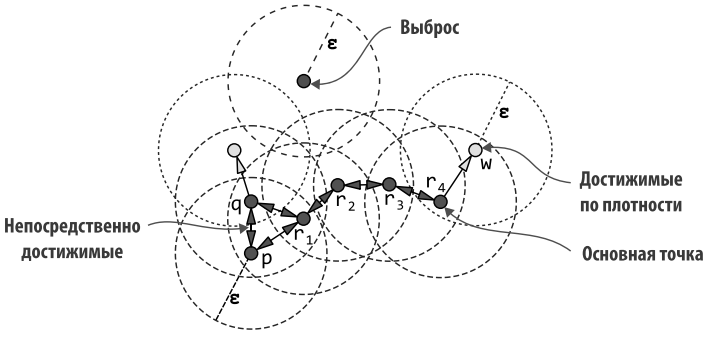


Рис. 12.9. Основные точки, непосредственно достижимые точки и точки, достижимые по плотности, при радиусе ε и пороге minPoints (минимальное количество точек в основной области), равном 3, согласно которому всякая основная точка должна иметь не менее двух соседей на расстоянии ε

12.3.2. От определений к алгоритму

Переход от определений к алгоритму на удивление прост.

Для заданной точки p нужно проверить, сколько она имеет соседей, лежащих в пределах радиуса ε. Если их больше определенного числа m, то p отмечается как основная точка и ее соседи добавляются в тот же кластер, иначе ничего не делается. На рис. 12.10 показан этот шаг, повторяющийся для каждой точки в наборе данных с помощью множества для определения точек, подлежащих обработке (в любом порядке) в текущем кластере.

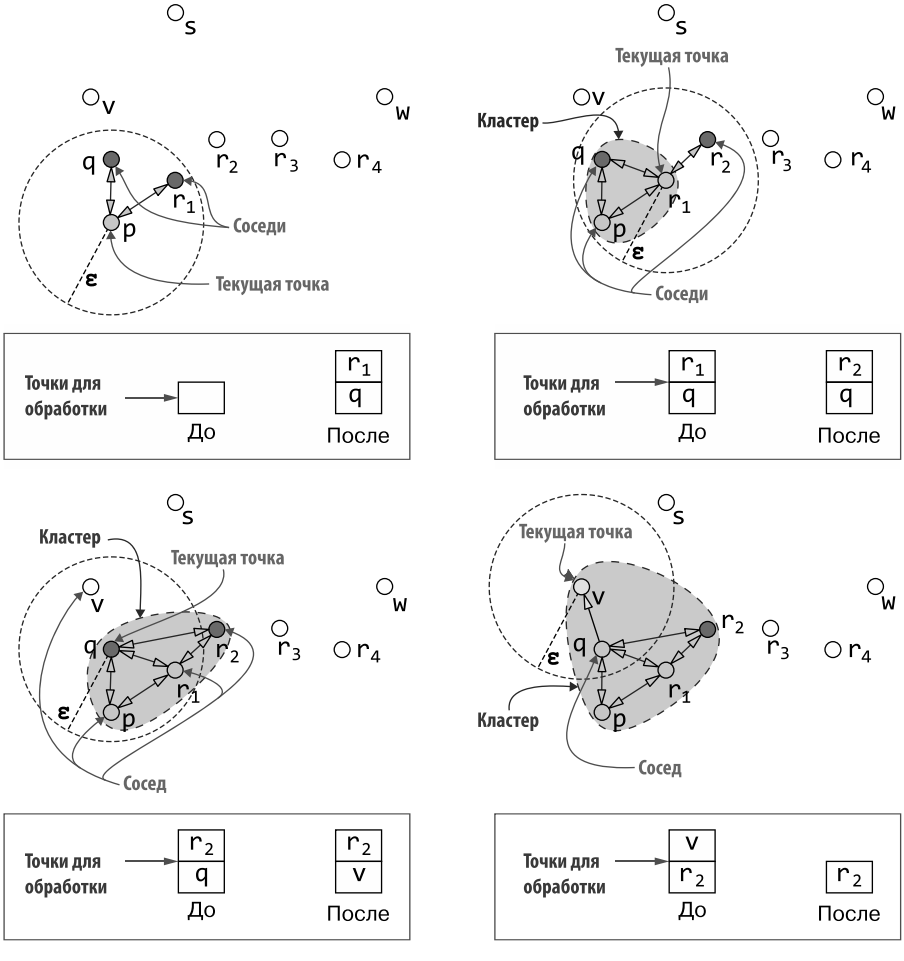


Рис. 12.10. Несколько шагов основного цикла алгоритма DBSCAN. При обработке основных точек (первые три шага на рисунке) все их необнаруженные соседи добавляются в текущий кластер и в множество (не в очередь) точек, подлежащих обработке. И наоборот, если текущая обрабатываемая точка не является основной, как v на последнем шаге, то в отношении нее никаких действий не предпринимается

Теперь возникает вопрос: что происходит при обработке точки w, которая не является основной точкой, но прямо достижима из основной точки p? Действительно ли не нужно предпринимать никаких действий при обработке w?

Как показано на рис. 12.9 и 12.10, если p — основная точка и w прямо достижима из нее, то расстояние между w и p должно быть не больше ε. Поэтому, проверяя точку p, добавим всех ее соседей в радиусе ε в один кластер с p, и, соответственно, w окажется в этом кластере.

А что, если есть две основные точки, p и q, достижимые друг из друга по плотности и обе достижимы из w? Тогда согласно определению получится цепочка основных точек $w_1 ... w_n$ между q и p, поэтому каждая основная точка в пути, в свою очередь, будет добавлена в один кластер с q и, наконец, p, когда настанет очередь w_n.

А что, если p и q — основные точки, недостижимые друг из друга, но достижимые из точки w? Может ли w стать основной точкой?

Доведем рассуждения *ad absurdum*[1]: предположим, что w достижима из основной точки p, которая обрабатывается до w. Следовательно, существует цепочка основных точек $p_1 ... p_n$, каждая из которых достижима из предыдущей, которая соединяет p с w.

Предположим также, что w к моменту обработки p_n уже была добавлена в другой кластер, отличный от кластера с точкой p. Это означает, что существует центральная точка q, из которой достижима точка w (и, следовательно, существует цепочка основных точек $q_1 ... q_k$ и т. д.), но p недостижима из q, поскольку они находятся в разных кластерах.

Теперь w может быть основной или неосновной точкой.

Если бы w была основной точкой, то существовала бы цепочка из основных точек q, $q_1 ... q_k$, w, $p_1 ... p_n$, p, где все точки достижимы друг из друга; следовательно, по определению p достижима из q, а это противоречит нашей первоначальной гипотезе.

Отсюда следует, что w не может быть основной точкой. Она неосновная точка, доступная как минимум из двух разных основных точек, что и показано на рис. 12.11.

В таких случаях достижимые точки могут добавляться в любой кластер, но разница будет примерно в одну точку. Это также означает, что два кластера разделены областью с плотностью ниже пороговой (хотя и не полностью пустой).

Порядок обработки точек не влияет на окончательный результат, потому что рано или поздно обнаружится, что все точки, достижимые по плотности, принадлежат одному кластеру. Тем не менее есть эффективный и неэффективный способы обработки точек. В зависимости от порядка, может понадобиться использовать разные методы отслеживания кластеров.

---

[1] От лат. reductio ad absurdum — «доведение до абсурда». Это логический прием, которым доказывается несостоятельность какого-либо утверждения, основанная на том, что в нем самом или в вытекающих следствиях обнаруживаются противоречия.

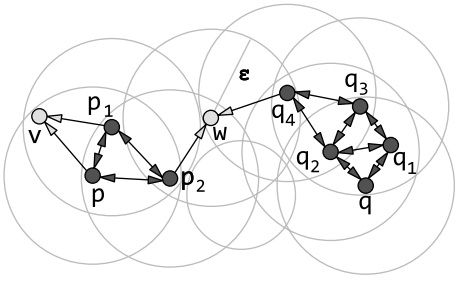


Рис. 12.11. Краевая точка w прямо достижима по крайней мере из двух разных кластеров. В этом примере порог minPoints установлен равным 4; путь от q до p выделен более жирными стрелками

Плохой способ — обработка точек в совершенно случайном порядке. В таком случае придется отслеживать кластеры, назначаемые каждой точке (изначально каждая точка находилась в своем кластере), а также решать, какие кластеры необходимо объединить (каждый раз, обрабатывая основную точку, нужно будет попытаться[1] объединить по меньшей мере `minPoints` – 1 кластеров). Это усложняет обработку и требует использования специальной структуры данных — непересекающегося множества, описанного в главе 5.

Если вместо этого обрабатывать соседей каждой основной точки p сразу после завершения работы с p, как показано на рис. 12.10, то станет возможно кластеры строить последовательно, увеличивая каждый кластер точка за точкой, пока в него нельзя будет добавить дополнительную точку без необходимости выполнять слияние кластеров или отслеживать историю слияний.

Следуя этому порядку, точки, подобные q и p на рис. 12.11, никогда не будут добавлены в один и тот же кластер, и на самом деле совершенно неважно, к какому из них будет присоединена (так называемая) краевая точка, такая как w. Только для краевых точек DBSCAN не является полностью детерминированным, потому что может добавить их в любой из кластеров, откуда они достижимы, и выбор кластера, куда они в конечном счете добавляются, зависит от порядка обработки точек в наборе данных.

Остался последний вопрос, на который нужно ответить: сколько раз повторять основной цикл DBSCAN? Число итераций точно совпадает с числом точек. Этим DBSCAN отличается от метода k-средних, выполняющего по всему набору данных множество итераций, состоящих из нескольких шагов. Метод k-средних — это эвристика поиска, корректирующая некоторые параметры для достижения локального минимума[2], а DBSCAN — это однопроходный детерминированный алгоритм, вычисляющий наилучшее разделение (и в то же время идентифицирующий выбросы) на основе плотности точек в разных областях.

---

[1] Некоторые соседи основной точки уже могли быть объединены.

[2] Как было показано в предыдущих разделах, функция стоимости, минимизируемая методом k-средних, — это евклидово расстояние до центроидов (и косвенно внутрикластерное евклидово расстояние).

12.3.3. И наконец, реализация

После описания общих принципов работы алгоритма можно перейти к реализации DBSCAN, показанной в листинге 12.6. В репозитории к книге на GitHub также можно найти реализацию на Python[1].

Листинг 12.6. Алгоритм кластеризации DBSCAN

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#dbscan.

В главе 11 отмечалось, что при кластеризации часто используются такие структуры данных, как k-мерные и SS-деревья. В методах k-средних и DBSCAN используется многомерная структура индексации для ускорения поиска по диапазону. Возможно, вы поняли это по определению *основных точек*: для того чтобы определить, является ли точка p основной, нужно проверить, сколько точек из набора данных находится в ее окрестностях с определенным радиусом.

Однако, в отличие от метода k-средних, где выполняется поиск ближайших соседей, алгоритм DBSCAN выполняет поиск по диапазону, чтобы отыскать все точки, имеющиеся в гиперсфере.

В репозитории книги также можно найти реализацию на Python[1] и блокнот Jupyter Notebook[2] для экспериментов с алгоритмом.

Очевидно, что в обоих алгоритмах, DBSCAN и k-средних, чтобы проверить окрестности каждой точки, можно использовать простой линейный поиск перебором. Однако, если в методе k-средних ускорение можно рассматривать просто как приятный бонус, не имеющий большого значения (и во многих реализациях k-средних об этом ускорении не беспокоятся), то в алгоритме DBSCAN прирост производительности чрезвычайно важен. Догадываетесь почему? Прежде чем прочитать объяснение, попробуйте приостановиться ненадолго и поразмышлять.

Отличие DBSCAN в том, что поиск выполняется по всему набору данных, тогда как в методе k-средних производится поиск только среди ближайших k центроидов (и обычно $k << n$).

В наборе данных с n точками разница во времени работы всего алгоритма будет равна разности между $O(n^2)$ и $O(n \times \log(n))$, если допустить, что поиск по диапазону занимает $O(\log(n))$[3]. Напомню, что для набора с одним миллионом точек это означает сокращение с $\sim 10^{12}$ (триллиона) операций до $\sim 6 \times 10^6$ (шести миллионов).

12.3.4. Достоинства и недостатки DBSCAN

Здесь уже были упомянуты несколько характерных особенностей DBSCAN, помогающих преодолеть некоторые ограничения, свойственные другим алгоритмам кластеризации, таким как метод k-средних или односвязная кластеризация. Перечислю их кратко:

- DBSCAN может сам определить количество кластеров (с учетом гиперпараметров, заданных на запуске), тогда как метод k-средних требует, чтобы это число было передано в виде параметра;

---

[1] http://mng.bz/ZPza.

[2] http://mng.bz/RXEO.

[3] Время выполнения такого поиска, как было показано в главе 9, не может быть лучше, чем $O(n)$ в худшем случае. Однако время поиска в диапазоне по гиперсфере зависит от радиуса сферы, и при определенных предположениях о значении ε величина $O(\log(n))$ может оказаться точной оценкой среднего времени выполнения.

- DBSCAN требует всего два параметра, которые также можно получить из предметной области (возможно, путем предварительного сканирования набора данных для сбора статистики);
- DBSCAN может обрабатывать шумы, идентифицируя выбросы;
- DBSCAN способен находить кластеры произвольной формы и выделять нелинейно разделимые кластеры (взгляните на рис. 12.12 и сравните его с рис. 12.6, где приводятся результаты кластеризации методом *k*-средних);
- путем настройки параметра `minPoints` можно уменьшить эффект образования односвязной цепи;
- алгоритм почти полностью детерминирован, и порядок обработки точек почти не имеет значения. Другой порядок может изменить привязку точек, находящихся на краю кластеров, только когда они одинаково близки к нескольким кластерам (как было показано на рис. 12.11).

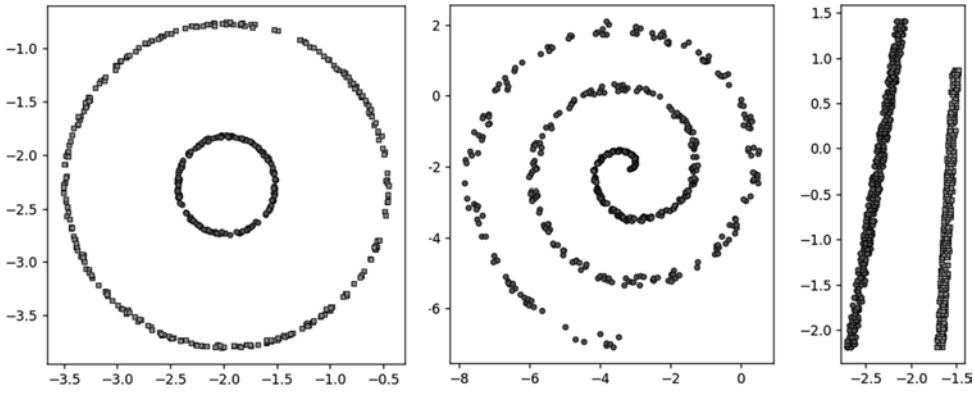


Рис. 12.12. Три примера наборов данных, которые метод k-средних не смог кластеризовать правильно (см. рис. 12.6). Алгоритм DBSCAN с правильно подобранными параметрами способен дать правильное разделение на кластеры во всех трех случаях

Но на этом хорошие новости заканчиваются. Как вы понимаете, у каждой розы есть шипы, и у алгоритма DBSCAN есть недостатки, перечислю их:

- DBSCAN *почти* полностью детерминирован, но не полностью. В некоторых случаях могут возникать проблемы, если точки на границе двух или более кластеров будут случайно назначаться одному из них;
- DBSCAN тоже страдает от проклятия размерности. Если используемой метрикой является евклидово расстояние, то, как было показано в подразделе 12.2.2, в многомерных пространствах все соседи точки находятся на одном и том же расстоянии и функция расстояния становится практически бесполезной. К счастью, в DBSCAN можно использовать и другие метрики;
- если в наборе данных есть области с различной плотностью, становится сложно, а иногда невозможно выбрать такие параметры ε и `minPoints`, чтобы все кла-

стеры были разделены правильно. На рис. 12.13 показан пример чувствительности результата, полученного с помощью DBSCAN, к выбору параметров, а на рис. 12.14 — другой пример с областями разной плотности, когда невозможно подобрать значение эпсилон, при котором обе области будут разделены правильно;

- в связи с этим аспектом одна из проблем успешного применения этого алгоритма заключается в сложности выбора оптимальных значений его параметров в отсутствие предварительных знаний о наборе данных.

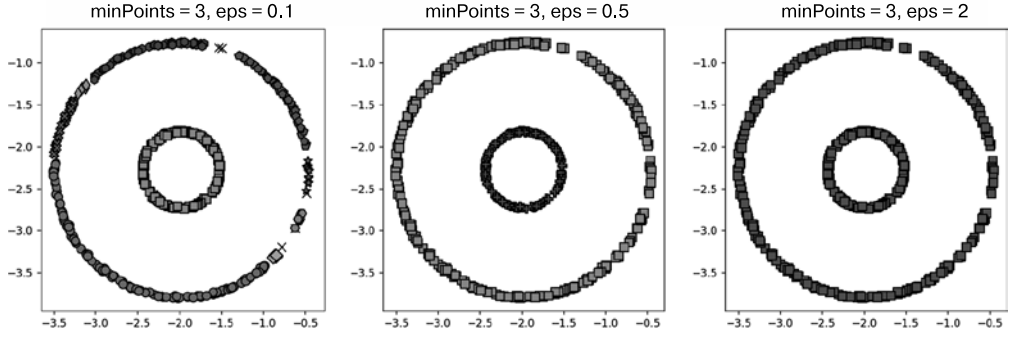


Рис. 12.13. На примере первого набора данных на рис. 12.12 можно видеть, насколько результат алгоритма DBSCAN чувствителен к выбору параметров (в частности, ε). Во всех примерах DBSCAN применялся к одному и тому же набору данных со значением 3 в параметре minPoints (размерность пространства плюс 1). *Слева:* использование слишком малого значения ε приводит к тому, что плотные области становятся мизерными, поэтому данные разбиваются на большое количество маленьких кластеров. Несколько точек, отмеченных крестиками, были даже классифицированы как выбросы. *По центру:* когда значения параметров выбраны правильно, DBSCAN идеально разделяет данные на два концентрических кольца. *Справа:* когда радиус ε устанавливается настолько большим, что плотная область внутренних точек оказывается слишком широкой и охватывает точки внешнего кольца, весь набор данных ошибочно оказывается в одном кластере

Настройка гиперпараметров[1], как это часто бывает в машинном обучении, имеет решающее значение и не всегда проста. Выбор значения `minPoints` обычно не представляет большой сложности. Как правило, для d-мерных наборов данных выбирается значение `minPoints` $> d$, при этом часто идеальными оказываются значения, близкие к произведению $2d$. Для особенно зашумленных наборов данных рекомендуется несколько увеличивать этот параметр, чтобы усилить фильтрацию шума. Выбор правильного значения ε, напротив, часто оказывается очень сложной задачей и требует либо глубоких знаний предметной области, либо приложения значительных усилий для проверки разных значений. На рис. 12.13 показано, как при фиксированном значении `minPoints` слишком малые (для заданного набора данных) значения ε

[1] В машинном обучении параметры, передаваемые алгоритму и фиксируемые до начала обучения, часто называют гиперпараметрами, чтобы отличить их от параметров модели (например, весов нейронной сети), вычисляемых алгоритмом.

вызывают чрезмерную фрагментацию набора данных, а слишком большие дают противоположный эффект, снижая способность алгоритма обнаруживать действительно разные кластеры.

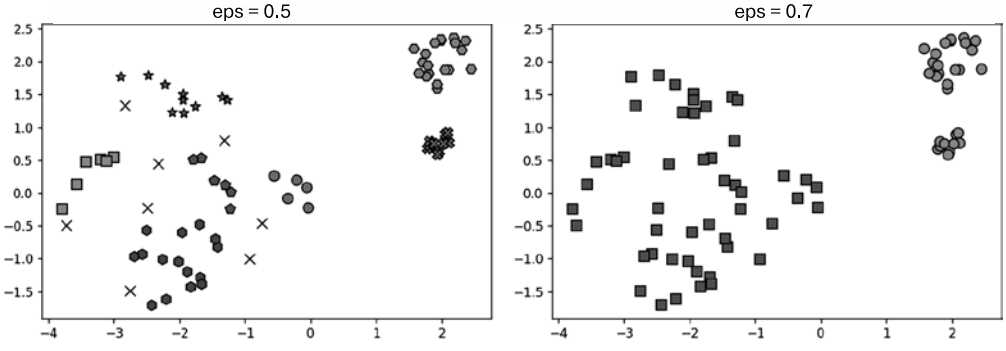


Рис. 12.14. Пример набора данных с областями неоднородной плотности, когда невозможно найти единственное значение ε, подходящее для всех областей. В наборе данных имеется кластер с низкой плотностью в левой области и два кластера с высокой плотностью в правом верхнем углу, расположенных близко друг к другу. *Слева:* при маленьком значении ε кластер с низкой плотностью разбивается на множество мелких кластеров, окруженных шумом. *Справа:* при более высоких значениях два кластера справа объединяются. Не существует значения ε, правильно разделяющего эти кластеры и одновременно обеспечивающего распознавание кластера слева как единого целого

Очевидно, что необходимость такой настройки возвращает нас к ситуации, напоминающей применение метода *k*-средних, когда нужно заранее знать, сколько кластеров должно быть. В этом двумерном примере довольно легко найти «правильное» число кластеров, но при переходе в многомерные пространства теряется возможность использовать интуицию и определять правильные значения для гиперпараметров алгоритма, опираясь на желаемое количество кластеров.

В следующих двух разделах рассмотрим два разных способа подбора гиперпараметров и решения связанных с ними проблем.

12.4. OPTICS

Как было показано в предыдущем разделе, DBSCAN — мощный алгоритм, способный выявлять нелинейно разделимые кластеры любой формы. Однако у него есть слабое место — выбор параметров, определяющих пороги плотности. В частности, иногда трудно найти оптимальное значение для эпсилон, радиуса области основной точки, определяющего достижимость точек друг для друга. Когда в наборе данных имеются области с разной плотностью, это затрудняет и даже делает невозможным выбор значения, одинаково хорошо подходящего для всего набора данных (как показано на рис. 12.14).

И неважно, как будет выбираться это значение — вручную или полуавтоматически (см. раздел 12.5). Одного только алгоритма DBSCAN недостаточно для обработки неоднородных наборов данных.

Прошло совсем немного времени, прежде чем ученые-информатики придумали новую идею, способную помочь в таких ситуациях. Она состояла в том, чтобы добавить важный шаг «упорядочения точек для определения структуры кластеризации» (Ordering Points To Identify the Clustering Structure), который авторы идеи превратили в аббревиатуру OPTICS[1] и использовали ее для названия изобретенного ими алгоритма.

Обсуждая DBSCAN, мы говорили о порядке обработки точек. Для DBSCAN имеет значение только тот факт, что если точки находятся в одном кластере (то есть основные точки, достижимые друг для друга), то обрабатываются вместе. Однако, как мы видели, это больше вопрос оптимизации, избавляющей от необходимости хранить непересекающиеся множества и позволяющей объединять кластеры при обнаружении пары прямо достижимых точек, потому что порядок обработки может влиять только на классификацию неосновных точек на краях кластеров.

Идея OPTICS состоит в том, что порядок имеет значение и, в частности, имеет смысл продолжать расширять «границы» текущего кластера, добавляя в него необработанную точку, ближайшую к кластеру (если она достижима из кластера).

На рис. 12.15 показан упрощенный сценарий, иллюстрирующий эту концепцию. Чтобы выбрать правильную точку, очевидно, необходимо отслеживать расстояния до всех необнаруженных соседей кластера[2], и для этой цели авторы статьи ввели два определения: каждой точке они присвоили основное расстояние и расстояние досягаемости.

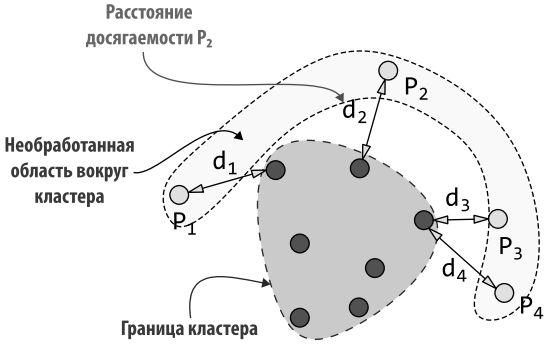


Рис. 12.15. Учет расстояния до необнаруженных точек (от P_1 до P_4) от определенного кластера (точек внутри него). Ключевая идея в OPTICS — «обнаружение» этих точек от ближней до самой дальней; в этом примере P_3 будет обработана следующей

[1] *Ankerst M. et al.* OPTICS: ordering points to identify the clustering structure // ACM Sigmod record. — 1999. — Vol. 28. — No. 2.

[2] Здесь под «соседом кластера» понимается точка, которая является соседом (и, следовательно, достижимой) любой точки, находящейся в данный момент в кластере.

12.4.1. Определения

Основное расстояние точки p — минимальное расстояние, при котором эта точка считается основной точкой. Определение основной точки, данное в описании алгоритма DBSCAN в разделе 12.3, зависит от параметров ε, радиуса плотных областей и `minPoints` (сокращенно M в следующих формулах), минимального количества точек, которые должны иметься в плотной области, поэтому подобные определения тоже зависят от гиперпараметров:

$$\text{Основное расстояние}_{\varepsilon, M}(p) = \begin{cases} \text{не определено, если } |N_\varepsilon(p)| < M; \\ M\text{-е наименьшее расстояние в } |N_\varepsilon(p)|, \text{если иначе.} \end{cases}$$

Естественно, если точка не является основной (и, следовательно, в ее ε-окрестности находится меньше M точек), то расстояние до нее не определено. Наоборот, ее основное расстояние будет расстоянием до M-го ближайшего соседа[1]: в этом случае основное расстояние можно интерпретировать как минимальное значение, которое можно присвоить параметру ε, чтобы p могла считаться основной точкой (при фиксированном значении `minPoints`).

После определения основного расстояния его можно использовать для задания метрики нового типа — расстояния достижимости между двумя точками:

$$\text{Расстояние достижимости}_{\varepsilon, M}(p) = \\ = \begin{cases} \text{не определено, если } |N_\varepsilon(p)| < M; \\ \max\left(\text{основное расстояние}_{\varepsilon, M}(q), \text{расстояние}(q, p)\right), \text{если иначе.} \end{cases}$$

Расстояние достижимости не симметрично. Для точки q расстояние достижимости p от q можно интерпретировать как минимальное значение ε, при котором p является достижимой по плотности из q. Поэтому оно должно быть не меньше основного расстояния p (для ближайших `minPoints` к p) или фактического расстояния между p и `q`. Более того, если p и q являются ближайшими соседями (в частности, если для p нет точки ближе, чем q), то это значение является наименьшим, которое можно присвоить параметру ε, чтобы p и q принадлежали одному кластеру.

12.4.2. Алгоритм OPTICS

Опираясь на эти два определения, легко описать сам алгоритм. Его основная идея аналогична DBSCAN: точки добавляются в текущий кластер, если они доступны из точек, уже находящихся в кластере. Однако сам алгоритм OPTICS имеет существенные

[1] Предполагается, что p рассматривается как собственный сосед... Мы уже договорились: чтобы p считалась основной точкой, ее должно окружать не менее M точек в радиусе ε, поэтому для вычисления основного расстояния берем расстояние до $(M - 1)$-го соседа.

отличия от DBSCAN. Во-первых, он производит не просто плоское разбиение точек, а строит иерархическую кластеризацию. Мы вернемся к этому чуть позже.

OPTICS принимает те же параметры, что и DBSCAN, но ε в данном случае имеет другое значение. Это самый большой радиус, который учитывается при определении основного расстояния и расстояния достижимости. По этой причине (и для устранения неоднозначности, как будет показано далее) в оставшейся части главы, обсуждая алгоритм OPTICS, будем называть этот параметр ε_{MAX}. Используя этот подход, алгоритм строит кластеры на основе плотности для бесконечного числа значений ε (все значения между 0 и ε_{MAX}) и, следовательно, для бесконечного числа плотностей.

В формулах предыдущего раздела эти расстояния считаются неопределенными, если ε-окрестность точки не имеет достаточного количества соседей. Это также означает, что расстояние достижимости от точки q (до любой другой основной точки) будет неопределенным, если это не основная точка с $M - 1$ соседями в радиусе ε_{MAX}.

Следовательно, чем больше значение ε_{MAX}, тем меньше точек будет помечено как шум и тем больше будет открытых вариантов благодаря большему основному расстоянию и тем больше будет точек в пределах определенного расстояния достижимости.

Однако чем больше радиус плотных областей, тем больше окажется точек в этих областях и тем медленнее будет работать алгоритм.

Фактически в основной части алгоритма для каждой обрабатываемой точки необходимо обновить расстояние достижимости всех необнаруженных точек в ее ε-окрестности. Чем больше будет таких точек, тем медленнее будет работать алгоритм.

Как уже отмечалось, основной цикл OPTICS увеличивает текущий кластер[1] C, добавляя ближайшую точку из необработанной области вокруг кластера, то есть точку с наименьшим расстоянием достижимости от любой точки, уже находящейся в C. Кластер формируется так же, как в алгоритме DBSCAN, потому что в него добавляются все точки, достижимые из начального зерна кластера (первой точки, добавленной в новый кластер). Однако формирование этих кластеров не является прямой задачей OPTICS.

Когда новая точка «обнаружена» и обработана, расстояние ее достижимости от кластера фиксируется. Это расстояние от кластера до точки не следует путать с расстоянием достижимости между двумя точками. Для точки q и кластера C расстояние достижимости от C до q определяется как минимальное расстояние достижимости между p и q для всех точек p в C:

$$\text{Расстояние достнжимостн}_{\varepsilon, M}(q, C) =$$
$$= \min\left(\text{расстоянне достнжимостн}_{\varepsilon, M}(q, p) \forall p \middle| p \varepsilon C\right).$$

[1] Технически, как уже говорилось, он одновременно конструирует бесконечное число кластеров для всех возможных значений плотности, но для простоты описания сосредоточимся на одном кластере, который был бы получен при $\varepsilon = \varepsilon_{MAX}$.

Если при обработке точки p обновлять расстояния достижимости всех точек в ее ε-окрестности и сохранять в приоритетной очереди все необнаруженные точки, находящиеся в окрестностях любой точки в C, то можно быть уверенными, что:

- расстояние достижимости всех точек на границе кластера (всех точек, являющихся соседями хотя бы одной точки в кластере) сохраняется правильно;
- сохраненное значение является наименьшим из расстояний достижимости от любой из уже обработанных точек до q (хотя нас интересуют только текущие кластеры);
- вершина очереди содержит ближайшую точку к кластеру C:
  - технически на вершине очереди находится точка q с соответствующим значением ε_q (наименьшее значение ε, при котором q все еще достижимо из C), таким, что учитываются любые другие точки w на границе C, $\varepsilon_q \leq \varepsilon_w$;
- следовательно, любая из необработанных точек будет иметь как минимум такое же расстояние достижимости от C или больше.

В листингах 12.7 и 12.8 показана реализация алгоритма OPTICS на псевдокоде, а в репозитории книги на GitHub[1] можно найти реализацию на Python и блокнот Jupyter Notebook для экспериментов с алгоритмом.

Листинг 12.7. Алгоритм кластеризации OPTICS

```
function optics(points, epsMax, minPoints)
  reachabilityDistances ← null (∀ p ε points)
  ordering ← []
  kdTree ← new KdTree(points)
  for p in points do
    if p in ordering then
      continue
    ordering.insert(p)
    neighbors ← kdTree.pointsInSphere(p, epsMax)
    if |neighbors| >= minPoints then
      toProcess ← new PriorityQueue()
      toProcess, reachabilityDistances ← updateQueue(
          p, kdTree, toProcess,
          reachabilityDistances, epsMax, minPoints)
```

Инициализируем нулем расстояние достижимости для каждой точки

Метод optics принимает список точек, (максимальный) радиус плотной области, определяющей основные точки, и минимальное количество точек, которое должно находиться в плотной области, чтобы точку можно было назвать основной. Эта функция возвращает кортеж: массив точек (индексов) в порядке обработки и второй массив с расстояниями достижимости для всех точек

Инициализируем порядок обработки точек. Изначально массив пуст; элементы будут добавляться в него по мере их обработки

Создаем k-мерное дерево для ускорения поиска по диапазону. Дерево инициализируется полным набором данных

Выполняем поиск по диапазону, чтобы выбрать все точки, попадающие в гиперсферу с центром в p. В этой реализации предполагается, что функция включит в возвращаемый список также саму точку p (согласно реализации в листинге 9.11)

Проверяем, обработана ли точка p; если обработана, то просто пропускаем ее

Создаем новую приоритетную очередь для точек, достижимых из p, которые будут обработаны следующими

Иначе добавляем ее в массив как следующую для обработки

Цикл по всем точкам в наборе данных

Обновляем очередь (добавив все точки, прямо достижимые из p), а также обновляем расстояние достижимости для соседей p

Проверяем, является ли p основной точкой, то есть вернул ли вызов pointsInSphere в строке 9 число, которое не меньше minPoints

[1] http://mng.bz/j4V8.

```
while not toProcess.isEmpty() do
  q ← toProcess.top()
  ordering.insert(q)
  toProcess, reachabilityDistances ← updateQueue(
    q, kdTree, toProcess, ordering,
    reachabilityDistances, epsMax, minPoints)
return ordering, reachabilityDistances
```

Цикл по всем точкам в очереди, упорядоченным по расстоянию достижимости

Возвращаем точки из набора данных и расстояния достижимости

Обновляем очередь (добавив все точки, прямо достижимые из q) и расстояние достижимости для соседей q

Добавляем q как следующую точку для обработки

Получаем из очереди точку q с наименьшим расстоянием достижимости (из множества уже обработанных точек)

Листинг 12.8. Метод updateQueue для алгоритма OPTICS

Выполняем поиск по диапазону, чтобы отобрать все точки в гиперсфере с центром в p

Метод updateQueue принимает точку p, k-мерное дерево со всеми точками из набора данных, очередь точек, достижимых из текущего кластера, массив с уже обработанными точками, массив с расстояниями достижимости для набора данных и параметры: эпсилон и минимальное количество точек. Возвращает кортеж с очередью и массивом расстояний достижимости (возможно, обновленный в процессе выполнения)

Проверяем, является ли p основной точкой (это имеет место, только когда вызов в строке 2 возвращает не менее minPts точек). Если нет, то просто возвращаемся, не выполняя никаких действий

```
function updateQueue(p, kdTree, queue, processed, rDists, eps, minPts)
  neighbors ← kdTree.pointsInSphere(p, eps)
  if |neighbors| < minPoints then
    return queue, rDists
  for q in neighbors do
    if q in processed then
      continue
    newRDist = max(coreDistance(p, eps, minPts), distance(p, q))
    if rDists[q] == null then
      rDists[q] ← newRDist
      queue.insert(q, newRDist)
    elsif newRDist < rDist[q] then
      rDist[q] ← newRDist
      queue.update(q, newRDist)
  return queue, rDists
```

Цикл по всем необработанным точкам в ∈-окрестности точки p

Если да, то q еще не добавлена в очередь, поэтому добавляем ее сейчас и устанавливаем расстояние достижимости для q

Проверяем, равно ли расстояние достижимости q из ранее обработанных точек значению null

Если это расстояние меньше, то обновляем расстояние достижимости и приоритет в очереди для q

Возвращаем обновленную очередь и массив с расстояниями достижимости

Иначе сравниваем расстояние достижимости q из p с расстояниями достижимости для ранее обработанных точек (возможно, что q ближе к p, чем эти точки)

Вычисляем расстояние достижимости q из p

Основу алгоритма (также показанного на рис. 12.16 в упрощенном примере) составляет цикл по всем точкам в наборе данных. Точки обрабатываются группами смежных достижимых точек (начальная точка выбирается случайным образом или в соответствии с ее положением в наборе данных), а неосновные точки «пропускаются» (точно так же, как в алгоритме DBSCAN).

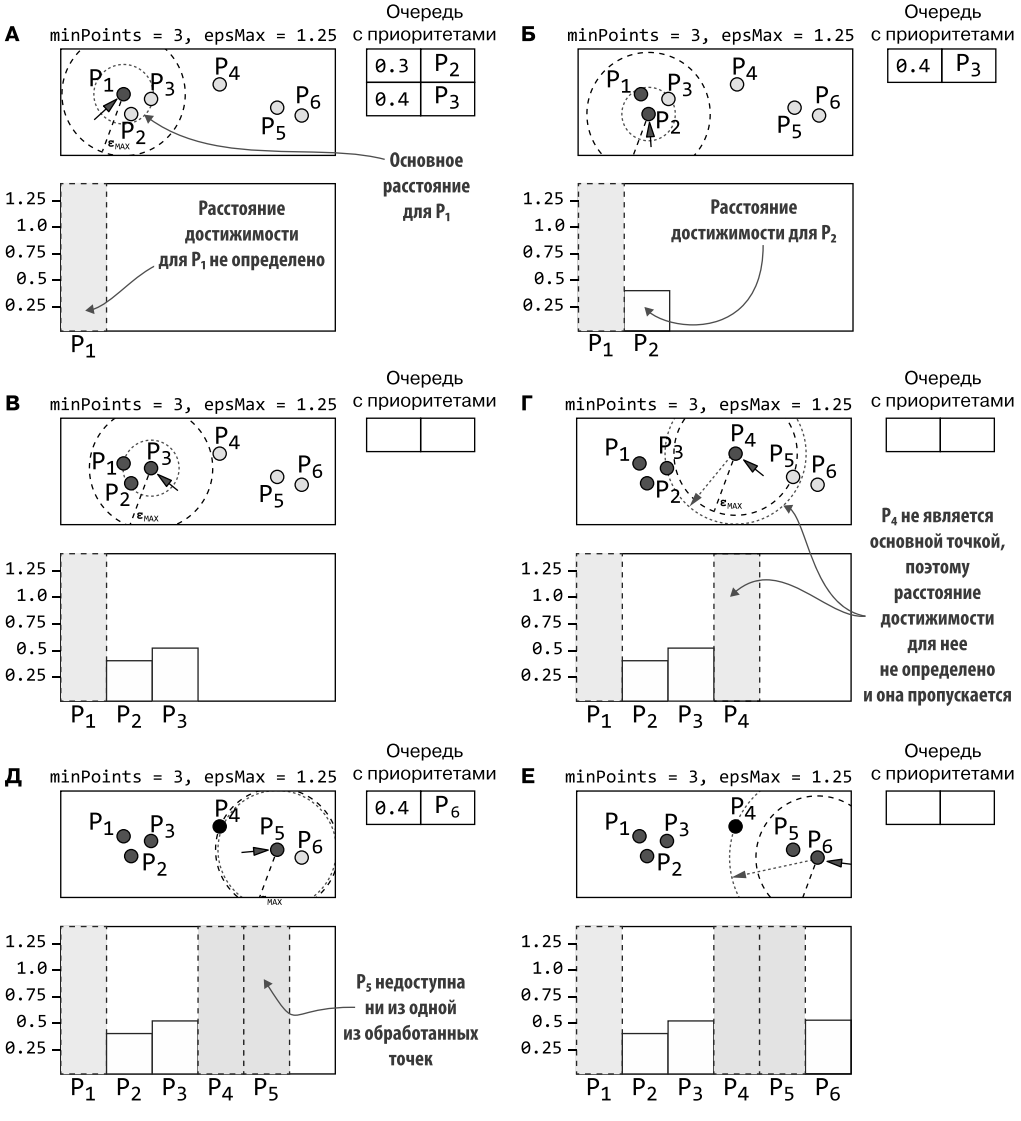


Рис. 12.16. Пример построения порядка обработки и списка расстояний достижимости алгоритмом OPTICS. Обратите внимание, что первая точка выбирается совершенно произвольно, возможен даже случайный выбор. Здесь принято решение начать с P_1 просто для удобства, чтобы сделать пример более ясным. Точно так же на шаге *E* можно было бы выбрать P_6 вместо P_5: в этом случае мы не смогли бы вычислить расстояние достижимости от P_6 до P_5 (или наоборот). Выбор большего значения для ε_{MAX} поможет избежать таких ситуаций

Точки, достижимые из уже обработанных точек, сохраняются в приоритетной очереди и извлекаются в соответствии с их расстояниями достижимости (точки с меньшими расстояниями достижимости оказываются ближе к началу).

До сих пор нами не рассмотрен ключевой фрагмент кода — метод `updateQueue`. Именно в нем обновляются расстояния достижимости (вместе с очередью приоритетов). Листинг 12.8 устраняет этот пробел. Основная задача метода `updateQueue` — обойти все точки q в ε-окрестности точки p (которая обрабатывается в данный момент) и обновить расстояния достижимости, проверяя, находится ли текущая точка ближе к точке p, чем любая другая, ранее обработанная точка.

Как показано на рис. 12.16, для отслеживания (промежуточных) расстояний достижимости используется очередь с приоритетами, и эти расстояния не изменяются, пока точка обрабатывается. Более того, расстояние достижимости первой обрабатываемой точки для каждого из кластеров заведомо будет либо неопределенным, либо больше ε_{MAX}.

Я понимаю, что до сих пор дискуссия носила преимущественно абстрактный характер. Чтобы понять истинную цель вычисления всех этих расстояний достижимости и порядка обработки набора данных, а также их применение для иерархической кластеризации, нужно забежать немного вперед и взглянуть на рис. 12.17, где показана диаграмма достижимости, которую можно считать результатом алгоритма OPTICS (вследствие чего после выполнения основного алгоритма требуется сделать еще один шаг, как будет показано в следующем разделе).

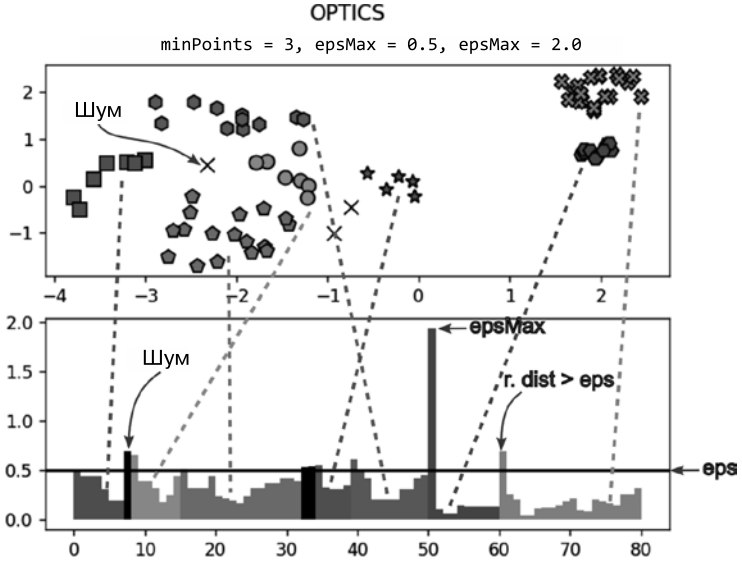


Рис. 12.17. Диаграмма достижимости. На верхней диаграмме показан набор данных после кластеризации, а на нижней — расстояния достижимости точек в наборе данных в порядке их обработки алгоритмом OPTICS. Выбросы показаны черным цветом (на верхней диаграмме отмечены крестиками). Точки, принадлежащие одному кластеру, окрашены одним цветом (или одним оттенком серого, если вы читаете черно-белую печатную версию книги) на обеих диаграммах (пунктирные линии соединяют соответствующие кластеры). Расстояния достижимости вычисляются алгоритмом OPTICS с учетом параметров minPoints и ε_{MAX}. Кроме того, существует параметр ε, определяющий порог для расстояний достижимости и, следовательно, управляющий фактическим разбиением на кластеры

12.4.3. От расстояния достижимости к кластеризации

Первое, на что следует обратить внимание на рис. 12.17, — диаграмма определяется тремя параметрами: `minPoints` и ε_{MAX} — это аргументы алгоритма OPTICS, но есть и еще одно значение ε, которое используется как порог для расстояния достижимости. Если представить этот порог как $\varepsilon \leq \varepsilon_{MAX}$, становится очевидно, что он применяется на шаге обработки результатов OPTICS, который выполняется отдельно (он описывается чуть ниже).

Установка порога ε для расстояния достижимости предполагает сравнение радиуса области вокруг основных точек со значением ε, что эквивалентно применению DBSCAN с этим конкретным значением в качестве радиуса.

Диаграмма достижимости на рис. 12.17 — это специальная диаграмма, состоящая из двух взаимосвязанных диаграмм. На верхней просто изображен окончательный результат (плоской) кластеризации, получаемый установкой порога равным ε, а нижняя показывает упорядоченную последовательность расстояний достижимости и объясняет, как была получена кластеризация. Кластеры и расстояния достижимости окрашены соответствующими цветами, поэтому хорошо видно, какие точки в верхней половине соответствуют участкам в нижней половине (выбросы отмечены черным цветом; и для удобства, а также для правильного отображения всех значений на диаграмме расстояния достижимости выбросов принимаются равными ε_{MAX} вместо `undefined`). Используя такой подход, мы фактически ослабляем требования к критерию достижимости для этих точек. Но поскольку это максимально возможное значение, которое можно присвоить порогу ε, для последующих шагов ничего не меняется.

Итак, каким образом формируются кластеры, учитывая, что $\varepsilon \leq \varepsilon_{MAX}$? Рисунок 12.18 иллюстрирует идею этого алгоритма (в общих чертах, объединяя его с элементами OPTICS). Сначала просматриваются расстояния достижимости в порядке обработки точек алгоритмом OPTICS, поэтому заранее известно, что расстояние достижимости следующей точки является наименьшим из всех необнаруженных точек (имейте в виду, что сохраненные расстояния достижимости — это расстояния от точки до кластера и именно поэтому порядок играет важную роль!).

Формирование начинается с первой обработанной точки — назовем ее P_1, — и сразу создается новый кластер C_1. Так как P_1 — это первая точка нового кластера, расстояние достижимости для нее не определено, поэтому пока неясно, является ли P_1 выбросом или частью кластера. Ясность наступит только после следующего шага, показанного в части *А* на рис. 12.18, — проверки расстояния достижимости (от C_1) для следующей точки P_2.

Поскольку это значение меньше ε (в примере 0,8), P_2 добавляется в C_1 и берется следующая точка P_3, которая тоже добавляется в C_1, как показано в разделах *Б* и *В*. С визуальной точки зрения на диаграмме достижимости видно, что расстояние достижимости для P_2 ниже пороговой линии (параллельной горизонтальной оси) для $\varepsilon = 0{,}8$.

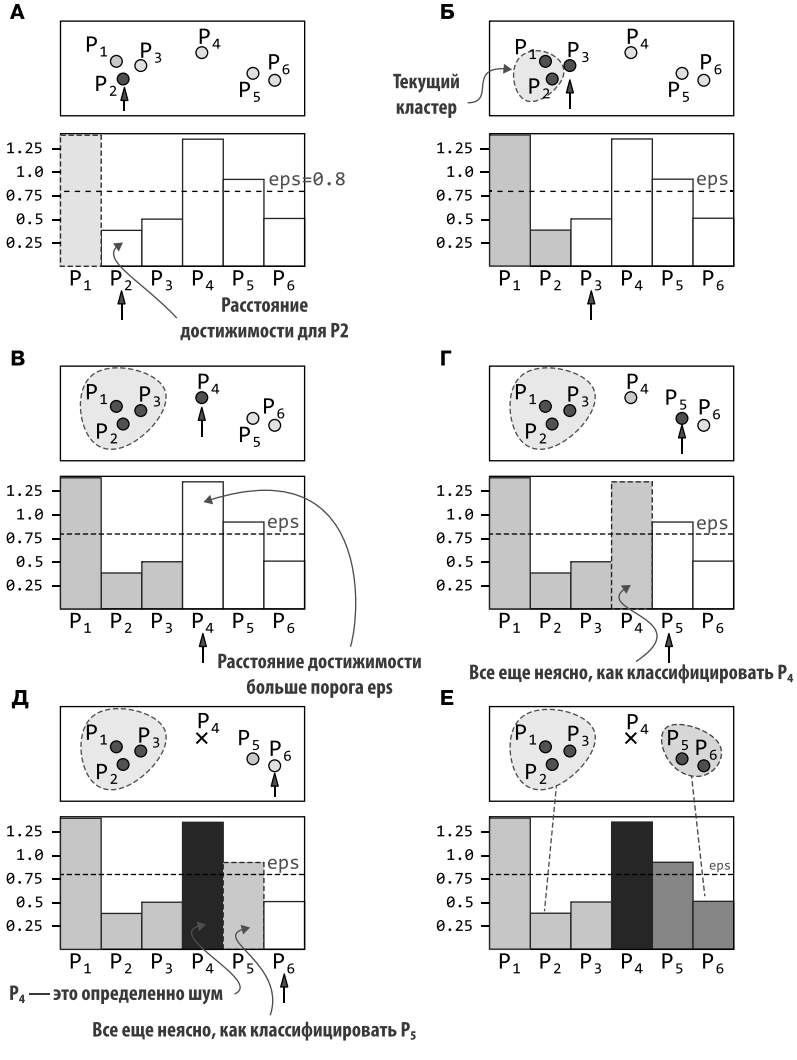


Рис. 12.18. Пример алгоритма плоской кластеризации на основе расстояний достижимости, полученных применением алгоритма OPTICS. Обратите внимание, что расстояния достижимости строятся на диаграмме слева направо, а точки проверяются в порядке увеличения соответствующих им расстояний достижимости, начиная со случайно выбранной записи (этот конкретный порядок обработки точек был выбран в рассматриваемом примере исключительно для удобства)

Когда приходит черед точки P_4, выясняется, что ее расстояние достижимости от C_1 (или технически от любой из точек, обработанных до точки P_4) превышает ε и, следовательно, точка P_4 недостижима из кластера C_1. Поскольку точки обрабатываются в порядке удаленности от текущего кластера, это означает, что для заданного радиуса

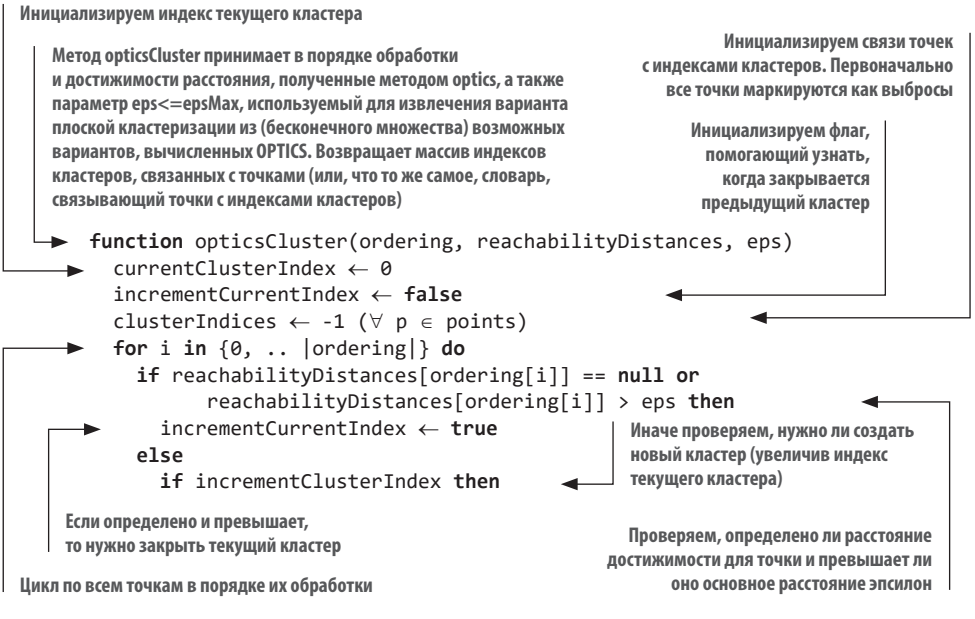
области основной точки, равного ε, не осталось еще необработанных точек, достижимых из C_1. Поэтому текущий кластер «закрывается», и начинается формирование нового.

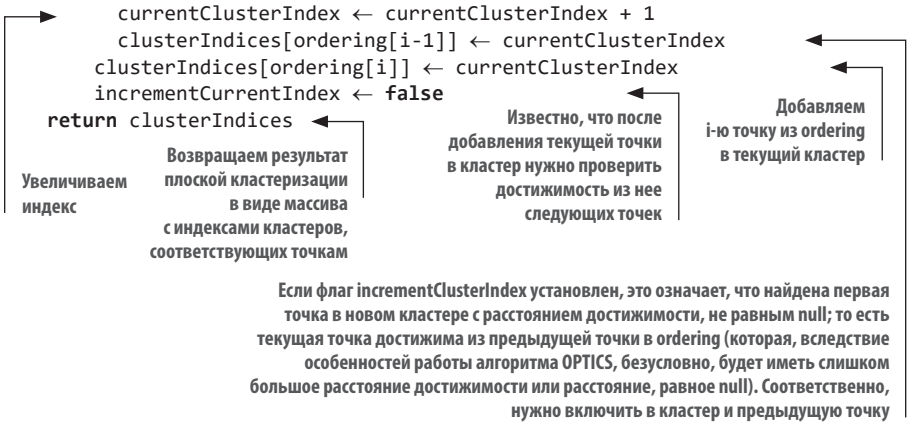
Создается новый кластер для P_4, но пока неясно, как его классифицировать. После проверки расстояния достижимости P_5, (*Г*) обнаруживается, что оно тоже больше ε, а это означает, что P_4 является выбросом (потому что она недостижима ни для одной точки в наборе данных при заданных значениях параметров `minPoints` и ε). Обратите внимание, насколько здесь важен радиус ε. Используя более высокое значение для ε_{MAX}, мы указываем алгоритму OPTICS, что он должен отфильтровывать как шум только точки с расстоянием достижимости больше ε_{MAX}, при этом фактическое решение откладывается до этого второго шага. Практически это позволяет вычислять расстояния достижимости только один раз и опробовать несколько значений ε (вплоть до ε_{MAX}) на этом втором шаге с минимальными затратами.

Процесс повторяется еще раз *(Д)* для P_6, и, поскольку расстояние достижимости (от P_5) меньше ε, следует вывод, что P_6 достижима из P_5 и обе точки можно добавить в один кластер *(Е)*. Если бы точек для обработки было больше, процесс продолжился бы в том же духе.

И последнее замечание перед переходом к реализации в листинге 12.9. Обратите внимание, что если запустить OPTICS дважды, начиная с одной и той же точки, то набор данных будет обрабатываться в том же порядке и результаты будут идентичными (за исключением возможных случаев совпадения расстояний достижимости). Поэтому алгоритм OPTICS может считаться полностью детерминированным.

Листинг 12.9. Метод opticsCluster алгоритма OPTICS





Одно важное замечание, касающееся метода `opticsCluster`: в оригинальной статье авторы предложили немного иной метод, использующий основное расстояние точек при принятии решения о создании нового кластера. Представленная здесь версия основана на соображении, что если достижимость точки q не определена или выше порога ε, а точка p, следующая за q в порядке обработки, имеет расстояние достижимости ниже ε, то q должна быть основной точкой. Расстояние достижимости p от множества точек, обработанных до q, на самом деле тоже должно быть неопределенным или больше, чем у q; в противном случае p была бы обработана до q. Следовательно, расстояние достижимости p на диаграмме — это расстояние достижимости между p и q, и, согласно подразделу 12.4.1, оно определено, только если q является основной точкой.

12.4.4. Иерархическая кластеризация

Теперь, узнав, как выполнить плоскую кластеризацию по результатам OPTICS, вы можете уверенно использовать этот алгоритм. Однако выше упоминалось, что OPTICS — это алгоритм иерархической кластеризации. Что это значит и как это работает?

Алгоритмы иерархической кластеризации производят многоуровневое разделение набора данных. Результаты иерархической кластеризации часто представляются в виде дендрограмм — древовидной структуры, содержащей стратифицированную структуру. Мне нравится думать об изучении дендрограмм как о расщеплении: можно взять участок дендрограммы и посмотреть, как выглядит плоская кластеризация, связанная с этим участком.

Но хватит абстрактных аналогий! Рассмотрим практический пример, чтобы выяснить, как это работает!

На рис. 12.19 показан график достижимости, полученный при тех же расстояниях достижимости и порядке обработки, что и на рис. 12.17, но с другим значением ε.

При использовании большего радиуса очевидно, что области достижимости вокруг точек будут больше и в результате сформируется меньше кластеров.

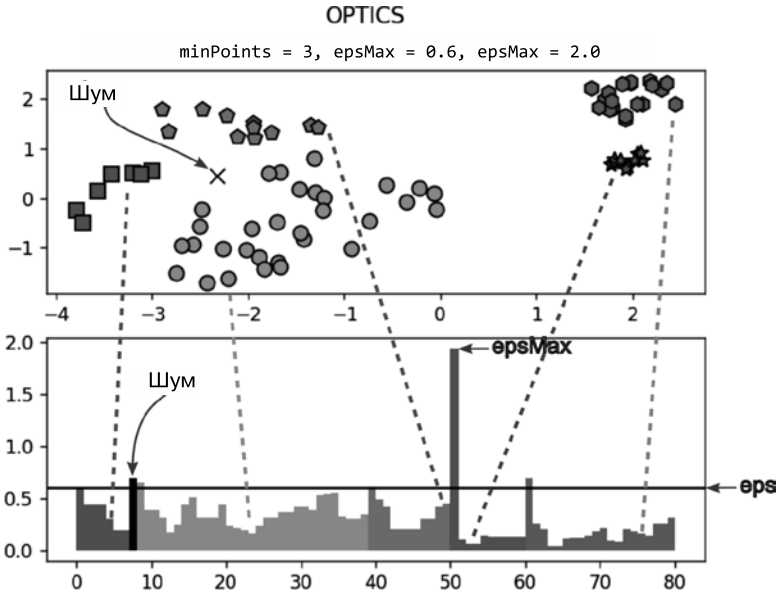


Рис. 12.19. График достижимости, аналогичный изображенному на рис. 12.17, но с другим значением ε. При большем его значении формируется меньше кластеров

Имеет ли это разделение на кластеры больше смысла, чем показанное на рис. 12.17? Трудно сказать. Чтобы ответить на этот вопрос, вероятно, потребуются некоторые знания предметной области или инструменты, которые будут представлены в следующем разделе. Но, поскольку теперь есть только один выброс, окруженный тремя кластерами, можно подумать, что существует определенный смысл в том, что все точки в левой половине образуют один кластер. Чтобы получить такое разделение, можно попробовать большее значение ε, как показано на рис. 12.20.

При ε == 0,7 цель достигнута, но есть загвоздка: два кластера справа тоже сливаются в один! Это произошло потому, что два пика на графике расстояния достижимости, выделенные на рис. 12.20 и отмечающие ребра кластеров C_1 и C_3 на том же рисунке, имеют значения меньше 0,7. Получается, эти два небольших кластера достижимы из самых больших кластеров, расположенных рядом.

Можно ли как-то отделить C_3 от C_4, но при этом объединить C_1 и C_2? Для DBSCAN, как было показано на рис. 12.14, это неразрешимая задача. Для OPTICS плоская кластеризация, разделяющая одну пару кластеров, но не другую, была бы по силам, но только если бы значение ε было меньше порога C_3 и больше C_1. Однако, как показано на рис. 12.21, это не так.

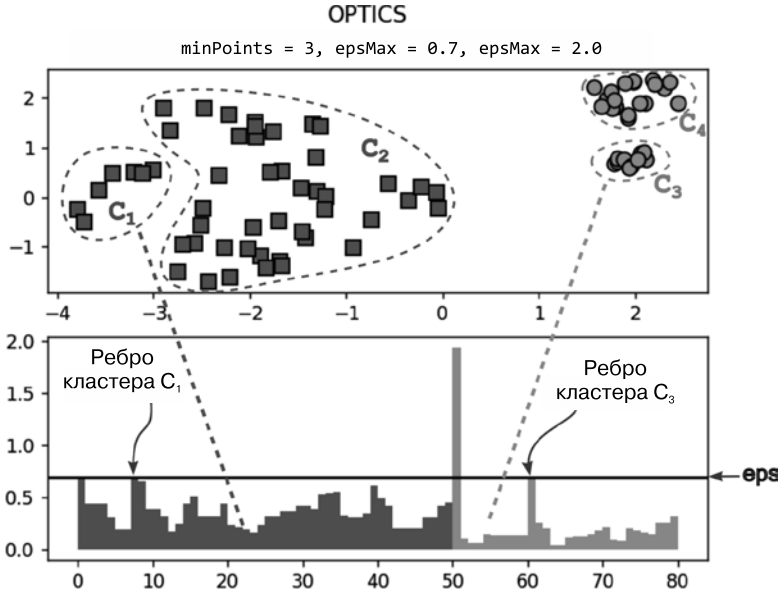


Рис. 12.20. Другой график достижимости, основанный на тех же результатах, полученных алгоритмом OPTICS, но с бóльшим значением $\varepsilon == 0{,}7$. К сожалению, это значение слишком велико и кластеры C_3 и C_4, которые желательно сохранить отдельными, объединяются

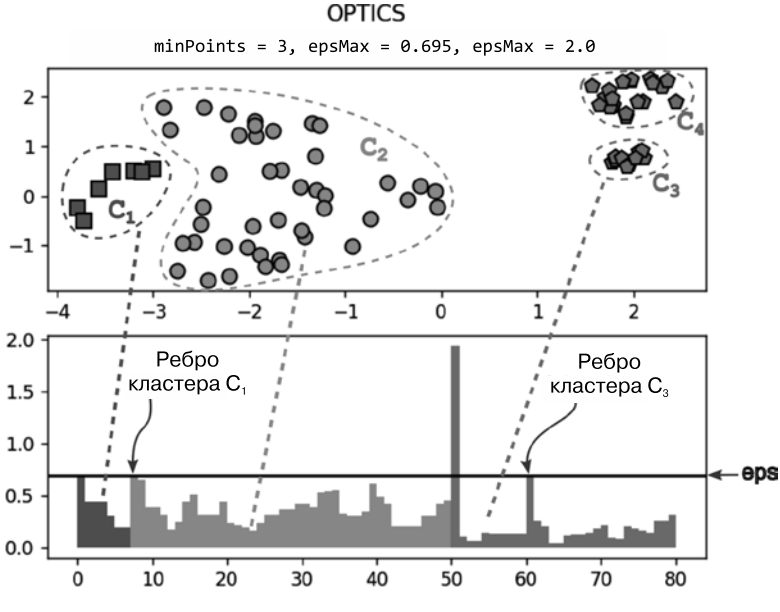


Рис. 12.21. Еще один график достижимости с $\varepsilon == 0{,}695$. Это значение меньше расстояния достижимости между кластерами C_1 и C_2, но недостаточно мало, чтобы предотвратить слияние кластеров C_3 и C_4

Кажется, мы вернулись к старой проблеме с DBSCAN: невозможно найти единственное значение ε, одинаково хорошее для всего набора данных, так как есть одна его «половина» (где $x < 0$), которая имеет заметно более низкую плотность, чем в остальной части набора данных. (Это можно также видеть по расстоянию достижимости: область, изначально выделенная зеленым цветом справа (начиная с индекса 50 на нижнем графике), имеет значительно меньшее среднее значение расстояния достижимости.) В таких случаях особую ценность приобретает иерархическая кластеризация: запуская основной алгоритм OPTICS, мы создаем график упорядочения и достижимости. Они не гарантируют немедленную кластеризацию, но предлагают (или, скорее, подразумевают) набор плоских кластеров, которые можно получить, «разрезав» график достижимости конкретным значением ε, взятым из $[0, \varepsilon_{MAX}]$.

Попробовав все возможные значения эпсилона и отследив получающуюся картину, можно проанализировать, как развивается разделение. Лучший способ выполнить такой анализ — использовать дендрограмму, древовидную структуру, показанную на рис. 12.22, где — для ясности — ось X показывает только восемь кластеров (плюс немного шума) в качестве записей самого низкого уровня на рис. 12.17, тогда как обычно для каждой точки в наборе данных имеется одна запись. Обратите внимание, как кластеры и точки шума упорядочены по оси X дендрограммы: все точки должны следовать в том же порядке, что и на графике достижимости (который, в свою очередь, соответствует порядку обработки точек алгоритмом OPTICS).

Теперь, глядя на дендрограмму, можно понять, почему этот алгоритм называется иерархической кластеризацией: он отслеживает иерархию кластеров, идущих (сверху вниз) от одного кластера, содержащего весь набор данных, к N одиночным элементам, которые являются протокластерами с единственной точкой данных в каждом. Движение от верха дендрограммы к ее низу соответствует виртуальному изучению всех возможных значений ε и плоских группировок, связанных с этими значениями: когда на рис. 12.17, 12.19 и т. д. выбирались ε = 0,5 или ε = 0,6. Мы, образно говоря, вырезали участок дендрограммы и брали пик на кластерах, образовавшихся на этом уровне. Однако, как уже было показано, разрезание такого участка линией, перпендикулярной оси ε (имеется в виду линия, где ε постоянна для всего набора данных), не работает для неоднородного набора данных.

Но самое замечательное в иерархической кластеризации заключается в том, что нет необходимости обрезать эту дендрограмму на одной высоте для всего набора данных! Другими словами, в разных ветвях дендрограммы можно использовать разные значения ε. А как решить, какие значения выбрать для тех или иных ветвей? Работая с двумерными наборами данных, можно руководствоваться интуицией, но при работе с многомерными данными решение должно основываться на знании предметной области или определяться с помощью какой-либо метрики. В нашем примере, к слову, можно рассмотреть разбиение после первого разбиения на дендрограмме, C_E и C_F, и сравнить их среднюю плотность. Поскольку плотность в двух ветвях явно различается, можно попробовать получить значения ε для каждой из них отдельно,

основываясь на статистике каждого подмножества: выше, где плотность ниже (или, что то же самое, где среднее расстояние достижимости выше).

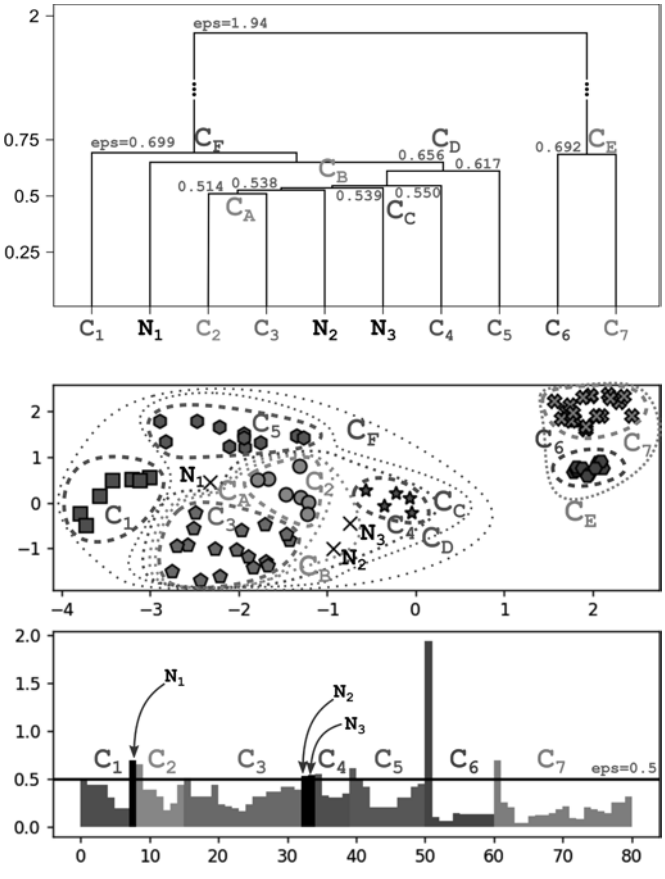


Рис. 12.22. Дендрограмма, построенная по результатам, полученным с помощью OPTICS для нашего набора данных с параметрами $\varepsilon_{MAX} = 2$ и minPoints = 3. Для ясности здесь опущена нижняя часть дендрограммы для $\varepsilon < 0{,}5$. Кластеры, сформированные для $\varepsilon == 0{,}5$, получили названия от C_1 до C_7 и считаются основными единицами на этом графике (обычно мы начинаем с отдельных точек). Сверхскопления, полученные их слиянием, получили названия от C_A до C_F

Если результат получится неудовлетворительным, можно продолжить обход каждой ветви дендрограммы, повторяя этот шаг, пока не выполнится одно из следующих условий:

- достигнута точка, когда обе ветви имеют схожие характеристики;
- получено желаемое количество кластеров (если имеется представление о предметной области);

- получено удовлетворительное значение некоторой определенной нами метрики (см. следующий раздел);
- или достигнут порог максимальной глубины обхода дерева.

На рис. 12.23 показано, как это выглядит для рассматриваемого примера, если предположить, что нам достаточно достигнуть первого разделения. Как видите, теперь вместо сегмента у нас есть ступенчатая функция, пересекающая график достижимости и дендрограмму. На рис. 12.23 сохранены только три кластера, C_F, C_6 и C_7, но оставлены видимыми границы всех подкластеров C_F.

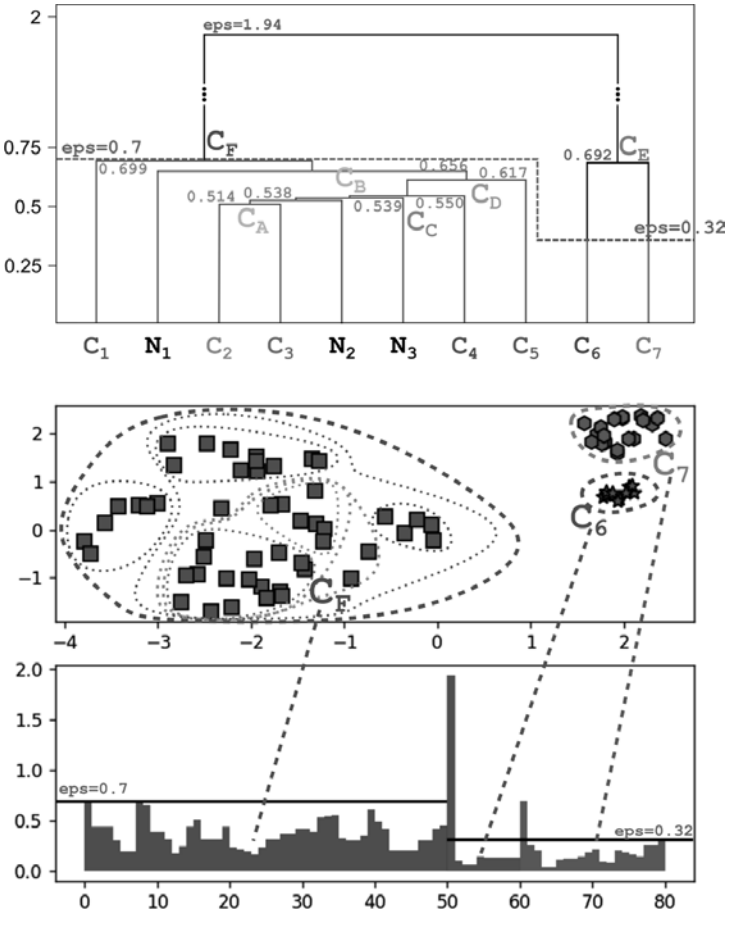


Рис. 12.23. Дендрограмма, построенная на основе результатов, полученных с помощью OPTICS для нашего примера данных с параметрами $\varepsilon_{MAX} = 2{,}0$ и minPoints=3. Для ясности здесь опущена нижняя часть дендрограммы для $\varepsilon < 0{,}5$. В двух ветвях дендрограммы заданы два разных порога

12.4.5. Анализ производительности и заключительные замечания

Иерархическая кластеризация не только мощный, но и ресурсоемкий инструмент по сравнению с плоской кластеризацией. В оригинальной статье предполагается, что основной алгоритм OPTICS работает примерно в 1,6 раза медленнее, чем DBSCAN (на тех же наборах данных), при этом сохранение иерархической кластеризации, а также построение и исследование дендрограммы, очевидно, тоже требуют дополнительной памяти и вычислений.

Что касается базового алгоритма, беглого взгляда на листинги 12.6 и 12.7 достаточно, чтобы заметить, что код обрабатывает каждую точку ровно один раз (каждая точка ровно один раз добавляется в очередь с приоритетами и удаляется из нее) и для каждой обработанной точки выполняет один поиск по диапазону. Однако записи в очереди приоритетов могут обновляться несколько раз, возможно, при обработке каждой точки. Размер очереди с приоритетами зависит от размера плотных регионов, и чем больше ε_{MAX}, тем больше точек будет в очереди. Потенциально очередь может содержать все точки, начиная с итерации 1 (если ε_{MAX} кратно максимальному из попарных расстояний), и все точки в очереди могли бы обновляться при обработке каждой новой точки. Точно так же время, необходимое для поиска ближайших соседей, даже при использовании наихудших ограниченных структур, таких как k-мерные деревья, зависит от ε_{MAX}, и если радиус области поиска достаточно велик, то поиск превращается в линейное сканирование всего набора данных.

По этим причинам время работы алгоритма в худшем случае является квадратичным! Тем не менее можно показать, что при соответствующем выборе ε_{MAX} поиск ближайшего соседа может быть выполнен за амортизированное логарифмическое время и аналогичным образом можно ограничить размер приоритетной очереди[1]. Следовательно, при разумном выборе ε_{MAX} среднее время выполнения может составлять всего $O(n \times \log(n))$ для набора данных с n точками.

Оригинальная статья содержит более формальное описание алгоритма и лежащей в его основе теории, а также автоматизированную процедуру построения иерархической структуры кластеризации на основе графика достижимости. Это хорошая отправная точка для желающих углубить свое понимание алгоритма.

Также можно порекомендовать довольно интересную статью об усовершенствованном алгоритме *DeLiClu*[2], который расширяет OPTICS идеями кластеризации с одной связью, что позволяет вообще избежать параметра ε_{MAX} и в то же время оптимизировать алгоритм и улучшить время его работы.

---

[1] Кроме того, при использовании кучи Фибоначчи метод «обновления приоритета» будет выполняться за амортизированное время $O(1)$, так что даже за линейно-логарифмическое число вызовов общее время работы составит не более $O(n \times \log(n))$.

[2] *Achtert E., Böhm C., Kröger P.* DeLi-Clu: boosting robustness, completeness, usability, and efficiency of hierarchical clustering by a closest pair ranking // Pacific-Asia Conference on Knowledge Discovery and Data Mining. — Berlin; Heidelberg: Springer, 2006.

12.5. ОЦЕНКА РЕЗУЛЬТАТОВ КЛАСТЕРИЗАЦИИ: МЕТРИКИ ОЦЕНКИ

К этому моменту вы познакомились с тремя разными (в порядке возрастания сложности и эффективности) алгоритмами кластеризации, их сильными и слабыми сторонами. Несмотря на существенные различия, все эти алгоритмы имеют одну общую черту: они требуют, чтобы человек установил один или несколько гиперпараметров для достижения наилучшего результата.

Настройка этих параметров вручную может оказаться слишком сложной задачей: если при работе с наборами двумерных данных еще можно посмотреть на результат кластеризации и решить, насколько хорошо он выглядит, то с наборами многомерных данных мы лишены такой роскоши, как использование интуиции. А значит, пришло время определить более формальный способ оценки качества кластеризации и поговорить о метриках оценки.

Для обсуждения этой темы вернемся к первоначальному примеру: сегментация клиентов для сайта электронной коммерции на основе двух характеристик, а именно годового дохода и средних месячных трат в вашем магазине. На рис. 12.24 показан синтетический набор данных с реалистичным распределением (полученным на основе данных с реального сайта). Набор данных предварительно обработан, и точки нормализованы. Предварительная обработка данных — стандартный шаг в науке о данных. Он помогает убедиться, что функции с большими значениями имеют одинаковый вес, учитываемый при окончательном решении. Так, в нашем примере можно обоснованно предположить, что годовая зарплата находится в диапазоне 100–200 тысяч долларов, а среднемесячные траты на сайте составляют где-то около 500–1000 долларов. Если бы это было так, то один признак имел бы значения на три порядка больше, чем другой, и набор данных был бы полностью искажен. Другими словами, если бы мы использовали евклидово расстояние для измерения близости двух точек, разница в годовой заработной плате внесла бы гораздо больший вклад в окончательное расстояние между двумя клиентами, чем разница в ежемесячных расходах, что сделало бы второй признак неактуальным.

Чтобы исключить этот эффект, выполним этап нормализации, вычитая среднее значение каждого признака из каждой отдельной точки, а затем разделим на стандартное отклонение признака[1].

Справа на рис. 12.24 показана возможная кластеризация набора данных. Не нужно быть экспертом в предметной области, чтобы понять, что количество кластеров выбрано неудачно.

---

[1] Это называется нормализацией *z*-показателя и характеризуется тем, что дает данные с нулевым средним и единичной дисперсией. Существуют и другие способы масштабирования признаков; чтобы узнать больше, можно, например, прочитать главу 2 из книги Machine Learning in Action Питера Харрингтона (Peter Harrington), выпущенной издательством Manning Publications в 2012 году.

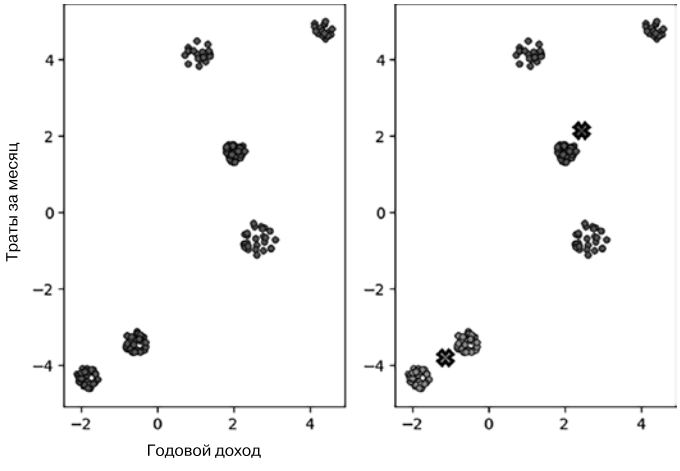


Рис. 12.24. Синтетический набор данных, показывающий клиентов гипотетического веб-сайта электронной коммерции после масштабирования признаков. Справа возможный (не самый лучший) вариант кластеризации набора данных

Можно ли написать метод или математическую формулу, чтобы выразить (плохое) качество этого выбора?

Начнем с вопроса: какие последствия может иметь плохая кластеризация? На рис. 12.24 видно, что точки, которые должны принадлежать разным кластерам (например, четырем кластерам в правом верхнем квадранте), оказались сгруппированы вместе, то есть точки, находящиеся далеко друг от друга, очутились в одном кластере. Вспомним определение метода k-средних: его целью является минимизация квадрата евклидова расстояния от точек до соответствующих им центроидов. Когда создается слишком мало центроидов (как здесь), это вынуждает связывать много точек с одним центроидом, даже если он находится далеко. Следовательно, среднее расстояние между точками в каждом кластере и их центроидом будет больше, чем если бы было выбрано правильное количество центроидов!

Если запустить алгоритм несколько раз с разными количествами центроидов, то можно вычислить среднее (или медианное) расстояние от точек до их центроидов и выбрать наименьшее значение. Это достаточно хорошо?

Хорошо, но не совсем. Если подумать, то, каким бы ни было значение k (количество центроидов), проверенное последним, если выбрать $k + 1$, с еще одним центроидом, среднее расстояние уменьшится еще немного, потому что каждый центроид будет иметь меньше точек в своем кластере и они будут располагаться ближе к нему. Если довести дело до крайности, выбрав $k = n$ (каждой точке в наборе данных соответствует свой центроид), то мы получим среднее расстояние до центроида, равное нулю.

Существует ли какое-то сбалансированное решение? Правда в том, что нет простого способа сбалансировать его, но есть один эмпирический метод, помогающий выбрать наилучшее значение. Вы познакомитесь с ним чуть ниже.

А прежде проясним еще один важный момент: расстояние от точек до центроидов — это только одна из многих возможных метрик. Кстати, этой метрикой можно оценивать только алгоритмы кластеризации, основанные на центроидах, она неприменима к OPTICS или DBSCAN. Для этих двух алгоритмов можно использовать похожую метрику — внутрикластерное расстояние: среднее попарное расстояние между точками в одном кластере. Другой интересной метрикой, которая может пригодиться при работе с категориальными признаками, является общая сплоченность (она подобна расстоянию до центра кластера, но вместо евклидова использует косинусное расстояние).

В оставшейся части этого раздела в качестве метрики будет принято среднее внутрикластерное расстояние. А теперь пришло время раскрыть *метод локтя* (elbow method), эмпирический инструмент, используемый в кластеризации и машинном обучении для определения наилучших значений гиперпараметров.

На рис. 12.25 показан пример применения метода локтя к набору данных с информацией о наших клиентах для определения оптимального количества кластеров на основе среднего внутрикластерного расстояния. Но его также можно было бы использовать для выбора наилучшего значения ε или `minPoints`, если бы мы использовали алгоритм DBSCAN.

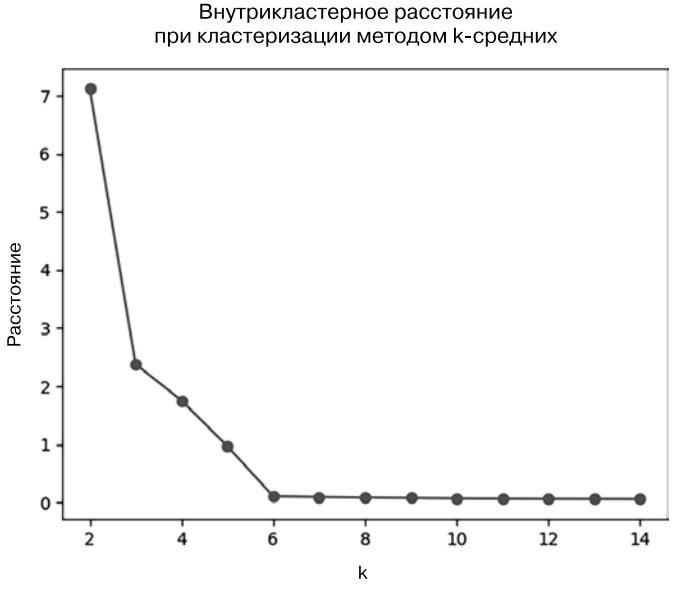


Рис. 12.25. Использование метода локтя в определении наилучшего значения количества кластеров для разделения набора данных на рис. 12.24

Глядя на рис. 12.25, нетрудно догадаться, откуда взялось такое название метода: график выглядит как согнутая рука, и оптимальным считается значение, соответствующее точке сгиба, где рост функции резко меняется.

Для нашего примера это значение равно $k = 6$. После него дальнейшее увеличение числа кластеров улучшает метрику очень незначительно. Для реальных наборов данных «сгиб» может быть менее выраженным (в нашем случае кластеры очень компактны и находятся далеко друг от друга, поэтому по достижении оптимального количества кластеров улучшение от добавления еще одного центроида почти равно нулю), но чаще точка похожа на водораздел: слева от нее наклон кривой ближе (или больше) к $-45°$, справа — ближе к $0°$.

Есть, конечно, несколько деталей, которые необходимо учесть для успешного применения этого метода. Поскольку метод k-средних является рандомизированным, важно запустить его несколько раз для каждого значения k. Затем можно выбрать наилучшее значение из всех прогонов (а также сохранить полученное разделение на кластеры, как это делается в блокноте Jupyter из репозитория книги) или среднее (медианное) значение метрики в зависимости от цели[1]. Более того, желательно подобрать лучшую метрику для вашей задачи. Можно, например, минимизировать расстояние между точками внутри одного кластера, чтобы кластеры были как можно более однородными, или максимизировать расстояние между разными кластерами, чтобы обеспечить четкое разделение между разными группами.

В листинге 12.10 показаны шаги, которые необходимо выполнить для успешного применения метода локтя (вплоть до построения графика, не включенного по понятным причинам). Как уже упоминалось, проверить на практике этот метод можно с помощью блокнота Jupyter в репозитории книги.

Листинг 12.10. Метод локтя

Метод kmeansElbow принимает список точек, список значений для проверки k (ksToTest), количество попыток кластеризации (maxIter) и максимальное количество итераций метода k-средних (runsPerK) для каждого проверяемого значения k. Возвращает ассоциативный массив с лучшим значением метрики внутрикластерного расстояния для каждого количества проверенных центроидов

Инициализируем массив для запоминания результатов (потому что алгоритм запускается несколько раз для каждого k)

```
function kmeansElbow(points, ksToTest, maxIter, runsPerK)
  results ← {}
  for k in ksToTest do
    M ← []
    for i in {1,..,runsPerK} do
      (centroids, clusters) ← kmeans(points, k, maxIter)
      M[i] ← intraClusterDistance(points, centroids, clusters)
    results[k] ← min(M)
  return results
```

Инициализируем ассоциативный массив с результатами

Повторяем runsPerK раз

Применяем метод k-средних к текущей комбинации исходных данных и значений гиперпараметров

Выбираем лучший результат для текущего количества центроидов

Вычисляем метрику внутрикластерного расстояния по результатам работы алгоритма

Цикл по всем проверяемым значениям k

Возвращаем результаты (при желании можно сохранить разделение на кластеры, соответствующее лучшему значению метрики для каждого значения k, а также вернуть список разделений)

[1] Если оптимизируется выбор для текущего набора данных, то предпочтительнее выбрать лучший результат. Если предполагается запускать алгоритм на нескольких наборах данных с похожими характеристиками, то предпочтительнее найти хорошо обобщающее значение, и лучшим выбором может стать медиана.

12.5.1. Интерпретация результатов

Для проверки работы метода локтя мы сохранили результаты с наилучшими значениями метрик для каждого протестированного значения k (значения показаны на рис. 12.25) и теперь можем рассмотреть некоторые из них (рис. 12.26), чтобы убедиться в правильности выбора. И действительно, при $k = 5$ слишком мало центроидов и два кластера были связаны с одним и тем же центроидом (слева внизу на левой диаграмме), а при $k = 7$ один «естественный» кластер разделяется на два (на диаграмме справа).

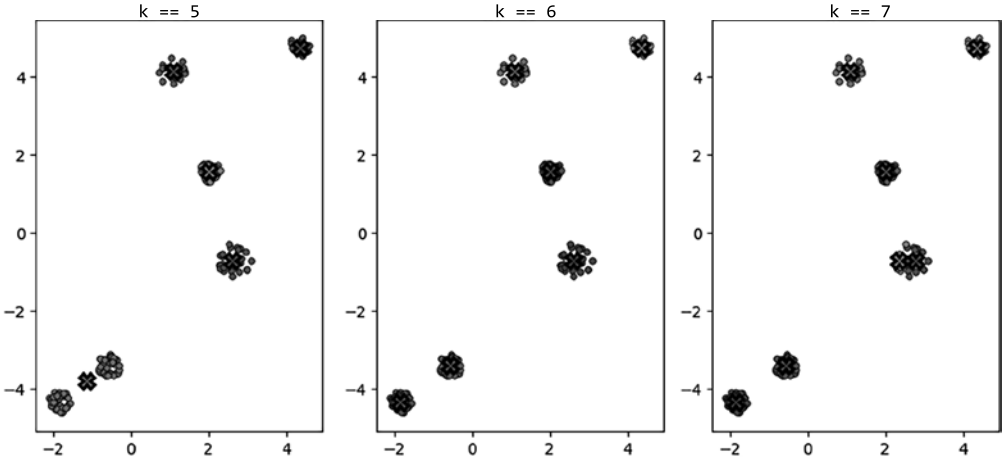


Рис. 12.26. Наилучшие варианты кластеризации рассматриваемого набора данных методом k-средних получаются при k, равном 5, 6 и 7. Обратите внимание, что 6 действительно является идеальным значением и наименьшим, которое дает достаточно центроидов для естественных кластеров в наборе данных

Выяснив, что шесть кластеров — лучшее решение для нашего набора данных, можно попробовать интерпретировать результат кластеризации. Например, вверху справа сосредоточены клиенты с высоким годовым доходом, которые щедро тратят деньги на веб-сайте. Чуть левее есть еще одна интересная группа: люди, которые, несмотря на более низкий доход, тратят на покупки почти столько же, сколько и люди с высоким доходом. В левом нижнем углу находятся две группы людей с низкими доходами, которые тратят мало денег на электронную коммерцию. В отношении этих двух групп можно проводить одинаковую маркетинговую политику или провести дополнительный анализ, чтобы понять различия и выделить категории товаров, наиболее подходящие для каждой из них. В любом случае, учитывая, что они приносят ограниченный доход, отдел маркетинга скорее предпочел бы попросить команду специалистов по анализу данных сосредоточиться на двух кластерах в центре диаграммы: на клиентах со средним доходом, которых можно было бы поощрить с помощью целевых рекламных кампаний и даже попросить пройти опрос, результаты которого помогли бы отделу маркетинга повысить удовлетворенность этих клиентов.

Мы рассмотрели только один пример того, как кластеризация может помочь процветанию компании, но в реальной жизни таких примеров гораздо больше. Теперь ваша очередь применить эти мощные методы к своим данным!

РЕЗЮМЕ

- Кластеризация — основной метод обучения без учителя, используемый для осмысления немаркированных данных и выявления закономерностей в исходных данных.
- Кластеризация может применяться, например, для маркетинговой сегментации, обнаружения шума (скажем, в сигналах) и предварительной обработки данных.
- Самый старый алгоритм кластеризации — метод k-средних. Он самый простой в реализации, но накладывает ограничения на форму кластеров. Он может обнаруживать только выпуклые кластеры и не способен выявлять нелинейно разделимые кластеры.
- Другой подход к кластеризации — алгоритм DBSCAN. Он основан на идентификации групп по плотности точек и может выявлять кластеры любой формы, в том числе и нелинейно разделимые, но плохо работает с наборами данных, имеющими области с разной плотностью. Кроме того, выбор лучших значений гиперпараметров алгоритма — трудная и весьма нетривиальная задача.
- OPTICS — более новый метод, основанный на DBSCAN. Он создает иерархическую кластеризацию, позволяет обрабатывать наборы данных, имеющие области с разной плотностью и упрощает выбор значений параметров.
- Чтобы оценить качество кластеризации набора данных, можно использовать оценочные метрики, такие как внутрикластерное расстояние, межкластерное расстояние или общая связность.
- Метод локтя — это инструмент, обеспечивающий графическую обратную связь для выбора наилучших значений гиперпараметров алгоритмов.

13 Параллельная кластеризация: MapReduce и кластеризация методом купола

В этой главе

- ✓ Особенности параллельных и распределенных вычислений.
- ✓ Кластеризация методом купола.
- ✓ Распараллеливание метода *k*-средних применением метода купола.
- ✓ Вычислительная модель MapReduce.
- ✓ Создание распределенной версии метода *k*-средних с использованием MapReduce.
- ✓ Кластеризация методом купола с применением модели MapReduce.
- ✓ MR-DBSCAN.

В предыдущей главе вы познакомились с кластеризацией и тремя различными подходами к разделению данных: методом *k*-средних, DBSCAN и OPTICS.

Эти алгоритмы используют однопоточный подход, в котором все операции выполняются последовательно в одном потоке управления[1]. Как раз сейчас стоит задуматься над одним вопросом о подходе к проектированию: действительно ли необходимо выполнять эти алгоритмы последовательно?

---

[1] Многопроцессорные машины могут оптимизировать выполнение ряда операций, распределяя их между несколькими ядрами. Однако этот уровень распараллеливания ограничен количеством ядер на кристалле, в настоящее время для самых мощных серверов их количество не превышает сотни.

В этой главе вы найдете ответ и познакомитесь с альтернативными шаблонами проектирования и примерами, которые помогут выявить возможности параллельного выполнения кода и затем использовать передовой опыт для значительного его ускорения.

Изучив материал, поймете разницу между параллельными и распределенными вычислениями, откроете для себя кластеризацию методом купола (canopy clustering), познакомитесь с MapReduce, вычислительной моделью распределенных вычислений, и, наконец, сможете переписать реализацию алгоритмов кластеризации, которые вы анализировали в предыдущей главе, для выполнения в распределенной среде.

13.1. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Модель ОЗУ (представленная в приложении Б) традиционно является однопоточной, и анализ алгоритмов обычно фокусируется на последовательном выполнении и улучшении времени работы однопоточных приложений. Тем не менее распараллеливание, когда оно применимо, может обеспечить огромное ускорение, и поэтому владеть этим приемом должен каждый инженер-программист.

> **ТЕМА МНОГОПОТОЧНОСТИ НА СОБЕСЕДОВАНИЯХ С ПРОГРАММИСТАМИ**
>
> Когда на собеседовании с претендентом на должность программиста дело доходит до анализа алгоритмов, иногда обнаруживается, что по этому вопросу существуют самые разные мнения. Вот почти анекдотическая ситуация из моего опыта: проходя собеседования, я встретил двух специалистов, проводящих их, которые занимали полярно противоположные позиции: один считал распараллеливание «обманом», непригодным для решения обсуждаемой задачи, а другой ожидал, что претендент предложит распараллеливание как возможное ее решение. Имейте это в виду, когда в следующий раз пойдете на интервью. Конечно, многое зависит от конкретной задачи и от того, куда ведущий собеседование стремится вас направить, но часто полезно спросить его о возможности многопоточного и параллельного выполнения (при условии, что вы знаете, о чем говорите!).

Чтобы дать вам представление о величине ускорения, о котором мы говорим, могу сказать, что лично видел, как время работы приложения сократилось с 2 ч до менее чем 5 мин благодаря использованию Kubernetes и Airflow для распределения нагрузки и параллельной обработки данных небольшими фрагментами. Конечно, деление данных и обработка каждого фрагмента по отдельности не всегда возможны; это зависит от предметной области и от алгоритма.

А теперь настал тот момент, когда мы можем спросить себя, является ли кластеризация той областью, где можно распараллелить выполнение и получить выгоды? Можно ли разбить наборы данных и применить алгоритмы, обсуждавшиеся в главе 12, к каждой получившейся части независимо?

13.1.1. Параллельные и распределенные вычисления

Прежде чем перейти к сути, сразу оговоримся: обычно термин *«параллельные вычисления»* относится только к вычислениям, которые выполняются на нескольких процессорах одной и той же системы, то есть к многопоточности. Когда речь идет об использовании нескольких процессоров на нескольких машинах, взаимодействующих по сети, мы имеем в виду то, что называется *распределенными вычислениями*. На рис. 13.1 показана диаграмма, иллюстрирующая это различие.

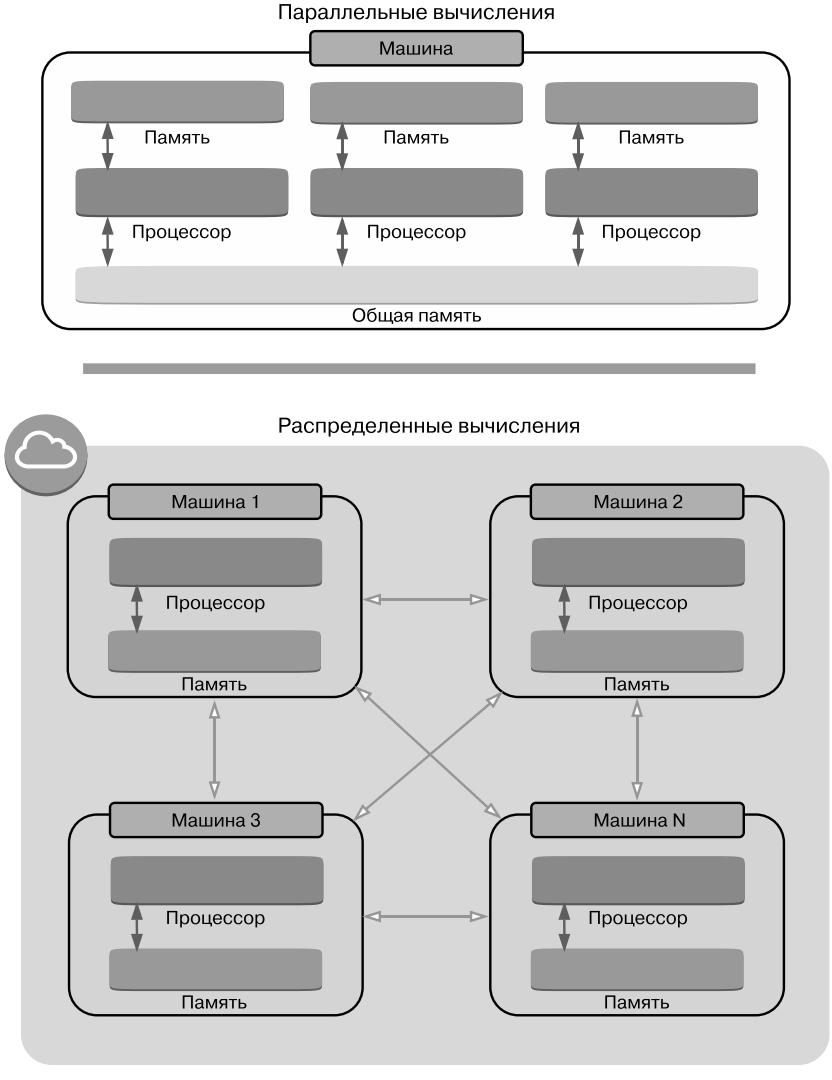


Рис. 13.1. Схематическое изображение различий между моделями параллельных и распределенных вычислений

Параллельные вычисления ограничены количеством процессоров, доступных на одном компьютере, тогда как распределенные вычисления лучше подходят для масштабирования систем и обработки огромных наборов данных. С другой стороны, если набор данных умещается в памяти одной машины, то обработка с применением параллельных вычислений выполняется заметно быстрее, потому что процессы могут обмениваться данными через общую память, тогда как узлы в распределенных системах вынуждены обмениваться информацией через сеть (разница в величине задержки[1] — 100 нс против 150 мс[2], то есть в 10^6 раз).

В этой главе я часто буду использовать термин «параллельные вычисления», подразумевая обе модели, которые, по сути, представляют собой программные абстракции, способные выполняться и в нескольких потоках на одной машине, и в распределенной системе. Единственным отличительным признаком будет объем входных данных и необходимых ресурсов, а не используемые алгоритмы.

13.1.2. Распараллеливание метода k-средних

Теперь перейдем к конкретике и попробуем ответить на вопрос о возможности распараллелить алгоритм *k*-средних.

Обзор каждого шага алгоритма по отдельности поможет нам применить к задаче принцип «разделяй и властвуй». Конкретное описание алгоритма кластеризации методом *k*-средних и его реализацию вы найдете в разделе 12.2.

Первым шагом является *инициализация*, выбор начальных предполагаемых центроидов. Если центроиды выбираются полностью случайно, то этот шаг не зависит от набора данных, а время его выполнения пропорционально только количеству k кластеров. Поэтому нет смысла распараллеливать полностью рандомизированную версию. Распараллеливание может пригодиться, когда точки извлекаются из распределения независимо. В подразделе 13.3.2 будет показано, как распараллелить этот шаг.

Шаг 2, *повторное центрирование*, вычисляет центр масс для каждого кластера. Этот шаг будет рассмотрен первым, потому что каждый кластер может обрабатываться независимо и для вычисления центра масс кластера нужны только точки, принадлежащие ему. Вне всяких сомнений, этот шаг можно распараллелить, выделив один процесс для каждого кластера. Последовательная версия должна выполнить $n \times d$ сложений и $k \times d$ делений, где n — количество точек в наборе данных, d — размерность (мощность каждой точки), а в параллельной версии каждый процесс, при условии равномерного разделения на кластеры (в лучшем случае, конечно), будет выполнять $d \times n/k$ сложений и d делений. Если все потоки будут завершаться одновременно и выполняться с той же скоростью, что и исходный последовательный алгоритм, получим ускорение в k раз.

---

[1] Для справки: http://norvig.com/21-days.html#answers.

[2] Здесь подразумеваются глобальные вычислительные сети WAN или высокопроизводительные облачные службы. В локальных кластерах в центрах обработки данных при правильной настройке задержку можно снизить на два порядка, до 1 мс.

Шаг 3, *классификация*, его распараллелить сложнее. Теоретически нужно проверить расстояния для всех точек, чтобы назначить их соответствующему центроиду. Однако если хорошенько подумать, то действительно ли нужны все точки? Если обратиться к рис. 12.3, можно заметить, что точка может перейти в другой кластер, только если он является соседним к ее текущему кластеру, и никогда не перейдет в кластер, находящийся дальше. Кроме того, если центроид c' сместится, отдалившись от второго кластера C (при условии, что центроид второго кластера не сдвинется с места), то точка из C не сможет перейти в c'. Однако с такими предположениями нужно соблюдать осторожность, подходить к процессу очень скрупулезно, поэтому распараллелить этот шаг будет несколько сложнее.

Даже просто распараллелив шаг 3 алгоритма k-средних, можно получить хорошее ускорение по сравнению с последовательной версией.

Можно ли добиться большего? Да, можно, по крайней мере, двумя разными способами. Чтобы понять, как это сделать, необходимо сначала познакомиться с новым алгоритмом, а затем с моделью программирования, которая изменит правила игры.

13.1.3. Кластеризация методом купола

Вопрос: как выполнить быструю грубую псевдокластеризацию перед запуском любого реального алгоритма кластеризации, чтобы получить хоть какое-то представление о распределении данных?

Для этой цели обычно используется кластеризация методом купола (canopy clustering). Такой алгоритм группирует точки в сферические области (круги с двумерными данными в нашем примере), подобно методу k-средних, но в отличие от него допускает перекрытие областей, и большинство точек связывается с несколькими областями.

Алгоритм кластеризации методом купола быстрее и проще метода k-средних, потому что выполняется за один проход, не требует вычисления центроидов куполов (сферических псевдокластеров) и не сравнивает каждую точку с каждым центроидом. Вместо этого он выбирает одну точку в наборе данных в качестве центра каждого купола и добавляет в этот купол точки, находящиеся вокруг него.

Алгоритм можно ускорить, если вместо точной метрики расстояния для точек в k-мерном пространстве использовать быструю приближенную метрику. Результат получится менее точным, но затем его можно уточнить, применяя более подходящий алгоритм кластеризации на следующем шаге. Как будет показано в следующем разделе, использование кластеризации методом купола в качестве первого шага перед применением других алгоритмов может ускорить их сходимость и сократить время работы[1].

[1] Этот прием может уменьшить количество необходимых итераций и операций, выполняемых в каждой итерации.

На рис. 13.2 показан пример работы кластеризации методом купола, а в листинге 13.1 представлен псевдокод реализации.

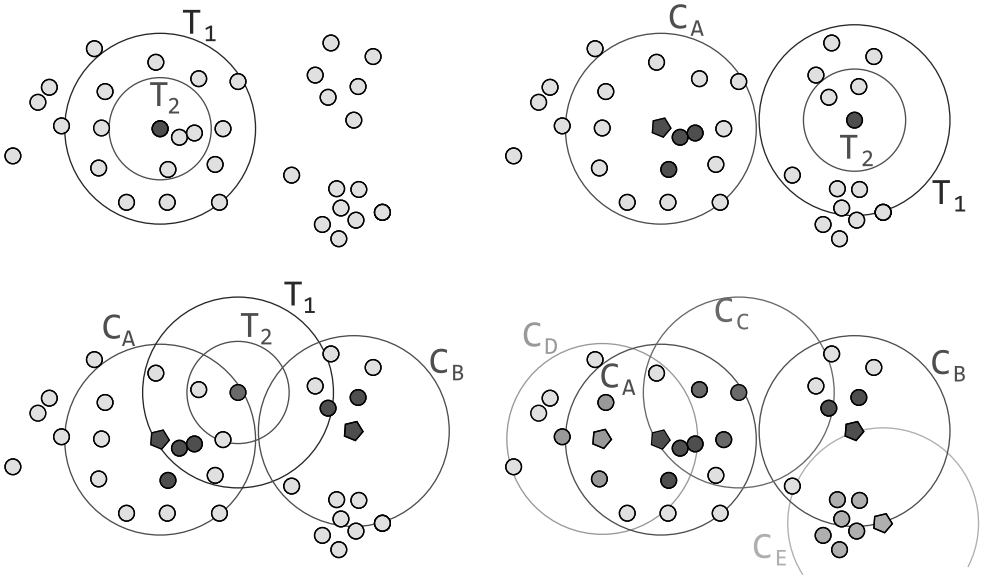


Рис. 13.2. Пример применения кластеризации методом купола к набору данных. Первые три диаграммы показывают создание куполов, начиная со случайно выбранных точек. Кружки, залитые сплошным цветом, отличным от самого светлого оттенка, — это точки, удаленные из списка возможных центроидов, а пятиугольники — это центроиды. Точки, обозначенные сплошной заливкой, удаляются из списка, потому что лежат в пределах внутреннего радиуса (T_2) от центроидов (как показано в первых трех шагах). На последней диаграмме видны кластеры, созданные после еще нескольких шагов. Обратите внимание, что все еще есть несколько светло-серых точек, поэтому будут созданы как минимум еще три купола (частично перекрывающие пять уже образованных) для включения этих точек

Листинг 13.1. Кластеризация методом купола

Если радиус T2 больше или равен радиусу T1, то генерируется ошибка

Инициализируем результат (список куполов) пустым списком

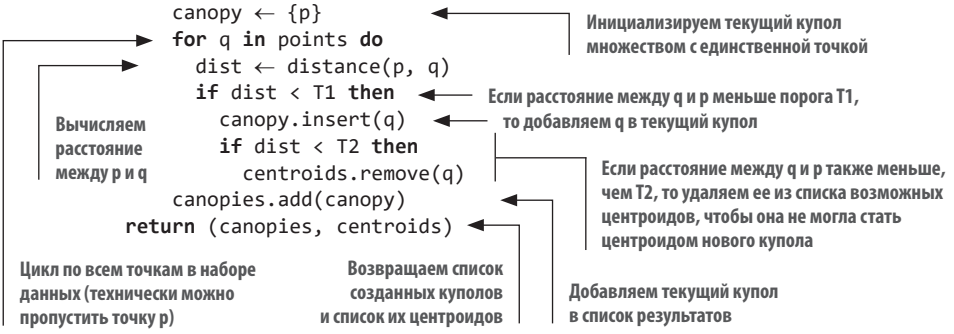
Метод canopyClustering принимает список точек и два порога, T1 и T2, и возвращает список куполов, то есть перекрывающихся наборов точек

Инициализируем множество потенциальных центров куполов по всему набору данных

Пока в списке возможных центроидов куполов остаются точки...

...получаем случайную точку из списка центроидов, исключив ее из этого списка

```
function canopyClustering(points, T1, T2)
  throw-if T1 <= T2
  centroids ← points
  canopies ← []
  while not centroids.isEmpty() do
    p ← centroids.drawRandomElement()
```



В общих чертах алгоритм можно описать несколькими простыми шагами.

1. Выбрать и удалить случайную точку p из набора данных и инициализировать ею новый купол (строки 6–7).
2. Для каждой оставшейся точки q сравнить расстояние между p и q с пороговым значением T_1 (строки 8–10); если расстояние меньше порога, то добавить q к текущему куполу (строка 11).
3. Если расстояние между p и q также меньше второго порога T_2, то удалить q из списка возможных центроидов (строки 12–13), чтобы она не смогла стать центром нового купола.
4. Повторять шаги 1–3, пока не останется ни одной точки (строка 5).

В результате выполнения этих шагов образуются сферические агломераты радиусом (не более) T_1. Внутренний радиус T_2 определяет критическое расстояние, в пределах которого точки можно с уверенностью считать связанными друг с другом (или, что то же самое, принадлежащими одному кластеру). Точка q (на шагах 2–3) уже могла быть добавлена к другому куполу, но если она находится в пределах внутреннего радиуса от центра тяжести текущего купола, то считается, что пара (q, c) тесно коррелирована, и даже если бы мы выбрали q в качестве центроида, то не смогли бы сформировать купол, который лучше соответствует q.

Если вас озадачило, почему можно положиться на эти расстояния и как получить хорошее значение для T_2 (и T_1), значит, вы ухватили суть проблемы: эти параметры необходимо настраивать (пробуя разные комбинации), и иногда опыт и знание предметной области помогают получить хорошие начальные оценки этих расстояний. Например, если требуется сгруппировать географические данные о сотах мобильной связи и известно, что никакие две ячейки не находятся дальше нескольких километров, то можно довольно уверенно предположить величину расстояния T_1.

Определение этих значений не такая уж сложная задача, как могло бы показаться; важно найти не наилучшие, а приемлемые значения для обоих параметров. Как уже упоминалось, этот алгоритм обеспечивает только грубую кластеризацию.

13.1.4. Применение кластеризации методом купола

Кластеризация методом купола часто используется для предварительной обработки перед применением метода k-средних, но ее также можно использовать перед применением DBSCAN и OPTICS и получить еще одно преимущество. Но прежде поговорим о том, как объединить кластеризацию методом купола с методом k-средних.

Самый простой способ: взять грубо нарезанные кластеры (с перекрытием), полученные кластеризацией методом купола, и для каждого вычислить центр масс. Поскольку эти кластеры могут перекрываться, некоторые точки будут принадлежать более чем одному кластеру; следовательно, эти купола нельзя рассматривать просто как результат итерации в методе k-средних! Однако полученные центроиды можно использовать для инициализации, заменив шаг инициализации по умолчанию (случайный выбор) более сбалансированным выбором.

Как вариант, можно запустить кластеризацию методом купола с грубыми значениями T_1 и T_2 и улучшить этап инициализации, выбирая часть начальных центроидов из каждой этой области. Если кластеризация методом купола вернула $m \leq k$ псевдокластеров, то можно взять k/m центроидов из каждого из них. Практические эксперименты показали, что такая начальная инициализация обеспечивает значительное ускорение сходимости метода k-средних.

Как обсуждалось в главе 12, при применении алгоритма DBSCAN (см. раздел 12.3) к наборам данных с неравномерной плотностью возникают проблемы, а применение алгоритма OPTICS (см. раздел 12.4) способен частично решить эти проблемы за счет большей вычислительной нагрузки и экспериментального подбора параметров. В идеале DBSCAN должен запускаться независимо в областях с разной плотностью, и его параметры (или просто значение ε) должны настраиваться отдельно для каждой из этих областей.

Применение кластеризации методом купола в качестве первого шага может помочь решить эту проблему. Запуская DBSCAN отдельно для каждого псевдокластера, можно ожидать, что меньшие регионы будут иметь более однородную плотность, а регионы с разной плотностью, скорее всего, будут отнесены к разным предварительно выделенным кластерам.

Однако вычислением всех кластеров для этих областей дело не ограничивается. Поскольку псевдокластеры могут перекрываться, локальные кластеры тоже могут перекрываться. Помимо проверки необходимости объединения перекрывающихся кластеров, есть и более тонкий эффект: как показано на рис. 13.3, два непересекающихся кластера могут иметь точки, находящиеся в ε-окрестности друг друга! Значит, нужно проверить и те псевдокластеры, расстояние между гиперсферами которых меньше, чем большее из значений ε, используемых для этих областей (в случае если к ним применялся алгоритм DBSCAN с разными значениями гиперпараметров).

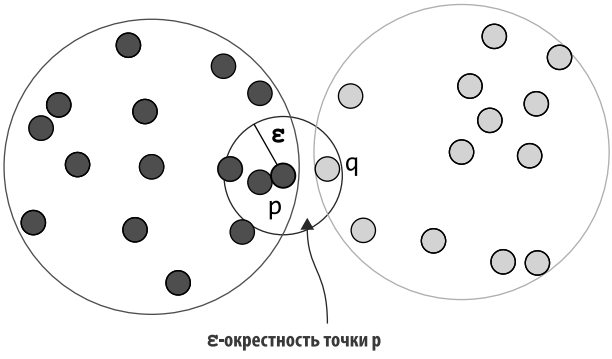


Рис. 13.3. Непересекающиеся псевдокластеры могут рассматривать алгоритмом DBSCAN как принадлежащие одному кластеру, если оба имеют точки, близкие к их границе, и ближайшие точки находятся в пределах ε-расстояния друг от друга

Хорошая новость заключается в том, что для таких куполов не нужно проверять каждую комбинацию точек, полученных из их декартова произведения[1], а ограничиться только точками во внешнем кольце каждой из них на расстоянии от центра купола, большем или равном $T_2 - \varepsilon$.

Можно распараллелить применение DBSCAN к каждому псевдокластеру, но проблема в том, что для объединения результатов нужно проверить все пары кластеров, созданных каждым параллельным процессом кластеризации, и все (отфильтрованные) пары точек, попавших в декартово произведение между внешними кольцами этих кластеров. Обязательно ли выполнять эти проверки в одном потоке на одной машине согласно модели последовательного выполнения?

13.2. MAPREDUCE

Долгое время инженеры пытались найти эффективный способ распараллеливания таких алгоритмов, как DBSCAN, стремясь по крайней мере, на практике преодолеть аппаратные ограничения и сложности, вызванные отсутствием программной инфраструктуры. Наиболее часто используемой моделью распределенного программирования были GRID-вычисления (сетевые параллельные вычисления), пока не стало понятно, что существуют другие подходы, использование которых может сделать вычисления не только более быстрыми, но и надежными. Именно тогда, в начале 2000-х, широкое применение нашла модель программирования MapReduce. К тому моменту она уже была запатентована в Google.

[1] Декартово произведение двух множеств образует новое множество, содержащее все упорядоченные пары, в которых первый элемент принадлежит первому множеству, а второй — второму.

Существует несколько реализаций MapReduce, точнее, несколько продуктов (таких как Apache Hadoop, Hive или CloudDB), использующих модель MapReduce, которые предлагают инструменты для управления распределенными ресурсами с целью решения задач.

Но я считаю, что главная ценность заключается в сущности самой модели, которую можно применить к множеству задач. По этой причине я попытаюсь на примере объяснить, как она работает.

13.2.1. Представьте, что вы Дональд Дак...

Представьте Дональда Дака, дремлющего в своем гамаке и, как всегда, бездельничающего, когда его внезапно будит телефонный звонок, звучащий громче, чем обычно (и более раздражающе!). Еще не успев ничего ответить, он слышит любезное приглашение милого дядюшки Скруджа Макдака поспешить к хранилищу денег — редкая просьба, на которую Дональд откликается с радостью, чтобы не остаться без денег (и не утонуть в долгах).

У его старого доброго дядюшки Скруджа нашлось долгое и скучное задание, которым должен заняться Дональд. На этот раз дядюшка решил переместить все монеты в другой гигантский сейф и хочет, воспользовавшись ситуацией (сюрприз, сюрприз!), пересчитать и каталогизировать все монеты в своем хранилище... к следующему утру.

Речь идет о миллионах монет, поэтому сделать это в одиночку просто невозможно. Когда Дональд Дак приходит в себя (по понятным причинам он упал в обморок, после того как дядюшка Скрудж изложил свою просьбу), он понимает, что ему понадобится любая возможная помощь, поэтому он бежит к Винту Разболтайло и убеждает его создать армию робоклонов, которые смогут научиться распознавать различные монеты и каталогизировать их.

Это «классический» шаг распараллеливания, когда работа (перекладывание монет в стопки) распределяется между несколькими копиями программного обеспечения (выполняющими подсчет/каталогизацию), после чего остается только записать вид монет и их количество в каждой стопке. Например, машина может создать такой список:

```
1 фунт: 1034 монеты
50 центов: 53 982 монеты
20 пенни: 679 монет
1 доллар: 11 823 монеты
1 цент: 321 монета
```

Итак, проблема решена? Не совсем. Роботы-клоны стоят дорого, и для их создания требуется время, поэтому даже такой гений, как Винт, мог создать только сотню, быстро перенастроив несколько неисправных роботов-официантов, которых он создал

в одном из своих экспериментов. Теперь роботы стали довольно быстро считать деньги, но каждый из них собрал огромную кучу монет, в результате чего получился длинный список с номиналами монет и их количествами. Сложившуюся ситуацию иллюстрирует рис. 13.4: как только роботы закончат работу, Дональд должен сложить сотни записей из всех этих сотен списков.

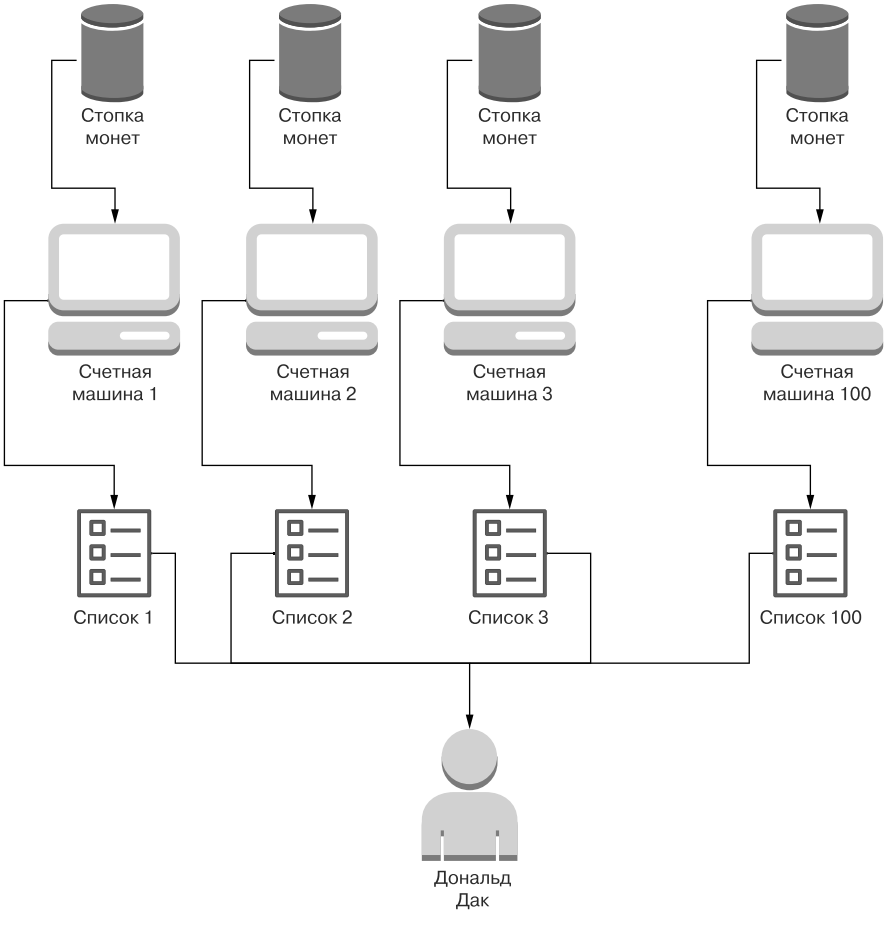


Рис. 13.4. Первая попытка распараллелить подсчет монет. В этом случае бедному Дональду все равно придется просуммировать сотни списков с сотнями записей в каждом

Снова потеряв сознание и очнувшись, когда дядюшка Скрудж поднес к его носу ватку, смоченную нашатырным спиртом (разумеется, Дональду придется заплатить за это), старина Дональд поплелся в лабораторию Винта с отчаянной просьбой о помощи.

К сожалению, Винт не может позволить себе построить больше счетных машин! Но он не был бы гением, если бы не смог решить эту проблему.

Для этого ему не понадобилось ничего строить; достаточно просто использовать другой алгоритм. После быстрых прикидок в уме он подсчитал, что существует около 200 различных номиналов монет. Поэтому он собирает членов своей семьи и дает каждому задание подсчитать количества монет пяти номиналов. Все участники получат по несколько списков (скажем, по сотне) от счетных машин, но в любом отдельном списке будет только пять записей для одних и тех же пяти номиналов монет с их количествами, подсчитанными счетными машинами.

Для этого он записал в каждую счетную машину адреса, по которым можно связаться с конкретными членами семьи Макдаков — например, адреса электронной почты, такие как huey.mcduck@duckmail.com, — и словарь, одинаковый для всех машин, в нем перечислены типы монет, которые будет подсчитывать каждый член семьи. Чтобы упростить ситуацию, можно представить, что за отдельным членом семьи закреплены все монеты из одной страны. Например, как показано на рис. 13.5, Хьюи мог получить все доллары США, Дьюи — все фунты стерлингов Великобритании, Луи — все евро и т. д. Но в реальном приложении каждый из них мог получить любую комбинацию номиналов монет.

Закончив подсчет, машина просматривает список членов семьи и отправляет каждому электронное письмо, в котором сообщает общее количество монет номиналов, соответствующих этому члену семьи. Затем каждый член семьи должен сложить количество монет каждого номинала в каждом списке и отправить окончательный результат дядюшке Скруджу — всего несколько сотен целочисленных сложений на утку. Эта работа, может быть, и утомительная, но она не должна занять слишком много времени (только не подпускайте Фетри к компьютеру!).

Например, если за Дейзи Дак закреплены монеты с номиналами 1 доллар, 50 центов, 25 центов и 1 цент, то все счетные машины отправят ей короткие списки, которые выглядят примерно так:

```
25 центов: 1034 монеты
1 доллар: 11 823 монеты
50 центов: 53 982 монеты
1 цент: 321 монета
```

На рис. 13.5 показана измененная парадигма. Если раньше узким местом вычислений был человек, который должен разобраться во всех списках, то теперь, после добавления промежуточного уровня и распределения работы между участниками на этом уровне, общая скорость решения задачи существенно увеличилась.

Ключевой аспект — возможность разделить на группы результаты, полученные на уровне 1, и обработать каждую из этих групп отдельно на уровне 2.

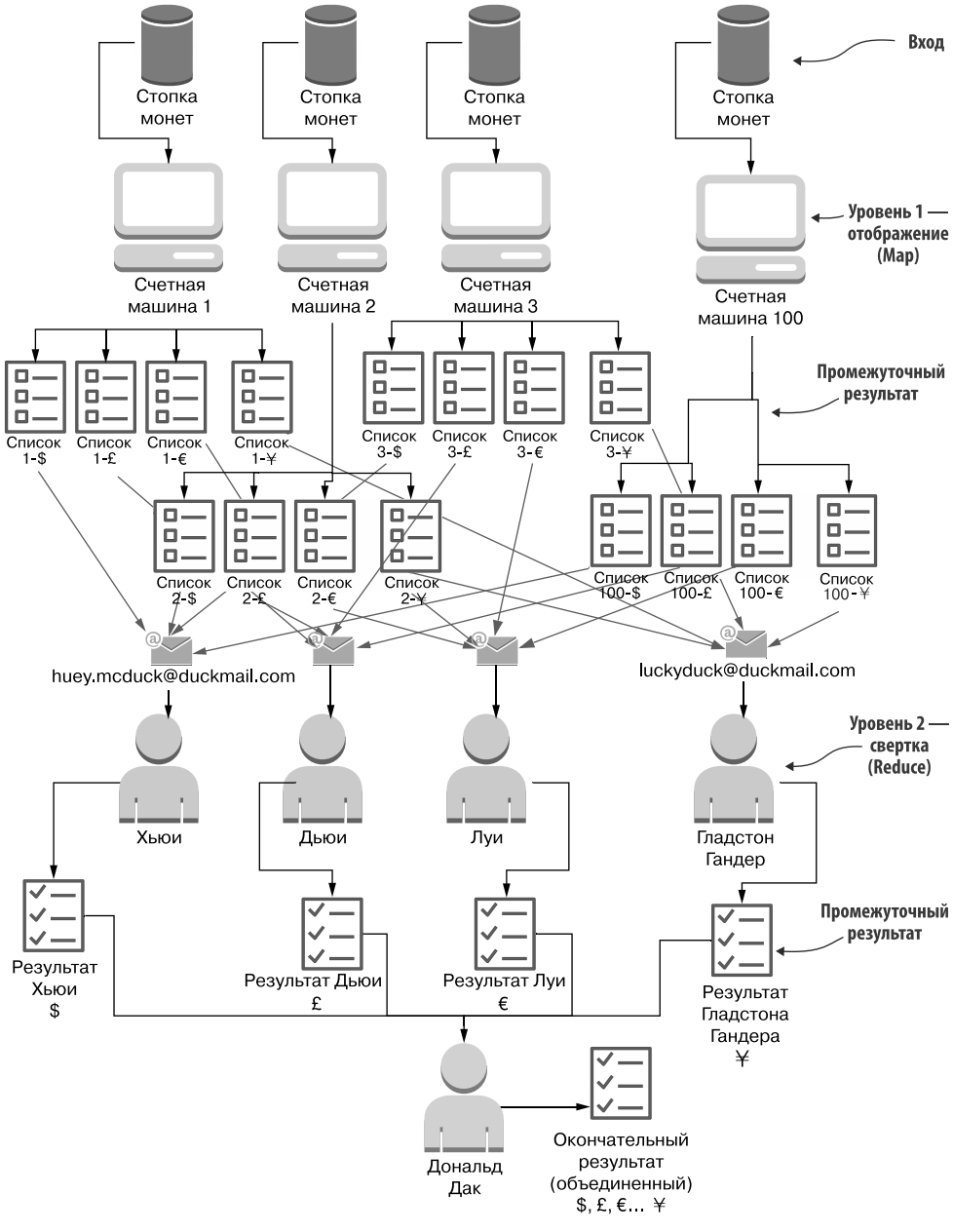


Рис. 13.5. Реорганизованный процесс подсчета монет с использованием модели MapReduce и небольшого изменения. Теперь каждая счетная машина выдает несколько списков, по одному на каждого члена семьи Макдаков на уровне 2. Каждый из них, в свою очередь, должен проверить сотню списков, с небольшим количеством записей в каждом из них, и просуммировать количество монет, указанное в списках

13.2.2. Сначала отображение, потом свертка

Пришло время отказаться от мультяшных героев и взглянуть на более реалистичный пример, когда вычисления на обоих уровнях выполняются машинами. Операция на уровне 1 называется *Map* — «отображение», потому что она отображает каждую запись из входного набора данных (точнее, из части входного набора данных, обрабатываемой машиной) во что-то еще, извлекая информацию, необходимую для вычисления конечного результата. Счетные машины в нашем примере, вероятно, могли бы просто выполнять программное обеспечение, «классифицирующее монеты» без подсчета[1], и отправлять списки с номиналами монет на машины уровня 2, например такие:

```
100 долларов: 1
50 центов: 1
100 долларов: 1
1 доллар: 1
25 центов: 1
...
```

Здесь информация, извлеченная классификаторами, просто отражает наличие монеты того или иного номинала.

Машины уровня 2 будут специализироваться на подсчете. Каждая машина на уровне 2 будет получать все записи для определенных номиналов монет и что-то делать с ними (например, считать их, но также может суммировать их значения или фильтровать). Этот шаг называется *Reduce* — «свертка», потому что он берет информацию, ограниченную однородной группой записей, и объединяет (свертывает) их для получения окончательного результата.

Как уже упоминалось, ключевой недостаток классических «плоских» параллельных вычислений заключается в том, что объединение результатов всех параллельных потоков/процессов/машин является узким местом всего процесса. Если один процесс должен порождать потоки, а затем получать их результаты и объединять, ему все равно придется последовательно обратиться ко всему набору данных хотя бы один раз. И даже если он распараллелит процесс объединения промежуточных результатов, сам все равно останется узким местом, как показано на рис. 13.6. *Слева* можно видеть, что при использовании простого параллелизма «промежуточный результат» отправляется оркестратору, который должен собрать его и отсортировать на машинах уровня 2.

---

[1] Обычно подсчет количества каждого ключа в выходных данных таких распознавателей выполняется третьей абстракцией — объединителями, которые работают на тех же машинах, что и распознаватели, только размещены на выходах распознавателей. Вместо списка с массой записей со значением 1 распознаватель может отправлять список всего с несколькими записями, уменьшая как объем передаваемых данных, так и нагрузку на объединителей.

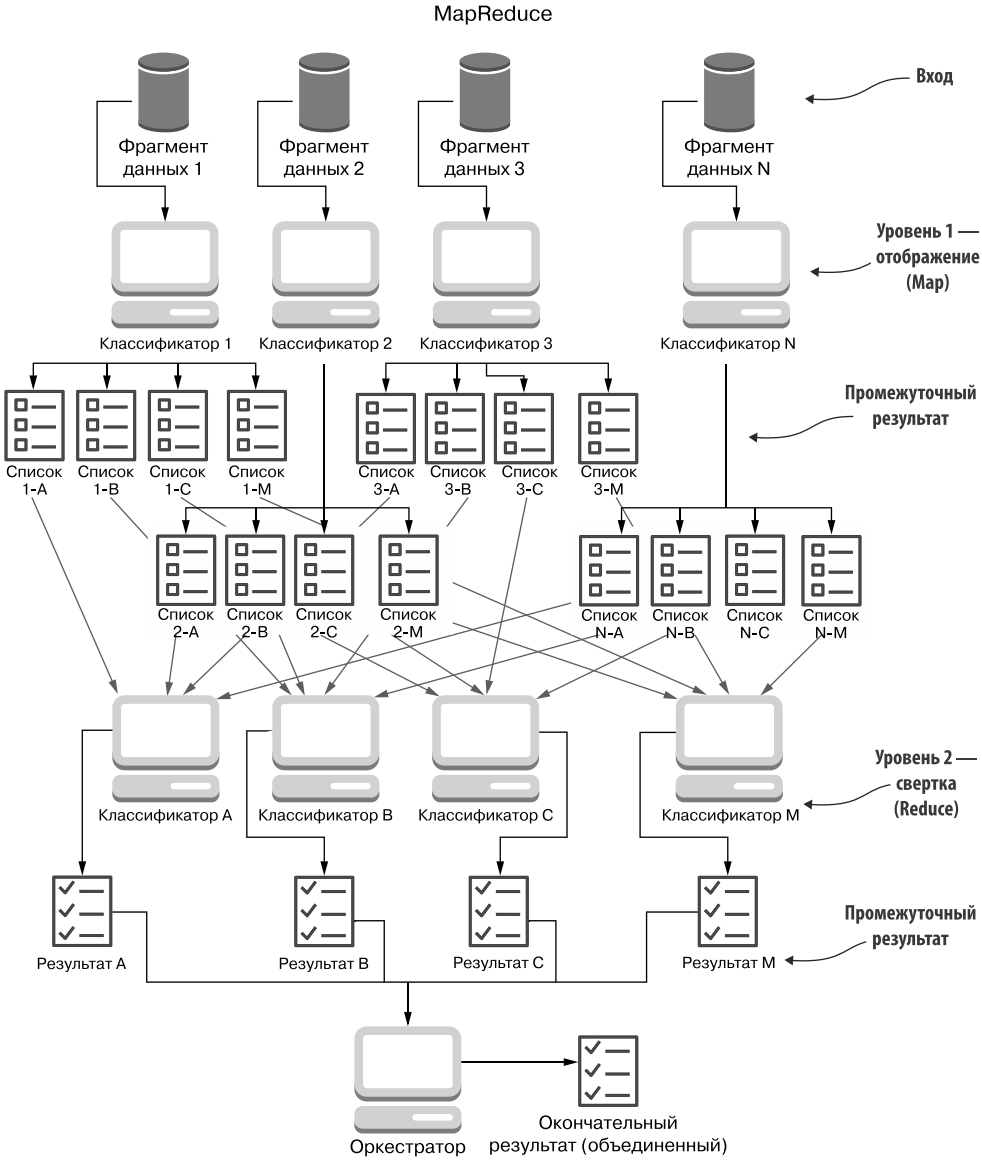


Рис. 13.6. Сравнение классического подхода к параллельным вычислениям с моделью MapReduce. Предполагается, что в обоих случаях данные уже разбиты на фрагменты и могут передаваться по местоположению, то есть через некоторый дескриптор (например, файла) без привлечения оркестратора. В базовом параллельном подходе с использованием процессов (работающих в потоках или на разных машинах) оркестратору необходимо запустить потоки и проанализировать их результаты. Он может объединить эти результаты сам или запустить дополнительные процессы для объединения (как показано на рисунке), но в любом случае сам останется узким местом

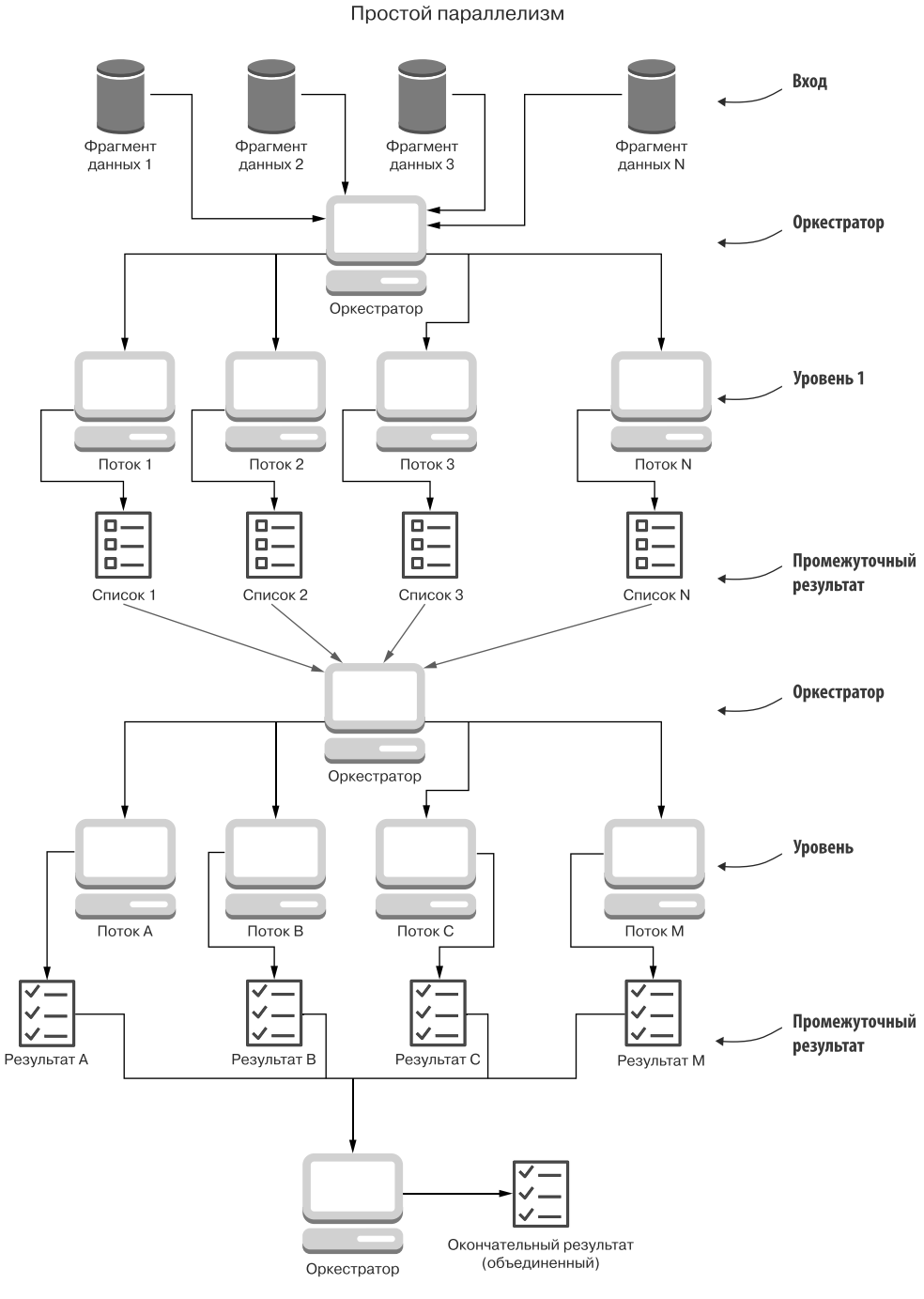


Рис. 13.6. Продолжение.

Однако в MapReduce каждый шаг, по сути, выполняется параллельно. Данные уже разбиты на части, которые можно обрабатывать независимо, и каждый классификатор самостоятельно передает свои результаты объединителям без привлечения центрального оркестратора.

Технически объединители читают информацию из каждого классификатора, а задача классификаторов — создавать временный файл для каждого объединителя в определенном месте (отдельном для каждого объединителя; представьте, например, что каждый классификатор создает для каждого объединителя отдельную папку или выделенный виртуальный диск).

Помимо скорости, подход MapReduce имеет еще одно преимущество: в случае сбоя машины ее можно заменить, не перезапуская все вычисления. Это, в свою очередь, может помочь повысить доступность и сократить задержки за счет выделения избыточных ресурсов для превентивных мер предотвращения неполадок.

Следует особо уточнить, что в модели MapReduce также имеется оркестратор — ведущий узел, который управляет вычислениями: запускает вычислительные узлы или запрашивает ресурсы, передает фрагменты входных данных классификаторам, определяет, как классификаторы будут передавать промежуточные результаты объединителям, обрабатывает ошибки. Однако, в отличие от канонической модели параллелизма, этот специальный узел на самом деле не читает входных данных, ничего не вычисляет и никак не обрабатывает промежуточные результаты, поэтому он не является узким местом в вычислительном конвейере. Тем не менее ведущий узел по-прежнему остается узким местом с точки зрения доступности, потому что в случае его сбоя вычисления не будут завершены. Однако, используя реплики (оперативные копии или первичную реплику), можно получить гарантии доступности за счет (ограниченной) избыточности.

Есть, конечно, некоторые шероховатости. Например, не все вычисления подходят для модели MapReduce (или вообще для параллельного выполнения). В общем случае, если записи данных каким-то образом связаны, а разрозненные фрагменты данных влияют на вклад друг друга в конечный результат, распараллеливание может оказаться невозможным: временные ряды — хороший пример данных, которые обычно приходится обрабатывать последовательно, потому что конечный результат зависит от последовательности обработки данных.

Для модели MapReduce требования еще строже. Чтобы получить преимущество от ее применения, нужны данные, которые можно группировать по каким-то признакам/полям и которые можно объединять по каждой из этих групп в отдельности.

Более того, операция, выполняемая в объединителях, должна быть ассоциативной, то есть порядок объединения промежуточных списков, передаваемых классификаторам, не должен иметь значение.

Стоит отметить, что, если бы вместо каталогизации всех монет потребовалось просто подсчитать их количество (не различая номиналы) или вычислить суммарную стоимость, нам бы не понадобились объединители. Параллельные процессы могли бы просто возвращать свои суммы, а центральный процесс — складывать их.

Вторая загвоздка заключается в отсутствии централизованного объекта, делящего работу и равномерно распределяющего ее между объединителями, поэтому один из них может быть перегружен, а другой — простаивать без дела. Если вернуться к нашей истории, например, Дейзи Дак придется подсчитать все американские монеты, которых в хранилище наверняка очень много, а Гладстону Гандеру достанутся все редкие монеты из маленьких стран (ему повезло), поэтому полученные им списки будут почти пустыми и ему понадобится выполнить всего несколько сложений.

13.2.3. Еще кое-что

Вы познакомились с некоторыми преимуществами модели MapReduce, но ее успех обусловлен не только ими — есть и другие аспекты, которые не сразу видны.

В главе 7, говоря о кэше и многопоточности, мы обсуждали блокировки и синхронизацию. Каждому параллельному вычислению с общим состоянием требуется синхронизация, будь то агрегирование результатов, разбивка данных и назначение их вычислительным единицам (потокам или машинам) или просто проверка завершения обработки.

Ключевое преимущество модели MapReduce в том, что она ограничивает объем общего состояния до минимума (за счет использования концепций функционального программирования, таких как неизменность[1] и чистые функции[2]), предлагая парадигму программирования, способ определения проблемы, который вынуждает формулировать задачу так, чтобы устранить общее состояние[3] и сократить необходимость в синхронизации.

---

[1] Методы неизменяемой структуры данных не изменяют объект A, для которого они вызываются, а создают новый объект B, состояние которого является результатом применения вызываемого метода к A. Например, метод, добавляющий элемент в список L1, создаст совершенно новый список L2 с $|L1| + 1$ элементами и оставит L1 без изменений.

[2] Чистой называется любая функция, не имеющая побочных эффектов: она принимает 0, 1 или более входных данных и возвращает результат (возможно, в виде кортежа), не изменяя глобального состояния. В некотором смысле чистая функция подобна математической функции.

[3] Это еще одна причина, почему, как отмечалось в предыдущих подразделах, описанную модель вычислений нельзя применить к задачам, которые нельзя сформулировать так, чтобы исключить совместное состояние.

13.3. РЕАЛИЗАЦИЯ МЕТОДА K-СРЕДНИХ С ПОМОЩЬЮ МОДЕЛИ MAPREDUCE

Чтобы эффективно распараллелить любой алгоритм, сначала нужно ответить на два вопроса: как точки данных влияют на вычисления и какие данные действительно нужны для выполнения определенного шага?

В обоих случаях кластеризации, методами *k*-средних и куполов, способ выполнения различных шагов диктует реализацию MapReduce. Рассмотрим каждый шаг в методе *k*-средних по отдельности[1].

- На *этапе классификации*, когда происходит привязка точек к куполам/кластерам, вычисления для каждой точки выполняются независимо, и единственное, что имеет значение, — список центроидов. Соответственно, данные можно сегментировать как угодно при условии, что каждый классификатор получает все центроиды.
- На этапе *повторного центрирования*, чтобы обновить каждый конкретный центроид, в методе *k*-средних, в котором каждая точка может быть связана только с одним центроидом, достаточно иметь только множество точек, связанных с центроидом, поэтому набор данных можно разбить на группы и обрабатывать каждую группу отдельно.
- Этап *инициализации* для метода *k*-средних сложнее; начальные точки должны случайно извлекаться из полного набора данных, и это, казалось бы, препятствует распараллеливанию. Однако и в этом случае есть несколько возможных стратегий распределения вычислительной нагрузки на выбор:
  - произвольно сегментировать набор данных, а затем из каждого извлечь случайный центроид (правда, в этом случае сложнее получить общее равномерное распределение образцов центроидов);
  - сначала выполнить кластеризацию методом купола и передать полученные центроиды алгоритму *k*-средних как начальный выбор центроидов, но тогда возникает вопрос: можно ли распараллелить кластеризацию методом купола? Как оказывается, это возможно, хотя и немного сложнее.

Из сказанного следует, что распараллеливание кластеризации методом купола является ключевым шагом, но также и самой сложной частью. Мы обсудим эту проблему ниже, а пока просто рассмотрим полное общее описание распределенного алгоритма *k*-средних:

- инициализировать центроиды, используя кластеризацию методом купола;
- повторить (не более *m* раз):
  - классифицировать точки:
    - сегментировать набор данных и отправить фрагменты вместе со списком центроидов классификаторам;

[1] Здесь шаги метода *k*-средних перечислены в порядке возрастания сложности.

    - каждый классификатор связывает точки с одним из центроидов и передает данные, агрегированные по выбранному центроиду, объединителям (в идеале на каждый центроид будет приходиться один объединитель);
- обновить центроиды:
    - каждый объединитель должен вычислить центр масс своего кластера и вернуть новый центроид.

Обобщенная реализация[1] этого алгоритма представлена в листинге 13.2, а ее работа для нашего примера показана на рис. 13.7.

Листинг 13.2. Реализация метода k-средних с помощью модели MapReduce

Разбиваем набор данных на numShards случайных фрагментов. Обычно это делается автоматически ведущим узлом MapReduce, но в этом случае используется собственная версия

Инициализируем numShards классификаторов; каждый будет вызывать метод classifyPoints, и все они будут оставаться активными в течение всего времени выполнения этого метода (и хранить копию одного из фрагментов набора данных). Классификаторы будут хранить одну и ту же копию входных данных в течение всего времени выполнения этого метода

Метод MRkmeans принимает список точек, количество желаемых фрагментов, на которые должен быть разбит набор данных, количество желаемых кластеров и два пороговых значения, T1 и T2, которые будут использоваться на этапе инициализации алгоритма кластеризации методом купола

Инициализируем центроиды, используя распределенную версию кластеризации методом купола

```
function MRkmeans(points, numShards, numClusters, maxIter, T1, T2)
  shards ← randomShard(points, numShards)
  centroids ← MRcanopyCentroids(points, numClusters, T1, T2)
  mappers ← initMappers(numShards, classifyPoints, shards)
  reducers ← initReducers(numClusters, centerOfMass)
  for i in {0, .., maxIter-1} do
    newCentroids ← mapReduce(centroids, mappers, reducers)
    if centroids == newCentroids then
      break
    else
      centroids ← newCentroids
  return combine(mappers, centroids)
```

Если ни один центроид не изменился, значит, алгоритм сошелся и можно прервать главный цикл

Иначе копируем обновленные кластеры из временной переменной

Выполняем одну итерацию MapReduce, используя уже созданные классификаторы и объединители. Классификаторы получат текущий список центроидов (кроме того, каждый классификатор уже содержит один фрагмент точек из набора данных). Объединители будут получать свои входные данные (точки в одном из кластеров) от классификаторов. Каждый объединитель будет возвращать координаты одного центроида, а результаты, полученные всеми объединителями, будут включаться в список во временной переменной

Используем комбинаторы для запуска классификаторов в последний раз, чтобы получить окончательную классификацию точек с учетом текущих центроидов

Повторяем главный цикл до maxIter раз

Инициализируем numClusters объединителей, по одному для каждого центроида. Объединители будут вызывать метод centerOfMass (вычисляющий центр масс набора точек) и оставаться активными в течение всего времени выполнения этого метода, но не будут хранить никаких данных

[1] Имейте в виду, что представлена не настоящая реализация, поэтому здесь используются некоторые сокращения, чтобы упростить объяснение основных идей. Задание в Hadoop MapReduce, например, будет выглядеть иначе.

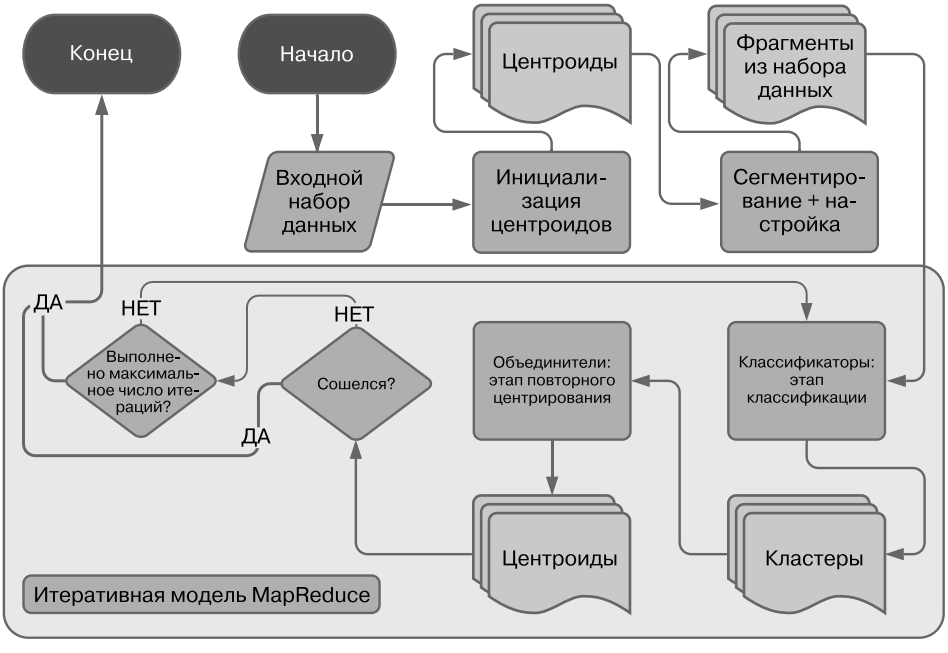


Рис. 13.7. Блок-схема процесса кластеризации методом k-средних, реализованного с использованием итеративной модели MapReduce

Первое, на что следует обратить внимание, — в данном случае мы не говорим о старой доброй модели MapReduce. Да, модель MapReduce лежит в основе алгоритма, но, поскольку эвристика *k*-средних состоит из повторения некоторых шагов *m* раз, потребуется выполнить вычисления несколько раз. Это иллюстрирует схема процесса кластеризации на рис. 13.7: выходные данные каждого задания MapReduce, список центроидов, одновременно являются входными данными следующего задания. И кроме того, сегментирование набора данных и его распределение между классификаторами — шаг, который не требует повторного выполнения для каждого задания, потому что нет причин, по которым фрагмент, переданный классификатору, должен измениться.

По изложенным причинам, вместо запуска MapReduce *m* раз в отдельности можно использовать более эффективную (в данном контексте) версию модели программирования, *итеративную модель MapReduce*[1].

Идея состоит в том, чтобы запустить классификаторы и объединители один раз, сегментировав данные во время настройки классификаторов и назначая фрагменты с точками каждому классификатору только один раз. Затем классический цикл MapReduce повторяется столько раз, сколько необходимо, и каждому классификатору передается только текущий список центроидов. Таким образом, объем данных,

[1] Хорошей отправной точкой для начального знакомства с итеративной версией MapReduce может послужить статья http://mng.bz/8NG5.

передаваемых каждому классификатору, будет на несколько порядков меньше размера исходного набора данных, а в идеале он к тому же будет значительно меньше размера каждого фрагмента. Можно также предусмотреть передачу улучшенному методу реализации k-средних количества классификаторов и настраивать этот параметр в зависимости от размера набора данных и емкости каждого классификатора.

На рис. 13.8 хорошо видно, как происходят вычисления, начиная с этого момента. Каждый классификатор выполняет этап классификации в группе переданных ему точек. Затем объединители (в идеале по одному на кластер) получают данные от каждого классификатора: i-й объединитель получит только точки, назначенные i-му центроиду (или центроидам, назначенным объединителю, если их несколько).

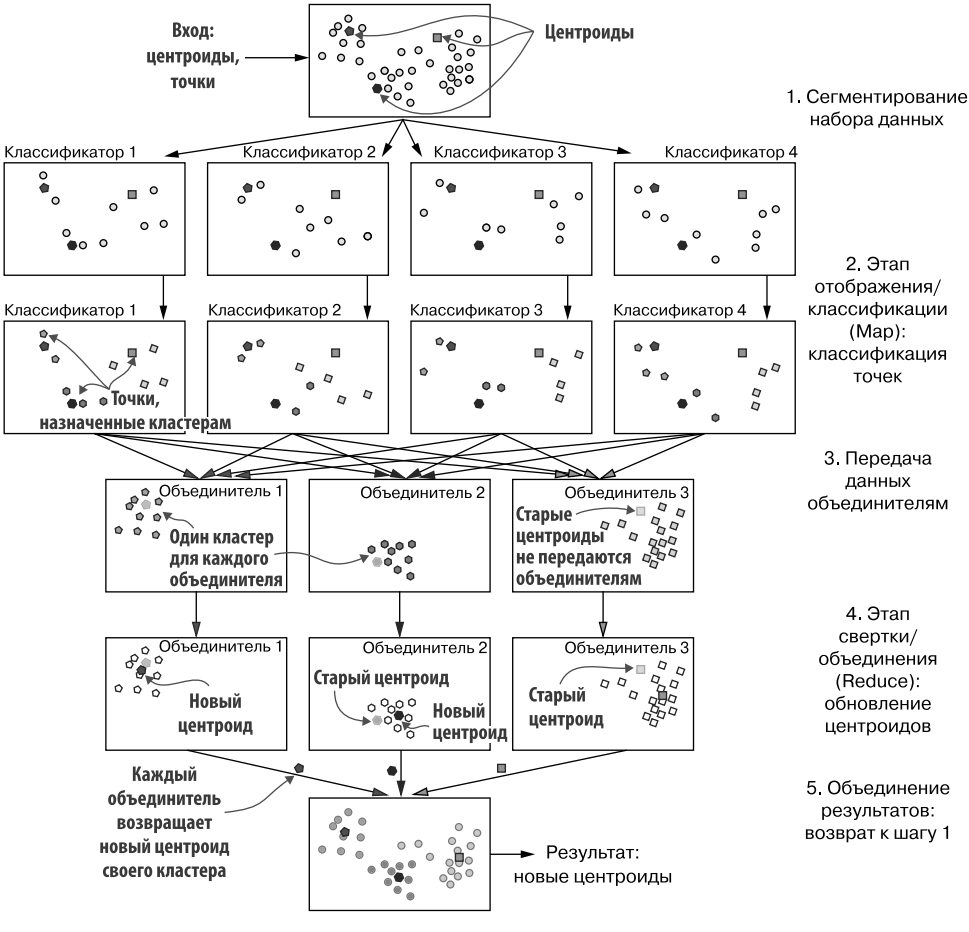


Рис. 13.8. Итерация основного цикла кластеризации методом k-средних, реализованная с помощью итеративной модели MapReduce. С учетом блок-схемы на рис. 13.7 первый шаг, показанный здесь, — это сегментирование плюс настройка, а остальная часть показывает одну итерацию «итеративной модели MapReduce»

Обратите внимание, что объединители не получают никакой информации о текущих центроидах (на иллюстрации на шаге свертки/объединения старые центроиды показаны как полупрозрачные полигоны), потому что каждому объединителю нужны только точки[1], принадлежащие кластеру, чтобы вычислить его центр масс.

Каждый объединитель в конечном итоге выводит вычисленный центроид (и только его; точки не возвращаются, чтобы сэкономить на передаваемом объеме данных и в конечном счете на времени, необходимом для этого). А ведущий узел MapReduce объединит k результатов (где k — количество центроидов/объединителей) в один список, который снова будет передан классификаторам в следующей итерации цикла!

По завершении цикла получим только центроиды, поэтому нужно запустить классификаторы в последний раз вне цикла, чтобы получить точки, связанные с каждым кластером (можно представить, что на этом шаге, в строке 12 в листинге 13.2, будет создан новый набор объединителей: фиктивные транзитные узлы, просто возвращающие свои входные данные).

Эта реализация k-средних использует алгоритм кластеризации методом купола только для ускорения сходимости за счет получения лучшего набора центроидов, который можно получить случайным выбором. Увеличение эффективности этапов распределения фрагментов между классификаторами и повторного центрирования уже дает большой выигрыш, и велика вероятность, что полученного выигрыша будет более чем достаточно для удовлетворения ваших требований.

Тем не менее, даже притом что кластеризация методом купола выполняется быстрее, чем одна итерация в методе k-средних, и может быть ускорена еще больше за счет использования недорогой приближенной метрики вместо евклидова расстояния, при работе с огромными наборами данных есть риск потерять бóльшую часть выгоды, полученной путем распараллеливания реализации k-средних с применением модели MapReduce. Это может случиться, если запускать кластеризацию методом купола на одной машине. К сожалению, иногда возможность разнести вычисления по нескольким компьютерам недоступна даже для огромных наборов данных, которые не помещаются ни на одной машине.

К счастью для нас, модель MapReduce можно применить и к кластеризации методом купола!

13.3.1. Распараллеливание кластеризации методом купола

Распараллелить кластеризацию методом купола немного сложнее, потому что этот алгоритм состоит всего из одного шага: извлечения центроидов куполов из набора данных и фильтрации точек, находящихся на определенном расстоянии от них, чтобы впоследствии они не были выбраны на роль центроидов. Проблема в том,

[1] Технически вычисление среднего значения набора значений тоже можно распараллелить: объединитель должен получить на входе центроиды фрагментов и количество точек в каждом фрагменте для вычисления общего центра масс.

что на этом этапе для каждого центроида, полученного из набора данных, нужно обойти все точки данных, чтобы выполнить фильтрацию. Теоретически для каждого центроида достаточно обработать только точки в его куполе, но мы не можем идентифицировать их заранее!

Чтобы достичь хорошего решения, полезно подумать о действительной цели кластеризации методом купола, а она такова: получить набор центроидов куполов, которые находятся не ближе некоторого расстояния T_2 друг к другу. Ключевой момент — между любыми двумя центрами куполов расстояние должно быть больше минимального значения, чтобы купола не перекрывались слишком сильно. Как уже упоминалось, это расстояние подобно «основному расстоянию» в DBSCAN, и можно предположить, что точки в радиусе T_2 с высокой вероятностью принадлежат одному и тому же кластеру.

Предположим, мы разделили наш исходный набор данных, как показано на рис. 13.9. Если затем к каждому фрагменту в отдельности применить кластеризацию методом купола, то получится определенное количество центроидов, вероятно, разное для разных классификаторов. Однако если соединить снова эти центроиды, то сойдет на нет гарантия, что они будут соответствовать требованию находиться не ближе чем на расстоянии T_2 друг от друга, потому что центроиды из разных фрагментов не сравнивались друг с другом.

Однако рано сдаваться! К счастью, есть простое решение этой проблемы, и решением является... все та же кластеризация методом купола!

На самом деле, если собрать вместе все центроиды из каждого классификатора, к этому новому (меньшему) набору данных можно снова применить алгоритм кластеризации методом купола, на этот раз имея гарантию, что никакие две точки в результате этого второго прохода не будут находиться ближе чем на расстоянии T_2. Взгляните на последний этап на рис. 13.9, это поможет вам понять, как работает второй проход. Решение довольно эффективно, потому что размер нового набора данных (содержащего только центроиды, созданные классификаторами) на несколько порядков меньше исходного набора данных (при условии, что расстояния T_1 и T_2 были выбраны правильно), и поэтому на шаге 2 можно запустить один объединитель и выполнить кластеризацию методом купола на одной машине и на всех центроидах, полученных на первом шаге.

Теперь я должен прояснить несколько моментов. При использовании в качестве предварительного шага для метода k-средних алгоритм кластеризации методом купола имеет немного другую цель (и другой результат), чем при использовании в роли самостоятельного алгоритма грубой кластеризации:

- для k-средних достаточно вернуть список центроидов;
- самостоятельный алгоритм также должен вернуть точки, связанные с каждым куполом.

То есть эти два случая должны рассматриваться по отдельности.

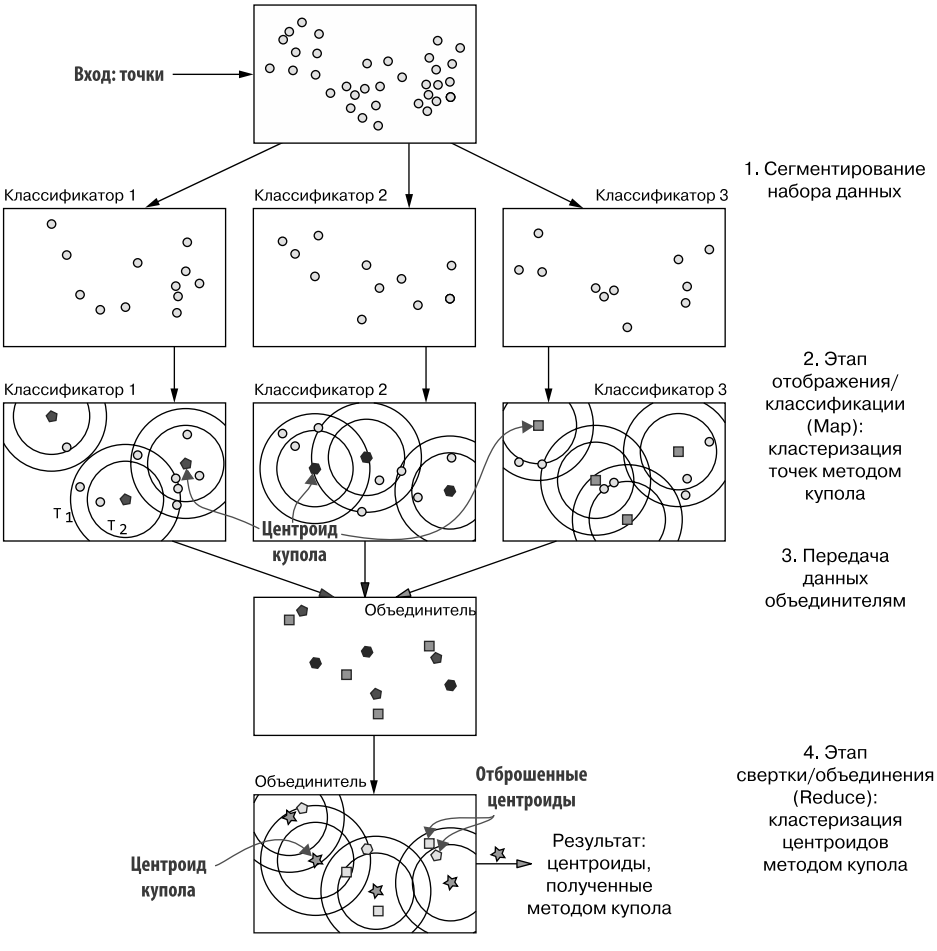


Рис. 13.9. Кластеризация методом купола, реализованная с применением модели MapReduce

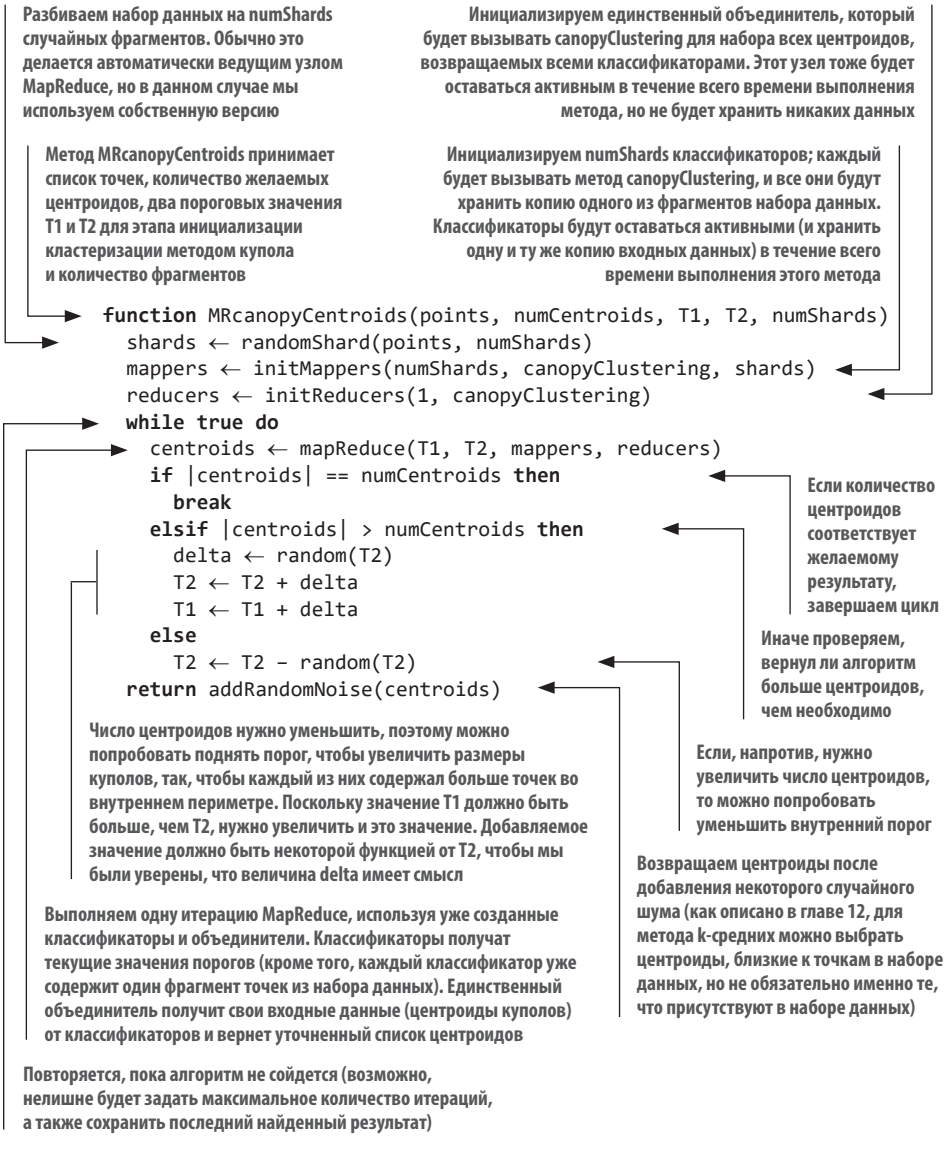
13.3.2. Инициализация центроидов с помощью кластеризации методом купола

Начнем с кластеризации методом купола в качестве шага инициализации для метода *k*-средних. В листинге 13.3 представлена возможная реализация задания MapReduce, решающего эту задачу. На первый взгляд, не нужно делать ничего, кроме того, что было показано в предыдущем разделе: мы просто возвращаем результат объединителя — список центроидов.

Но есть одна загвоздка (загвоздка всегда есть!): как выбрать количество центроидов, которое должно быть возвращено алгоритмом?

Дело в том, что управлять этим числом напрямую возможно только через значения двух порогов, переданных в алгоритм, и то лишь до определенной степени. В конце концов, алгоритм представляет собой рандомизированную эвристику, и количество создаваемых куполов может меняться при каждом запуске, даже при одинаковых значениях гиперпараметров.

Листинг 13.3. Генерирование центроидов методом купола с использованием модели MapReduce



Чтобы решить проблему, необходимо мыслить нестандартно. Поскольку существует сильная случайная составляющая, влияющая на результат каждого прогона, можно запустить кластеризацию методом купола несколько раз и получить результат, который ближе всего к нашим ожиданиям. Однако этого может быть недостаточно, потому что дисперсия между разными запусками ограничена, и, если начать с «неправильных» значений порогов, алгоритм всегда может выдать слишком много (или слишком мало) центроидов.

Существует два варианта решения проблемы: либо вручную настраивать пороги после каждого прогона, либо, как вариант, выполнять какой-то поиск, пробуя разные значения порогов: можно добавлять при каждом прогоне случайное значение к первоначальному выбору для T_1 и T_2, или же настраивать пороги в зависимости от количества возвращаемых куполов (уменьшая T_2, когда в последнем прогоне получено слишком мало центроидов, и увеличивая его, когда их получено слишком много). При желании можно даже использовать метод максимального подобия.

Первая идея предполагает рандомизированный поиск методом перебора, но целенаправленное решение кажется более многообещающим, потому что позволяет направить поиск в сторону значений, которые должны дать лучшие результаты. Можно даже использовать градиентный спуск[1], но с более простым алгоритмом, который просто определяет направление обновления, не заботясь о наклонах и градиентах. Например, можно запустить цикл, в котором будет корректироваться значение внутреннего порога (и при необходимости значение внешнего порога) в зависимости от разницы между полученным результатом и количеством необходимых центроидов.

Однако, учитывая фактор случайности в этом алгоритме, можно с очевидностью утверждать, что описанный процесс — все еще простой поиск возможных значений T_2, и он рискует зациклиться. Чтобы исправить ситуацию, резонно добавить условие (и соответствующий аргумент), проверяющее превышение максимального количества итераций. Одновременно можно предусмотреть сохранение результата, наиболее близкого к искомому, во временной переменной, обновляемой при каждом запуске, и возвращать этот результат всякий раз, когда достигнуто максимальное количество итераций, даже если не было найдено ровно `numCentroids` куполов. В большинстве случаев оказывается приемлемым вернуть, например, 101 центроид вместо 100 (у вызывающего кода будет возможность проверить результат и принять решение).

Я оставляю вам в качестве самостоятельного упражнения добавление в листинг 13.3 автоматического выбора порогов, а сейчас перейдем к описанию распределенной версии алгоритма полноценной кластеризации методом купола, который возвращает не только центроиды, но и перекрывающиеся наборы точек, связанные с каждым куполом.

---

[1] Градиентный спуск — это алгоритм оптимизации поиска локальных минимумов функции F. Он систематически исследует область определения функции, выполняя шаги, пропорциональные градиенту F в текущей точке X, который (упрощенно) можно геометрически интерпретировать как направление наибольшего изменения F в X.

13.3.3. Кластеризация методом купола с использованием модели MapReduce

Этап *классификации*, связывающий каждую точку с одним или несколькими куполами, может быть реализован несколькими разными способами, приведу их ниже.

- Как продолжение метода, описанного в листинге 13.3. Однако поскольку классификация будет проверять все точки, то, если не распараллелить этот шаг, мы потеряем большую часть преимуществ применения алгоритма для параллельного выбора центроидов.
- Как тот же метод, что и при инициализации центроидов, но с другим заданием MapReduce.
- В том же задании MapReduce, что описано в предыдущем разделе, но с объединителем, который также выполняет классификацию и одновременно выбирает центроиды для сохранения. В частности, объединитель получит от классификаторов все списки точек, связанных с каждым центроидом, поэтому вместо повторного перебора всех точек в наборе данных он может повторно использовать эти списки (далее в этом разделе я покажу, как это реализовать).

Первый вариант проще, но он слишком наивен. Реализация шага классификации хотя бы тем же методом, каким выполняется выбор центроидов, дает два основных преимущества:

- теоретически можно повторно использовать одни и те же классификаторы, которые уже хранят свои фрагменты данных, просто передать им список центроидов куполов;
- при классификации возникает проблема с фильтрацией центроидов, которая выполняется в объединителях. До сих пор ее можно было игнорировать, но совместный запуск фильтрации центроидов и классификации позволяет эффективно решить эту проблему.

На рис. 13.10 показано, что может произойти, если в предыдущем алгоритме отфильтровать один или несколько центроидов на этапе объединения. До сих пор об этой проблеме не упоминалось, потому что она влияет на алгоритм, только когда точки привязываются к куполам, и не проявляется, когда выполняется простой поиск центроидов, как на этапе инициализации в методе *k*-средних.

Проблема в том, что, когда отфильтровывается какой-то из центроидов, выбранных на этапе классификации, часть точек может оказаться за пределами купола, даже того центроида, который был выбран вместо него. *Слева* на рис. 13.10 изображен центроид, выбранный из списка центроидов (он отмечен звездочкой), и отфильтровывается другой центроид, оказавшийся в пределах внутреннего радиуса T_2 выбранного. Однако затененная область не охватывается куполом с центром в другом центроиде (обозначенном звездочкой).

Иногда потерянный охват компенсируется другими куполами, но не всегда: *справа* на том же рис. 13.10 показан пример, когда несколько точек остаются неохваченными после выбора центроидов, которые нужно сохранить.

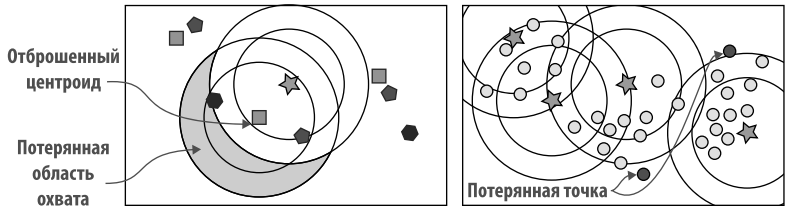


Рис. 13.10. Влияние фильтрации центроидов (на шаге объединения в кластеризации методом купола с использованием модели MapReduce) на охват набора данных куполами. *Слева:* когда центроид C отбрасывается из-за того, что он слишком близок к выбранному, часть области, охватываемой куполом C, становится неохваченной. *Справа:* это может привести к тому, что некоторые точки окажутся за пределами какого бы то ни было купола

Есть несколько вариантов решения этой проблемы.

- Рассматривать «потерянные» точки как выбросы (хотя это не очень надежное решение, потому что точки могут лежать в середине какого-то большого кластера, который не может быть охвачен одним куполом).
- Увеличить размеры куполов: отброшенные центроиды можно соответствующим образом отметить в процессе кластеризации методом купола, когда выбирается один из центроидов. В этот момент можно проверить внешний радиус, связанный с каждым куполом, и сделать его достаточно большим, чтобы охватить все точки, связанные с куполом отброшенного центроида. Самый дешевый способ сделать это — установить радиус всех куполов равным $T_1 + T_2$, но это, очевидно, увеличит перекрытие между куполами.
- Выполнить перебор неохваченных точек (находящихся на расстоянии больше T_1 от любого из уцелевших центроидов) в конце этапа классификации и назначить их ближайшему центроиду. Это решение минимизирует перекрытие куполов (радиус каждого купола будет не больше $T_1 + T_2$ и не больше расстояния до самой дальней из этих новых точек), но оно более дорогостоящее.
- Если выполнить классификацию в том же задании MapReduce, где происходит выбор центроидов, можно получить вполне эффективное решение. Классификаторы также должны будут создать список с наборами точек, связанных с каждым центроидом, и передать этот список объединителю вместе со списком центроидов всех фрагментов. В объединителе запустится специальный вариант кластеризации методом купола, и когда извлечется центроид C, наборы точек, связанных со всеми центроидами в радиусе T_2 от C, объединятся и окажутся связанными с куполом C. Это лучшее решение с точки зрения производительности, потому что экономит одну итерацию по всем точкам во всех сегментах и требует только одного объединителя. Однако оно имеет свой недостаток: объединителю требуется передать множество списков куполов, а также специализированную версию алгоритма кластеризации методом купола, обрабатывающую слияние центроидов.

В листинге 13.4 представлена обобщенная реализация алгоритма кластеризации методом купола с использованием модели MapReduce и с классификацией, выполняемой тем же методом, но во втором задании MapReduce с учетом выбора центроидов.

В строках 2–5 выполняется тот же алгоритм, который в листинге 13.3 реализует распределенное вычисление центроидов куполов. Строки 6–9, с другой стороны, запускают код, специфичный для этой версии, связывая каждую точку p со всеми куполами, центры которых находятся от p на расстоянии не больше $T_1 + T_2$. Как уже упоминалось, выбор большего радиуса гарантирует, что ни одна точка не останется неохваченной.

Листинг 13.4. Реализация кластеризации методом купола с помощью модели MapReduce

Метод `classifyPoints`[1], выполняющий привязку точек, запускается в каждом классификаторе применительно к каждому фрагменту в отдельности. Что касается

[1] Реализация classifyPoints не включена в книгу, но ее можно получить из листинга 12.3, изменив проверяемое условие, но не забывая, что в этом случае метод ищет не ближайший центроид, а все центроиды на определенном расстоянии. Кроме того, функция classifyPoints была бы подходящим способом решения проблемы с неохваченными точками. Напомню, что самым простым и универсальным решением могло бы быть использование порогового расстояния, равного $T_1 + T_2$.

метода *k*-средних, этот шаг можно выполнять независимо для каждой точки, если у классификатора есть полный список центроидов.

Результатом работы каждого классификатора будет список списков с одним элементом для каждого центроида: списком точек, связанных с этим центроидом. Обратите внимание, что каждая точка может быть связана как минимум с одним, но потенциально с несколькими центроидами. Каждый классификатор будет иметь элементы для нескольких центроидов, и каждый центроид будет иметь точки, связанные с ним несколькими классификаторами: вот почему в этом задании MapReduce также нужны объединители по числу куполов (то есть по числу центроидов).

Затем каждый объединитель займется обработкой одного купола, объединяя списки для этого купола, созданные каждым классификатором. Конечным результатом будет список куполов, производимых каждым объединителем.

Эта подпрограмма, описанная в строках 6–8, также проиллюстрирована на рис. 13.11.

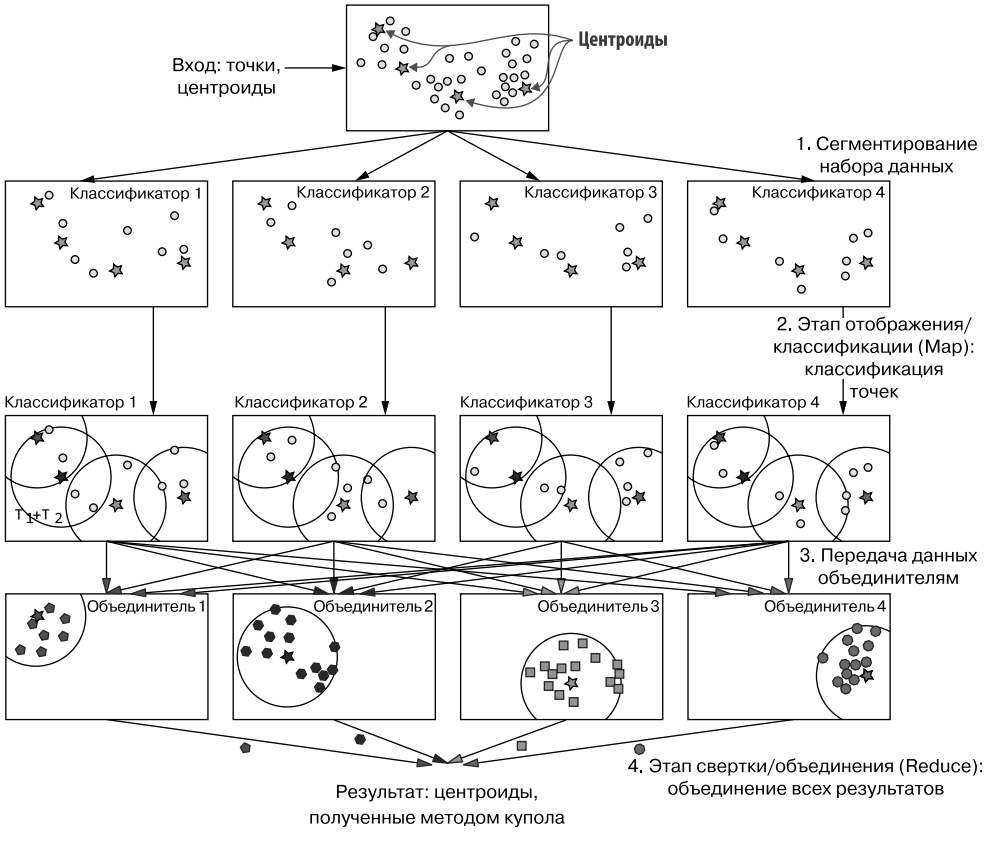


Рис. 13.11. Пример задания MapReduce для классификации точек между куполами по списку центроидов

На этом завершаем обсуждение кластеризации методом купола. Далее я предлагаю читателю попытаться самостоятельно написать задание для этого алгоритма, используя одну из реализаций MapReduce с открытым исходным кодом, например Hadoop. Обратите также внимание на Mahout — фреймворк распределенных вычислений линейной алгебры от Apache Foundation, который имеет свою распределенную реализацию кластеризации методом купола[1].

13.4. РЕАЛИЗАЦИЯ DBSCAN С ИСПОЛЬЗОВАНИЕМ MAPREDUCE

Пока все идет нормально. В нашей первой попытке создать распределенную реализацию кластеризации мы взяли за основу метод *k*-средних, и нам повезло: оказалось, что его легко переписать в распределенной форме, потому что шаги алгоритма можно выполнять независимо для каждой точки (классификация) или для каждого кластера (повторное центрирование). Была также применена модель MapReduce к кластеризации методом купола, даже притом что первый его шаг, извлечение центроидов куполов, не поддавался распараллеливанию и требовал более глубоких рассуждений.

Чтобы замкнуть круг и завершить тему, в этом разделе обсудим применение парадигмы MapReduce к алгоритму кластеризации, который по своей сути не является распараллеливаемым, по крайней мере на первый взгляд: к DBSCAN.

Как было показано в разделе 12.3, в алгоритме DBSCAN кластеры вычисляются путем изучения и использования отношений между точками. Сегментация набора данных изменит ε-окрестность большинства точек, а некоторые из основных точек могут не распознаваться как таковые, потому что их ε-окрестности окажутся разбросаны по нескольким фрагментам.

Можно, конечно, подумать о задании MapReduce, параллельно вычисляющем размеры ε-окрестностей всех точек, но ядро DBSCAN полагается на последовательный перебор точек и их соседей, и поэтому кажется, что лучше поискать какой-то другой вариант.

В подразделе 13.1.4 уже показано, что в качестве первого шага резонно использовать кластеризацию методом купола, применить DBSCAN к каждому куполу отдельно, а затем итеративно объединить кластеры в соседних куполах, когда точки, близкие к их границам или в перекрывающихся областях, находятся в пределах ε-окрестности друг от друга.

Однако реализовать этот подход проблематично по нескольким причинам, представленным в списке ниже.

- Трудно отслеживать купола, которые необходимо проверить, и точки внутри куполов для сравнения. Необходимо сравнить все пары куполов, чтобы исследовать,

[1] http://mng.bz/E2EX.

перекрываются ли они и находятся ли на расстоянии меньшем, чем ε, а затем для каждой пары куполов сравнить все точки с другим куполом.

- Учитывая форму оболочек (гиперсфер), сложно вычислить расстояние между точками внешних колец купола и другим куполом (см. подраздел 10.4.3 и рис. 10.23, чтобы получить представление о сложных геометрических последствиях для двумерного случая, а они осложняются еще больше для гиперсфер в более высоких измерениях).
- Правильно определить пороги для кластеризации методом купола сложно: из-за сферической формы куполов придется использовать больший, чем необходимо, радиус T_1 и допускать значительное перекрытие между куполами, чтобы зафиксировать отношения между кластерами в разных куполах.
- Когда купола имеют существенное перекрытие, появляется большое число точек, связанных с несколькими куполами, такие точки нужно обрабатывать несколько раз для каждой пары куполов. Это легко может превратиться в вычислительный кошмар и свести к нулю все усилия по распараллеливанию.

Таким образом, описанное решение имеет ограниченное применение в конкретных случаях и конфигурациях, но не подходит для наборов многомерных данных.

И все же основная идея верна, и мы можем заставить ее работать, просто изменив способ сегментирования набора данных. Вместо деления данных на случайные фрагменты или сферические купола его можно разбить на обычную сетку с ячейками одинакового размера. Ячейки образуют гиперпрямоугольники, а не гиперсферы, и для каждой координаты область может быть разделена по-разному, в результате чего разные стороны ячеек (гиперпрямоугольников) будут иметь разную длину.

В этом случае упрощается идентификация точек, близких к границам. Можно просто сравнить абсолютное значение разности между точкой и границей по некоторой координате с порогом вместо вычисления расстояния между точкой и гиперсферой; кроме того, само сегментирование набора данных по сетке обходится проще и дешевле.

Но это не все хорошие новости! Вместо сравнения точек, близких к границам соседних ячеек, можно определить сетку с немного перекрывающимися ячейками и точно перекрывающимися для каждой координаты в прямоугольной области со стороной ε, как показано на рис. 13.12.

Таким образом, каждая ячейка перекрывает соседнюю на ширину 2ε, и ε-окрестность каждой точки на перекрывающемся участке будет частью хотя бы одной из двух соседних ячеек (например, точка p на рисунке имеет свою ε-окрестность, полностью содержащуюся в S_1). Хитрость заключается в том, что, если p является основной точкой в одном из фрагментов, она также будет основной точкой в объединении фрагментов, и, следовательно, ее соседи по обеим сторонам от границы ячеек должны быть напрямую достижимы[1] из p и, в свою очередь, оказываются в од-

[1] Определение достижимости и некоторые примеры вы найдете в подразделе 12.3.1.

ном кластере. Следовательно, если p связывается с кластером C_1 в фрагменте S_1 и с кластером C_2 в фрагменте S_2, то из этого следует, что кластеры C_1 и C_2 должны быть объединены.

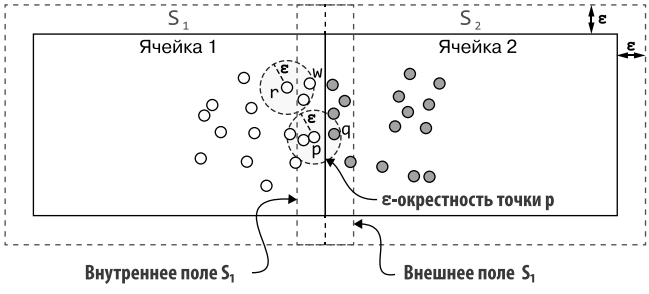


Рис. 13.12. Сегментирование данных для алгоритма DBSCAN с использованием модели MapReduce (MR-DBSCAN). После сегментирования области на ячейки одинакового размера в каждый фрагмент нужно включить все точки, находящиеся в одной ячейке, плюс все точки в пределах расстояния ε от ее границ. На практике вместо квадратных ячеек используются прямоугольники, полученные путем растягивания ячейки во все стороны, и каждая из сторон будет иметь размер, равный соответствующей стороне ячейки плюс 2ε. В результате основные точки, близкие к границам ячеек, будут иметь свои полные ε-окрестности, даже проникающие в соседние фрагменты

И наоборот, если взять любую точку, находящуюся за границей фрагмента, например r на рис. 13.12, которая расположена слева от внутреннего поля S_1, то мы сможем уверенно утверждать, что:

- ее расстояние от границы фрагмента больше ε, поэтому ее ε-окрестность не пересекается с внешним полем S_1;
- если в S_2 есть точка z, достижимая из r, то должна существовать цепочка основных точек, непосредственно достижимых из z и r (определение достижимости приводится в подразделе 12.3.1), и по крайней мере одна из этих точек — пусть это будет w — должна находиться либо во внутреннем, либо во внешнем поле S_1, потому что эти области простираются ровно на расстояние, равное 2ε, что в точности равно диаметру ε-окрестности основных точек;
- следовательно, точку r можно игнорировать, потому что ее кластер будет объединен с z в ходе исследования точки w.

Из этого неформального доказательства[1] вытекает, что вместо сравнения каждой точки в ячейке со всеми точками в соседних ячейках, близкими к границе, можно просто отслеживать основные точки на расстоянии ε от границы ячейки (или 2ε от

[1] Формальное доказательство и подробное описание алгоритма можно найти в статье: *He Y. et al.* MR-DBSCAN: A scalable MapReduce-based DBSCAN algorithm for heavily skewed data // Frontiers of Computer Science, 2014. — Vol. 8.1 — P. 83–99.

границы фрагмента) и объединять кластеры, которые имеют точки во внутреннем или во внешнем поле, с основной точкой в любой из соседних ячеек.

На рис. 13.13 показан пример того, как задание MapReduce будет выполнять распределенную кластеризацию набора двумерных данных с использованием DBSCAN, а в листинге 13.5 — этап объединения, описанный в этом разделе.

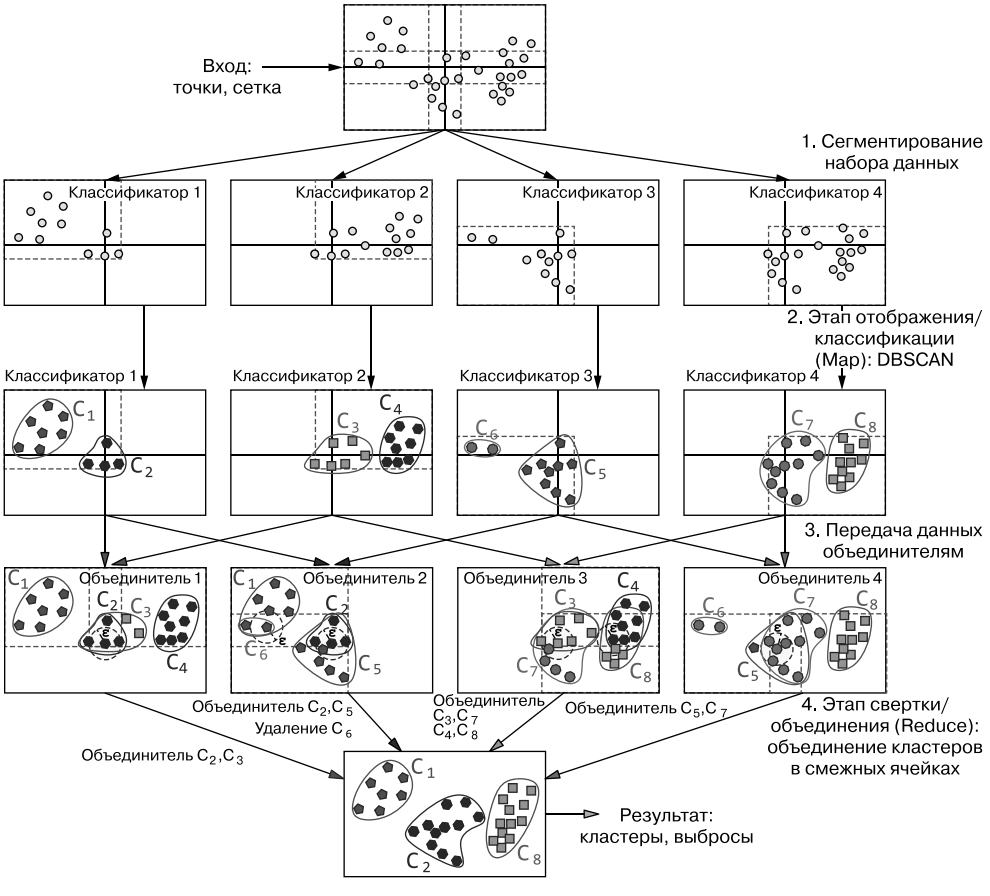


Рис. 13.13. Пример работы алгоритма DBSCAN, реализованного с использованием модели MapReduce (алгоритм работает с четырьмя ячейками). Нам понадобится один преобразователь для каждой (расширенной) ячейки и один объединитель для каждой пары соседних ячеек. Набор данных в этом примере невелик, и используются всего четыре ячейки с большим значением ε, поэтому доля точек в полях на границах ячеек необычно высока. В реальных ситуациях точек на границах обычно меньше, а экономия, получаемая за счет распараллеливания алгоритма, на порядки выше

Первый шаг — сегментирование набора данных по обычной сетке. Каждая ячейка сетки затем расширяется во всех направлениях на величину ε, и формируются

фрагменты путем фильтрации точек внутри этих расширенных прямоугольников (которые, что важно помнить, пересекаются с соседними ячейками).

Листинг 13.5. Реализация алгоритма DBSCAN с помощью модели MapReduce

```
function MRdbscan(points, numShards, eps, minPts)
  shards, adjList, marginPoints ← gridShard(points, numShards, eps).
  mappers ← initMappers(numShards, dbscan, shards, eps, minPts)
  reducers ← initReducers(adjList, mergeClusters, marginPoints)
  clusters, noise, mergeList ← mapReduce(mappers, reducers)
  return combine(clusters, noise, mergeList)
```

Разбиваем набор данных на numShards ячеек; каждая ячейка будет растянута на величину eps во всех направлениях, чтобы определить фрагменты, которые будут сохранены в shards. Поскольку сегментирование выполняется по сетке, нужна отдельная функция сегментирования, возвращающая также список смежных фрагментов (лучше, если это будет список пар смежных фрагментов без дубликатов) и список точек в краевых областях для каждой пары смежных фрагментов

Метод MRdbscan принимает список точек, количество ячеек для сегментирования набора данных и параметры алгоритма DBSCAN — радиус и минимальное количество точек, определяющих плотную область

Инициализируем numShards классификаторов; каждый будет запускать метод, реализующий алгоритм DBSCAN и хранить копию одного из фрагментов. Классификаторы будут оставаться активными (и хранить одну и ту же копию данных) в течение всего времени выполнения этого метода

Перед возвратом нужно сгруппировать результаты, возвращаемые объединителями, объединив кластеры (и, возможно, переиндексировав их) и зафиксировав список выбросов

Выполняем итерацию MapReduce, используя уже созданные классификаторы и объединители. Классификаторы будут использовать уже имеющиеся у них фрагменты данных. Объединители будут получать входные данные (кластеры и выбросы, локальные для фрагментов) от классификаторов (в дополнение к уже имеющейся у них информации о том, какие точки находятся в промежуточном поле между каждой парой соседних фрагментов). Каждый объединитель будет возвращать для каждой пары фрагментов список кластеров для слияния и точек-выбросов, которые нужно сохранить

Инициализируем по одному объединителю для каждой пары фрагментов в списке смежных фрагментов. Объединители будут вызывать метод mergeClusters, оставаться активными в течение всего времени выполнения этого метода, хранить данные о смежных фрагментах, с которыми они работают, а также о точках в поле между этими двумя фрагментами

Каждый классификатор выполняет кластеризацию по алгоритму DBSCAN в своем фрагменте, а на следующем шаге для каждой пары соседних фрагментов запускается объединитель (в этом примере использована сетка 2×2, поэтому получилось четыре пары соседних ячеек).

Объединители должны получить от каждого классификатора список найденных кластеров, список точек шума (если они есть) и список точек, попавших в поле между двумя фрагментами, которые будет обрабатывать объединитель.

Затем объединитель проверяет присутствие основной точки в области общего поля и, если таковая имеется, объединяет два кластера, которым принадлежат точки. В примере на рис. 13.13 намеренно использована глобальная инкрементная индексация

кластеров на этом этапе, но на самом деле выполнить такую глобальную индексацию за один проход нелегко!

Это объясняется тем, что заранее не известно, сколько кластеров найдет классификатор. Объединение кластеров можно выполнять и локально, но в какой-то момент потребуется дополнительный шаг для глобальной переиндексации всех кластеров.

Однако если предположить, что кластеры имеют глобальную индексацию, можно будет обрабатывать объединяющиеся кластеры, используя структуру данных, с которой вы уже хорошо знакомы: непересекающееся множество, описанное в главе 5.

Возможно, вы заметили, что есть крайние случаи, о которых следует помнить. Например, кластеры в одном фрагменте могут быть подмножествами более крупного кластера в другом фрагменте. В примере объединителю 2 достаются кластеры C_1 и C_6, при этом последний полностью включается в первый. В этом случае, даже если ни одна точка в пограничных областях не является основной, очевидно[1], что нужно объединить два кластера (или, что то же самое, избавиться от C_6).

Точно так же, если в одном фрагменте точка p классифицируется как основная точка, а в другом как шум, то объединения кластеров не произойдет: p уже находится в правильном кластере, но при этом важно проявить осторожность и исключить ее из списка выбросов.

Каждый объединитель возвращает список или, как вариант, непересекающееся множество локальных кластеров для слияния. Следующий короткий этап объединения может позаботиться о создании списка с точками, закрепленными за конечными кластерами, и списка выбросов на основе результатов работы объединителей.

Псевдокод для этого задания MapReduce проще, чем для любого другого задания в этой главе, но пусть вас не обманывает эта простота — основные сложности скрыты в методах `gridShard`, `dbscan` и `mergeClusters`.

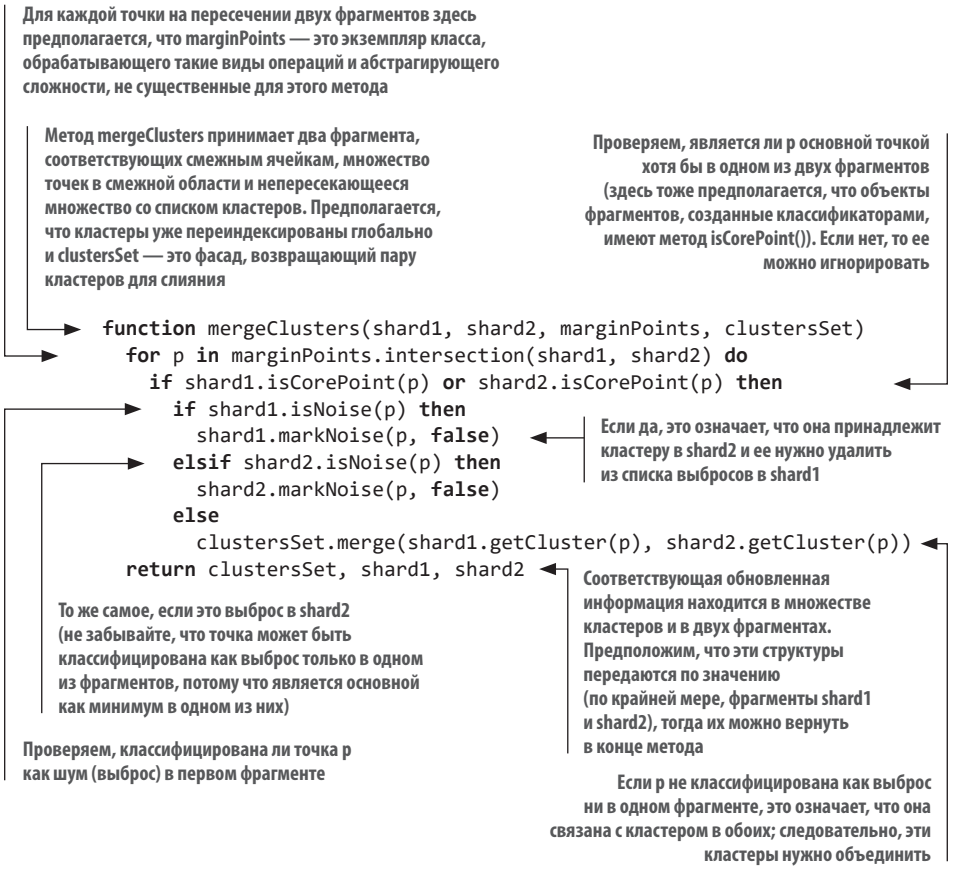
Метод `dbscan` в точности соответствует описанному в разделе 12.3. В большинстве языков программирования его можно использовать повторно без каких-либо изменений. `gridShard` в своей самой простой версии просто перебирает точки и вычисляет индекс ячейки, выполняя деление по модулю (плюс выполняет несколько проверок, чтобы выяснить, находится ли точка на границе соседних ячеек). Далее в этом разделе мы рассмотрим некоторые проблемы, связанные с этим методом, но не будем вдаваться в подробности его реализации.

Наконец, метод `mergeClusters` вполне подходит для применения непересекающегося множества, структуры данных, описанной в главе 5. Возможная реализация этого

[1] C_6 не подходит для слияния с C_1, потому что ни одна из его точек в пограничной области не является основной. Чтобы распознать такие ситуации, нужно явно выполнить некоторую дополнительную проверку (например, для каждой пары кластеров установить, является ли один из них подмножеством другого).

метода показана в листинге 13.6. Она принимает аргумент `clustersSet` — экземпляр непересекающегося множества, совместно используемого всеми объединителями. С практической точки зрения это невозможно[1], но концептуально это эквивалентно возврату из объединителей списка кластеров для слияния и выполнению операций над непересекающимся множеством на этапе слияния после того, как объединители закончат работу. По этой причине `clustersSet` можно рассматривать как фасад[2], упрощающий процесс создания пар кластеров, формирования фактического непересекающегося множества и выполнения операции слияния кластеров.

Листинг 13.6. Метод mergeClusters



[1] Наличие общего объекта в модели MapReduce лишено всякого смысла! Сможете объяснить почему? (Подсказка: кроме очевидных технических проблем, так ли необходимо вводить общее состояние?)

[2] https://en.wikipedia.org/wiki/Facade_pattern (https://ru.wikipedia.org/wiki/Фасад_(шаблон_проектирования). — *Примеч. пер.*).

Реализация выполняет обход всех точек в пограничной области между двумя фрагментами (используйте для справки рис. 13.12) и проверяет, является ли какая-либо из этих точек основной точкой хотя бы в одном из фрагментов. Затем остается только убедиться, что эта точка не является выбросом в другом фрагменте (обрабатывается как пограничный случай), и объединить два кластера (C_1 будет в `shard1`, а C_2 — в `shard2`). При желании можно также проверить, не является ли C_1 или C_2 подмножеством другого, и обработать этот случай отдельно.

Прежде чем завершить обсуждение реализации алгоритма DBSCAN с использованием модели MapReduce, необходимо рассмотреть еще одну деталь: этап сегментирования. Остановитесь на минуту и поразмышляйте, чем этот случай отличается от того, что вы видели раньше, и с какими проблемами можно столкнуться на этом этапе.

Заметили проблему? Детерминированное сегментирование точек по прямоугольной сетке не такая дешевая операция, как случайное сегментирование, которое вы видели до сих пор! На самом деле перед настройкой и запуском задания MapReduce для выполнения этого сегментирования необходимо запустить однопоточный процесс, создающий сетку и назначающий каждую точку тому или иному фрагменту в зависимости от ее местоположения. Если сетка состоит из m ячеек, а набор данных содержит n точек с d координатами каждая, то такой шаг потребует в худшем случае $O(n \times d \times m)$ сравнений.

Чтобы ускорить процесс, можно использовать R-дерево. Как упоминалось в главе 10, R-деревья могут хранить объекты с ненулевыми размерами и, в частности, такие фигуры, как прямоугольники (см. рис. 10.5). Иначе говоря, можно создать R-дерево, элементами которого являются ячейки сетки, и для каждой точки найти ближайшую ячейку. Однако, учитывая, что R-деревья имеют линейное время работы в худшем случае, их применение не улучшит асимптотический результат (но на практике R-деревья часто обеспечивают более быстрый поиск, чем простой метод перебора).

Однако, чтобы существенно ускорить процесс, нужно распределить также этап сегментирования, определив отдельное задание MapReduce. Как известно, каждую точку можно сравнивать с ячейками независимо друг от друга. Поэтому, если разделить набор данных на случайные фрагменты, то их можно обработать с помощью целой батареи классификаторов. Объединители в этом случае просто группируют точки по расширенным ячейкам[1] и, наконец, создают (на этот раз не случайным образом) новые фрагменты, которые затем можно использовать на первом этапе задания MR-DBSCAN (показанном на рис. 13.13). В качестве дополнительной оптимизации, учитывая, что объединители для этого задания (сегментирования) уже будут иметь всю информацию о ячейках, те же машины можно перепрофилировать на роль классификаторов в задании MR-DBSCAN.

[1] Поскольку ячейки перекрываются, точка может быть назначена более чем одной ячейке.

Стоит также отметить, что количество смежных ячеек линейно растет с размерностью пространства, так как количество граней d-мерного гиперкуба равно $2d$. Это означает, что алгоритм сегментирования может масштабироваться до более высоких размерностей.

Наконец, стоит упомянуть, что, выполняя шаг сегментирования как независимое задание MapReduce, необходимо использовать обычную сетку. Напротив, в статье Xe (He) и его коллег, где представлен алгоритм MR-DBSCAN[1], применяется другой, более сложный подход, когда на первом проходе собирается статистика о наборе данных. Тем самым область разбивается на нерегулярный набор параллельных прямоугольников (рис. 13.14), каждый из которых имеет идеально однородную плотность. Используя собранную статистику, можно настроить параметры DBSCAN для адаптации к разной плотности в разных ячейках.

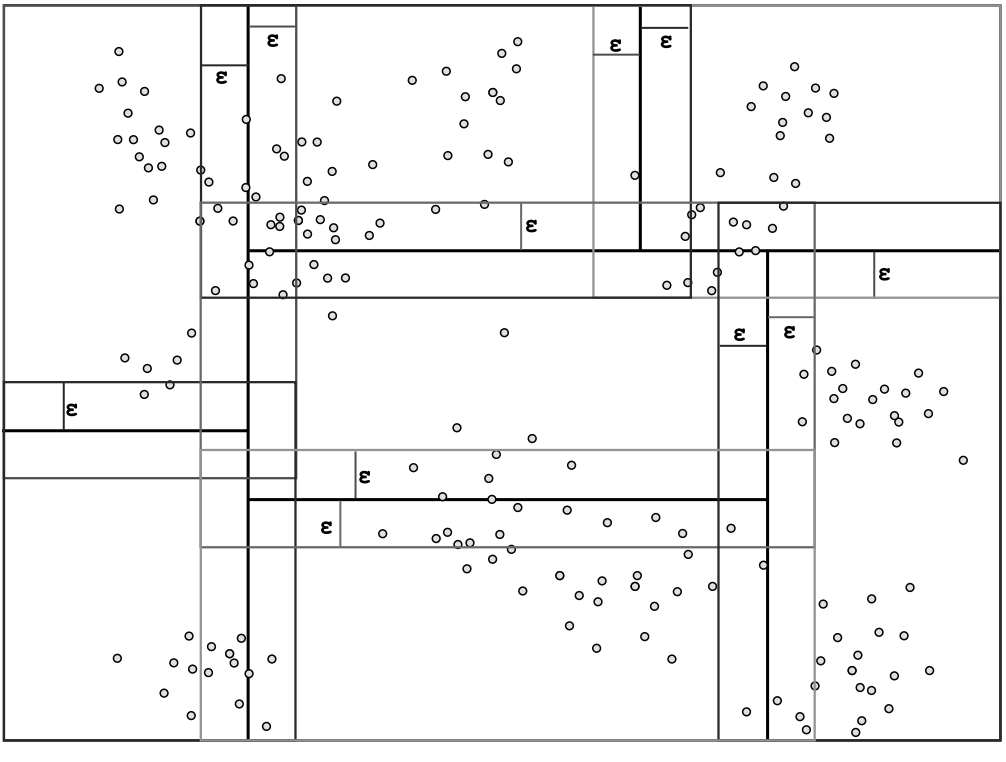


Рис. 13.14. Пример сетки неправильной формы для сегментирования набора данных в MR-DBSCAN. Обратите внимание, что фрагменты (прямоугольники, образованные пунктирными линиями) строятся путем расширения ячеек в каждом направлении на расстояние, равное ε — радиусу плотной области

[1] *He Y. et al.* MR-DBSCAN: A scalableMapReduce-based DBSCAN algorithm for heavily skewed data // Frontiers of Computer Science. 2014. — Vol. 8.1 — P. 83–99.

РЕЗЮМЕ

- Кластеризация методом купола — это алгоритм грубой псевдокластеризации набора данных. Основное его преимущество — он очень недорогой по сравнению с более точными алгоритмами, такими как метод *k*-средних или DBSCAN.
- Различаются параллельные вычисления, выполняемые в нескольких потоках на одном компьютере, и распределенные вычисления, когда программное обеспечение запускается на нескольких компьютерах и, возможно, использует ресурсы облака.
- MapReduce — это вычислительная модель, использующая облако для масштабирования обработки больших наборов данных.
- Метод *k*-средних, кластеризацию куполом и DBSCAN можно переписать в стиле для распределенного исполнения с использованием модели MapReduce.

Часть III

Планарные графы и минимальное число пересечений

В материалах заключительной части книги фигурирует в основном единственная структура данных: *графы*. Однако упоминаться и использоваться в последующих главах они будут скорее как пробный камень для сравнения различных методов, а основное внимание будет уделено *алгоритмам оптимизации*.

Мы не будем углубляться в основы графов, но начнем эту часть с краткого введения в основные понятия. Вы познакомитесь с некоторыми базовыми алгоритмами обхода графов.

Это знакомство необходимо для исследования интересной задачи, которую часто игнорируют: отображения графов в двумерной плоскости. Ее частенько не принимают в рассмотрение, хоть сама задача и находит широкое применение в нашей отрасли. Проблема сложная, не имеет эффективного решения на классических компьютерах. Но часто бывает достаточно найти ее приближенное решение, а это уже дает нам хороший повод познакомиться с алгоритмами оптимизации, настоящей жемчужиной третьей части.

В последних главах книги описываются три метода, широко используемые для решения задач оптимизации и стимулирующие развитие в области искусственного интеллекта и больших данных: *градиентный спуск*, *имитация отжига* и *генетические алгоритмы*.

Глава 14 содержит краткое введение в *графы*, основы этой фундаментальной структуры данных, необходимые для понимания всего остального, о чем рассказывается в части III. Материал в ней описывает алгоритмы *DFS*, *BFS*, *Дейкстры* и *A** и порядок их применения для решения задачи поиска кратчайшего пути.

Глава 15 знакомит с *векторным представлением графов* (graph embeddings), планарностью и парой задач, которые мы попытаемся решить в оставшихся главах: поиском *минимального числа пересечений* (Minimum Crossing Number, MCN), векторным представлением графов и отображением графов в виде диаграмм.

Глава 16 описывает *градиентный спуск* — фундаментальный алгоритм машинного обучения — и показывает, как его можно применить к графам и векторным представлениям графов.

Глава 17 основана на предыдущей главе и представляет *имитацию отжига* — более мощную, по сравнению с упомянутыми выше, технику оптимизации, цель которой заключается в преодолении недостатков градиентного спуска, когда приходится иметь дело с недифференцируемыми функциями или функциями с несколькими локальными минимумами.

Наконец, в главе 18 описываются *генетические алгоритмы* — еще более продвинутый метод оптимизации, помогающий ускорить сходимость.

14 Введение в графы: поиск кратчайшего пути

В этой главе

- ✓ Представление графов с теоретической точки зрения.
- ✓ Стратегии обучения для реализации графов.
- ✓ Поиск наилучшего пути в графе.
- ✓ Знакомство с поисковыми алгоритмами на графах: BFS и DFS.
- ✓ Использование BFS для поиска пути, пересекающего наименьшее количество блоков.
- ✓ Поиск кратчайшего пути с помощью алгоритма минимального расстояния Дейкстры.
- ✓ Поиск кратчайшего пути с помощью алгоритма A* оптимизации поиска.

Вы уже познакомились с основами деревьев в приложении B и использовали несколько видов деревьев в предыдущих главах: двоичные деревья поиска, кучи, *k*-мерные деревья и т. д. Графы можно было бы считать обобщением деревьев, однако в действительности все как раз наоборот: именно деревья являются частным случаем графов. Дерево, по сути, является связным ациклическим неориентированным графом. На рис. 14.1 показан пример двух графов, из которых только один является деревом. Не беспокойтесь, если вы затрудняетесь с ходу определить, какой из них — дерево. В этой главе нам с вами предстоит подробно рассмотреть свойства, присущие именно деревьям, и затем это знание поможет лучше понять свойства графов и разобраться в этом на примерах.

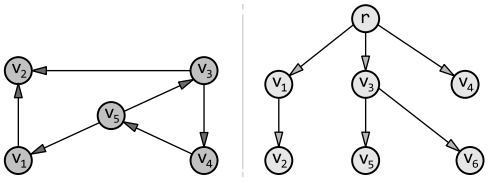


Рис. 14.1. Два графа, из которых только один является также деревом. Можете определить — какой из них?

Но не будем забегать вперед, сначала дадим формальное определение графов и попробуем понять, как можно их представлять. Заложив основу, затем сможем начать моделировать интересные задачи с помощью графов и разрабатывать алгоритмы для их решения.

В частности, в этой главе сосредоточимся на задаче поиска кратчайшего пути. В главах 8–11 был разработан пример платформы электронной коммерции. А чтобы найти ближайший склад такой, чтобы отправляемые из него посылки преодолевали наименьшее возможное расстояние, были применены k-мерные деревья и поиск ближайшего соседа. Однако, чтобы доставить заказ покупателю, все равно придется преодолеть некоторое расстояние, и чем больше времени и топлива тратить на доставку, тем меньше окажется наша прибыль. И наоборот, сумев оптимизировать проложенный маршрут, мы могли бы сэкономить немного денег на каждой посылке и много денег в масштабе тысяч или миллионов заказов.

В этой главе решим задачу поиска наилучшего пути для доставки одного заказа (от склада до дома покупателя) с помощью графов, используем для этого разные уровни абстракции, чтобы показать, как работают алгоритмы поиска, такие как BFS, Dijkstra и A* (произносится как «A-звездочка»).

14.1. ОПРЕДЕЛЕНИЯ

Граф G обычно определяется в терминах двух множеств:

- множество *вершин V*: независимых, отдельных объектов, которые могут появляться в любом порядке. Граф может иметь 1, 2, 100 и любое другое их количество, но, как правило, графы не поддерживают повторяющихся вершин;
- множество *ребер E*, соединяющих вершины: ребро определяется парой вершин, первая из которых обычно обозначается как *исходящая* вершина, а вторая — как *входящая* вершина.

Итак, мы пишем $G = (V, E)$ для явного указания, что граф состоит из определенных множеств вершин и ребер; например, граф на рис. 14.2 формально записывается как:

$$G = ([v_1, v_2, v_3, v_4], [(v_1, v_2), (v_1, v_3), (v_2, v_4)])$$

Ребро, исходящая и входящая вершины которого совпадают, называется *петлей* (рис. 14.3). *Простые графы* не могут иметь ни петель, ни нескольких ребер между

одной и той же парой узлов. И наоборот, мультиграфы могут иметь любое количество ребер между двумя различными вершинами. И простые графы, и мультиграфы можно расширить и разрешить циклы.

В этой книге не будем обсуждать мультиграфы и сосредоточимся исключительно на простых, обычно без петель.

Предыдущие определения можно выразить более формально. Учитывая множество ребер E:

- для простых графов — $E \subseteq \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$;
- для простых графов, поддерживающих петли, — $E \subseteq V^2$.

Каждому ребру можно сопоставить вес. В этом случае граф называется *взвешенным графом* или, что то же самое, *сетью*, где каждое ребро становится тройкой, а множество ребер графа становится:

- для простых графов — $E \subseteq \{(x, y, w) \mid (x, y) \in V^2 \wedge w \in \mathbb{R} \wedge x \neq y\}$;
- для простых графов, поддерживающих петли, — $E \subseteq V^2 \times \mathbb{R}$.

На рис. 14.3 показан пример взвешенного графа с петлями.

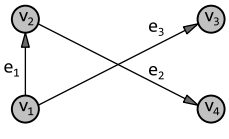


Рис. 14.2. Пример графа (ориентированного)

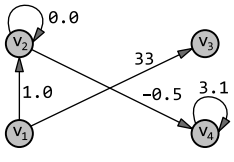


Рис. 14.3. Ориентированный взвешенный граф с петлями (крайние метки опущены для ясности)

14.1.1. Реализация графов

В предыдущем разделе графы были формально определены. Однако при переходе от теории к практике часто приходится сталкиваться с дополнительными проблемами, нюансами и преодолевать ограничения.

Математическая нотация проясняет, как следует обозначать графы на бумаге, но нужно также решить, например, как лучше хранить их в структуре данных.

Есть несколько вопросов, на которые следует ответить с учетом контекста: нужно ли хранить метки для вершин и ребер или достаточно присвоить индексы вершинам, используя натуральные числа, и перечислить ребра в соответствии с естественным порядком пар индексов?

Хранить вершины относительно просто (можно использовать списки или словарь, чтобы связать каждую вершину с ее меткой), однако есть еще один вопрос, выходящий за рамки любого контекста: как хранить ребра?

Этот вопрос не так прост, как может показаться: дело в том, что в какой-то момент может понадобиться проверить наличие ребра между двумя вершинами или, скажем,

найти все ребра, исходящие из определенной вершины. Если просто сохранить все ребра в одном списке, сортированном или несортированном, то, чтобы получить ответ на этот вопрос, придется просмотреть весь список.

Даже при использовании сортированного списка время доступа к элементам составит $O(\log(|E|))$, для совершения упомянутых в предыдущем абзаце операций и $O(|E|)$ — для перечисления всех ребер, входящих в вершину.

Неудивительно, что есть более удачное решение. Существует две основные стратегии хранения ребер графа:

- *списки смежности* — для каждой вершины v хранится список ребер (v, u), где v — исходящая вершина, а u — входящая вершина, другая вершина в G;
- *матрица смежности* — матрица $|V| \times |V|$, где ячейка (i, j) содержит вес ребра, исходящего из i-й вершины и входящего в j-ю вершину (в случае невзвешенных графов может использоваться значение `true/false` или `1/0`, определяющее наличие/отсутствие невзвешенного ребра между этими двумя вершинами).

Прежде чем рассматривать плюсы и минусы обеих стратегий, проиллюстрируем их на примере. Для графа на рис. 14.2 его представление в виде списка смежности может выглядеть как словарь, отображающий вершины в списки ребер (показано ниже):

```
1 -> [(1,2), (1,3)]
2 -> [(2,4)]
3 -> []
4 -> []
```

или в виде матрицы смежности:

	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	0	0	0	1
V_3	0	0	0	0
V_4	0	0	0	0

Как видите, представления очень разные. Что сразу бросается в глаза: в матрице смежности большинство ячеек заполнено нулями. Это связано с тем, что граф на рис. 14.2 имеет лишь несколько ребер из множества возможных.

Известно, что ребра определяются парами вершин, в простом графе максимальное количество ребер равно $O(|V|^2)$. А какое минимальное число их может быть?

Какое угодно; граф может (гипотетически) вообще не иметь ребер. Однако связный граф должен иметь не менее $|V| - 1$ ребер.

Имея это в виду, дадим определение, которое пригодится далее в этом разделе: граф $G = (V, E)$ называется *разреженным*, если $|E| = O(|V|)$; граф G называется *плотным*, если $|E| = O(|V|^2)$.

Другими словами, количество ребер в разреженном графе сравнимо с количеством вершин, поэтому они слабо связаны друг с другом, тогда как в плотных графах каждая вершина связана с большинством других вершин.

В табл. 14.1 перечислены плюсы и минусы двух различных представлений графов. Если говорить кратко, то представление в виде списка смежности лучше подходит для разреженных графов, потому что требует намного меньше памяти, а также потому, что для разреженных графов $|E| \approx |V|$ и, соответственно, большинство операций могут выполняться эффективно.

Таблица 14.1. Плюсы и минусы представлений графов

Операция	Список смежности	Матрица смежности
Вставка ребра	$O(1)$	$O(1)$
Удаление ребра	$O(\|V\|)$	$O(1)$
Список исходящих ребер	$O(\|E\|)$	$O(\|V\|)$
Список входящих ребер	$O(\|E\|)$	$O(\|V\|)$
Необходимый объем памяти	$O(\|E\| + \|V\|)$	$O(\|V\|^2)$
Вставка вершины	$O(1)$	$O(\|V\|)^*$
Удаление вершины	$O(\|V\| + \|E\|)^{**}$	$O(\|V\|)^*$

* Оптимистичная оценка, предполагающая, что матрица смежности может изменяться динамически. В противном случае более верной оценкой будет $O(|V|^2)$.

** Нужно проверить все ребра, чтобы удалить те, для которых удаляемая вершина была входящей.

Для плотных графов, наоборот, из-за того, что число ребер близко к максимально возможному, представление матрицы смежности оказывается компактнее и эффективнее. Более того, подобное представление позволяет продуктивнее реализовать некоторые алгоритмы, такие как поиск связанных компонент или транзитивных замыканий.

В общем случае, когда нельзя делать никаких предположений о графе и контекст не требует иного, предпочтительнее представление в виде списка смежности, потому что оно более гибкое и упрощает добавление новых вершин.

14.1.2. Графы как алгебраические типы

Есть еще один аспект представления графа, не зависящий от способа хранения ребер: непротиворечивость.

Справедливости ради следует отметить, что непоследовательность гораздо более вероятна при представлении в виде списка смежности, но даже при использовании матрицы смежности в определенных ситуациях она иногда возникает.

Проблема заключается в следующем. Абстрагируемся от формы представления и рассмотрим следующий граф: $G = ([1, 2], [(1, 2), (1, 3), (2, 2)])$.

Граф G имеет две вершины [1, 2], но в нем есть ребро с входящей вершиной 3. Такое может произойти по разным причинам: например, небрежность при удалении вершины 3 или ошибка при добавлении ребер.

Кроме того, в графе есть петля, ребро (2, 2). А что, если G должен быть простым графом без петель?

Конечно, в методы класса `Graph` можно добавить проверки, предотвращающие такие ситуации, но сама структура данных не может гарантировать отсутствие подобных ошибок.

Чтобы преодолеть эти ограничения, можно определить графы как алгебраический тип[1]; тогда понятие графа определится как одно из следующих:

- пустой граф;
- граф-синглтон, имеющий одну вершину без ребер;
- соединение между двумя графами G и G': определяются одно или несколько ребер, исходящие вершины которых находятся в G, а входящие — в G';
- объединение двух графов $G = (V, E)$ и $G' = (V', E')$, когда просто вычисляется объединение множества вершин и ребер, чтобы получить $G'' = (V \cup V', E \cup E')$.

Такое определение предотвращает появление несоответствий, упоминавшихся выше, гарантирует невозможность получить искаженные графы, позволяет формально определять алгоритмы как преобразования в графах и математически доказывать их корректность.

Попытка представить графы в виде алгебраического типа — полезное упражнение, которое поможет вам глубже понять эту структуру данных. В то же время следует признать, что, учитывая накладные расходы на эти операции, практическое их использование ограничено и в большей мере подходит для функциональных языков, поддерживающих сопоставление с типом, таких как Scala, Haskell или Clojure[2].

14.1.3. Псевдокод

В завершение обсуждения в листинге 14.1 приводится определение класса, который мы будем использовать для представления графов в этой книге. Он использует списки смежности и моделирует ребра и вершины как классы, что позволит

---

[1] Алгебраический тип данных — это особый вид составного типа, образованный путем объединения других типов, обычно с определением «по индукции», с одним или несколькими базовыми типами и операторами для их объединения. Следующая презентация прекрасно объясняет, как их определить в C++ и в чем их преимущества: https://www.youtube.com/watch?v=ojZbFIQSdl8.

[2] Пример на Haskell см.: *Mokhov A.* Algebraic graphs with class (functional pearl). ACM SIGPLAN Notices. — Vol. 52. — No. 10. — ACM, 2017. Реализацию на Scala (была доступна на момент написания книги) можно найти здесь: https://github.com/algebraic-graphs/scala.

реализовать различные типы графов, изменяя детали этих моделей (например, подавить поддержку взвешенных ребер).

Листинг 14.1. Класс Graph

```
class Vertex
  #type string
  label

class Edge
  #type Vertex
  source
  #type Vertex
  dest
  #type double
  weight
  #type string
  label

class Graph
  #type List[Vertices]
  vertices
  #type HashTable[Vertex->List[Edge]]
  adjacencyList

  function Graph()
    adjacencyList ← new HashTable()

  function addVertex(v)
    throw-if v in vertices
    vertices.insert(v)
    adjacencyList[v] ← []

  function addEdge(v, u, weight=0, label="")
    throw-if not (v in vertices and u in vertices)
    if areAdjacent(v, u) then
      removeEdge(v, u)
    adjacencyList[v].insert(new Edge(v, u, weight, label))

  function areAdjacent(v, u)
    throw-if not (v in vertices and u in vertices)
    for e in adjacencyList[v] do
      if e.dest == u then
        return true
    return false
```

Добавляя новую вершину при условии, что она не является дубликатом, сначала нужно добавить ее в список вершин

А также стоит инициализировать список смежности для новой вершины (это упростит нам жизнь позже!)

Проверяем, являются ли вершины смежными, то есть существует ли ребро от v до u

Если да, то сначала нужно удалить старое ребро (этот метод опущен, но его можно получить из areAdjacent)

В список смежности исходящей вершины добавляем новое ребро, созданное на основе аргументов

Итерации по всем ребрам в списке смежности исходящей вершины

Если найдено ребро, входящая вершина которого совпадает с u, можно вернуть true. Если такие ребра не найдены, то вершины не являются смежными

Конкретную реализацию на Java можно найти в репозитории книги на GitHub[1], а на JavaScript — в библиотеке JsGraphs; последняя также включает реализации алгоритмов, описываемых в следующих разделах.

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#graph.

14.2. СВОЙСТВА ГРАФА

Как уже упоминалось, графы очень похожи на деревья. И те и другие состоят из сущностей (вершин), связанных отношениями (ребрами), есть только пара отличий:

- в деревьях вершины обычно называют узлами;
- в деревьях ребра не имеют явного представления. Поскольку любое ребро может идти только от родителя к потомкам, чаще говорят об отношениях «родитель — потомок», но не о ребрах. Кроме того, по этой причине деревья неявно представлены списками смежности.

Вдобавок к перечисленному, у деревьев есть и другие отличительные признаки, делающие их ограниченным подмножеством целого множества графов. В частности, любое дерево является простым, неориентированным, связным и ациклическим графом.

В предыдущем разделе показано, что подразумевается под словами *«простой граф»*. На самом деле дерево не может иметь несколько ребер между двумя узлами и не может иметь петли; допускается только одно ребро между узлом и каждым из его дочерних элементов.

Теперь посмотрим, что означают остальные три свойства.

14.2.1. Неориентированный

Как упоминалось в разделе 14.1, граф называется ориентированным, если по всем его ребрам можно пройти только в одном направлении, от исходящей вершины (первой из соединяемых ребром) к входящей.

В неориентированных графах, напротив, по ребрам можно проходить в обоих направлениях. Разница показана на рис. 14.4.

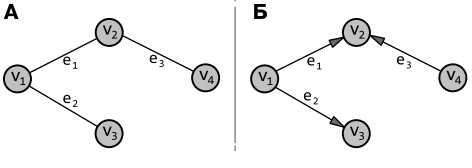


Рис. 14.4. Сравнение неориентированного *(А)* и ориентированного *(Б)* графов, имеющих одно и то же множество вершин. Эти два графа неэквивалентны; в частности, последний нельзя преобразовать в эквивалентный неориентированный граф

Неориентированный граф легко представить в виде ориентированного графа, разложив каждое неориентированное ребро (u, v) на пару ориентированных ребер (u, v) и (v, u).

Обратное обычно неверно, и многие ориентированные графы нельзя преобразовать в неориентированные изоморфные аналоги[1].

[1] Любой ориентированный граф, в котором есть ребро (u, v), но нет обратного ребра (v, u), нельзя преобразовать в изоморфный неориентированный граф (имеется в виду неориентированный граф такой же формы, то есть имеющий тот же набор вершин, соединенных таким же образом).

Поэтому, если контекст приложения не предполагает иного, то выбор представления в виде ориентированных графов является наименее ограничивающим.

Стоит также отметить, что, если используется представление в виде матрицы смежности, то неориентированные графы можно представить с использованием только половины матрицы, потому что их матрица смежности: $A[u, v] = A[v, u]$ всегда симметрична для каждой пары вершин (u, v).

14.2.2. Связный

Граф называется *связным*, если для любой пары его вершин (u, v) существует последовательность вершин $u, (w_1 ... w_k), v$, где $k \geq 0$, такая, что любые две смежные вершины в последовательности соединены ребром.

Для *неориентированных* графов это означает, что в связном графе из любой вершины можно достичь всех остальных вершин, а для *ориентированных* графов — что каждая вершина имеет по крайней мере одно входящее или исходящее ребро.

Для любых графов это означает, что количество ребер не меньше $|V| - 1$.

На рис. 14.5 показано несколько примеров связных и несвязных неориентированных графов. Понятие связности лучше подходит для неориентированных графов, тогда как для описания ориентированных графов вместо этого используется понятие *сильно связанных компонент*, как показано на рис. 14.6.

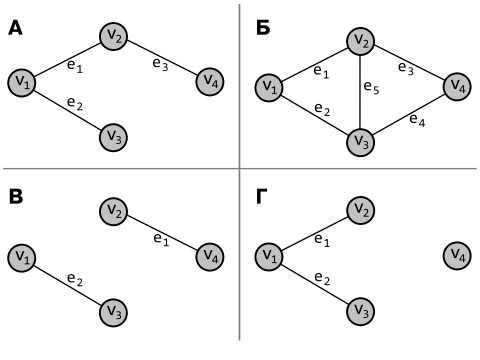


Рис. 14.5. Связные графы (*А*, *Б*) и несвязные (*В*, *Г*)

В *сильно связанной компоненте* (Strongly Connect Component, SCC) каждая вершина достижима из любой другой вершины; следовательно, сильно связанные компоненты должны иметь циклы (подраздел 14.2.3).

Понятие сильно связанной компоненты особенно важно. Оно позволяет определить граф сильно связанных компонент, который будет значительно меньше исходного графа, и применять множество алгоритмов к этому графу. Предварительно изучив граф на высоком уровне, а затем (возможно) изучив взаимосвязи внутри каждого SCC, можно получить большой прирост скорости.

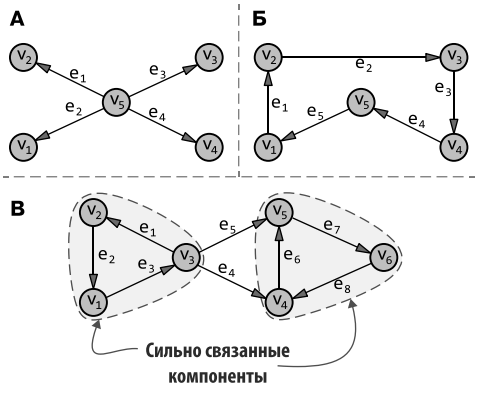


Рис. 14.6. Только граф *Б* является сильно связанным, а граф *В* имеет две сильно связанных компоненты

Дерево обычно рассматривается как неориентированный граф, по крайней мере в теории графов; но в конкретных реализациях каждый узел обычно хранит ссылки на своих потомков и лишь иногда — на своего родителя. Если ссылки на родителей не хранятся в дочерних узлах, то по ребру нельзя пройти в направлении от потомка к родителю, что фактически делает его ориентированным ребром.

14.2.3. Ациклический

Цикл в графе — это непустая последовательность ребер (u, v_1), (v_1, v_2)... (v_k, u), которая начинается и заканчивается в одной и той же вершине.

Ациклический граф (рис. 14.7) — это граф, не имеющий циклов.

Оба вида графов, ориентированные и неориентированные, могут иметь циклы, поэтому существует подмножество ациклических графов, представляющее особый интерес: *ориентированные ациклические графы* (Directed Acyclic Graph, DAG).

Ориентированный ациклический граф (DAG) имеет несколько интересных свойств: в графе должна быть хотя бы одна вершина, не имеющая входящего ребра (иначе возник бы цикл); более того, поскольку ациклический граф не имеет циклов, множество ребер определяет частичный порядок его вершин.

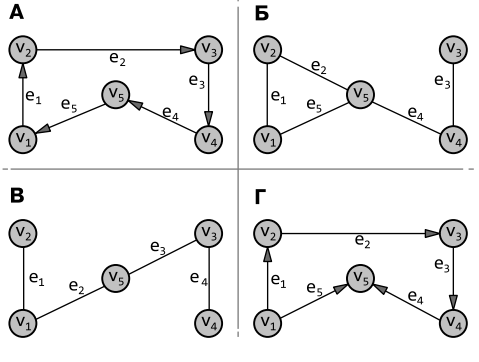


Рис. 14.7. Циклические графы (*А* — ориентированные, *Б* — неориентированные) в сравнении с ациклическими графами (*В* и *Г*)

Для ориентированного ациклического графа $G = (V, E)$ на самом деле частичный порядок — это отношение $\leq$, такое, что для любой пары вершин (u, v) будет выполняться ровно одно из трех условий:

- $u \leq v$, если существует путь, состоящий из любого числа ребер, начинающийся в u и заканчивающийся в v;
- $v \leq u$, если существует путь, состоящий из любого числа ребер, начинающийся в v и заканчивающийся в u;
- $u <> v$; они несопоставимы, потому что нет никакого пути из u в v или наоборот.

На рис. 14.8 показаны пара универсальных DAG и цепочка; последняя — единственный вид DAG, определяющий общий порядок своих узлов.

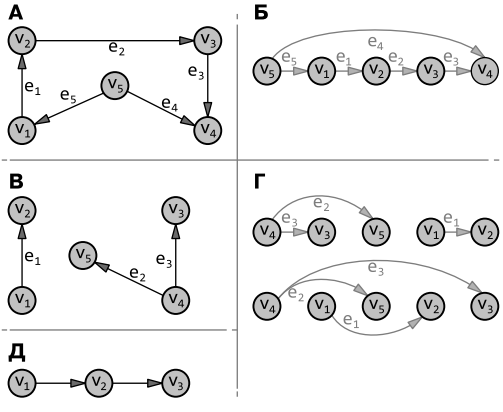


Рис. 14.8. Несколько примеров DAG. *А*. Связный граф DAG. *Б*. Топологическая сортировка графа; обратите внимание, что результатом топологической сортировки является простая последовательность вершин, а ребра не учитываются. Тем не менее здесь показано, что они направлены слева направо, только если вершины в топологической сортировке перечислены горизонтально (слева направо). *В*. Несвязный граф DAG. *Г*. Пара возможных примеров топологической сортировки для графа *В*. *Д*. Цепной граф: для этого графа DAG существует только одна возможная топологическая сортировка

Свойство частичной упорядоченности DAG обеспечивает возможность *топологической сортировки* — упорядочения вершин таким образом, что для любого ребра в графе его начальная точка находится перед конечной. Обычно каждый граф имеет несколько эквивалентных топологических порядков. Все цепные графы, подобные тому, что изображен на рис. 14.8, *Д*, всегда имеют единственную уникальную топологическую сортировку, но нецепной граф тоже может иметь единственную топологическую сортировку (см. рис. 14.8, *А*, *Б*).

Ориентированные ациклические графы и топологическое упорядочение имеют множество фундаментальных вариантов применения в информатике, от планирования (на всех уровнях) до разрешения зависимостей символов в компоновщиках.

14.3. ОБХОД ГРАФА: АЛГОРИТМЫ BFS И DFS

Для выполнения поиска в графе, а также для применения многих других алгоритмов необходимо выполнить обход графа по его ребрам.

Как и в случае с деревьями, существует множество возможных способов обхода графа, в зависимости от порядка перебора исходящих ребер. В деревьях, напротив, всегда ясно, с чего начать обход: с корня. Начиная с корня, всегда можно обойти дерево и добраться до всех его узлов. Однако в графах нет особой вершины, такой как корень дерева, и вообще, при определенном выборе вершины в качестве отправной точки может не оказаться той последовательности ребер, которая позволит посетить все вершины.

В этом разделе мы сосредоточимся на простых ориентированных графах, не делая никаких других допущений. Не будем ограничиваться сильно связанными графами и в общем случае будем считать, что у нас нет никакой дополнительной информации, которая помогла бы выбрать вершину для начала поиска. Следовательно, не будет гарантии, что за один обход мы охватим все вершины графа. Наоборот, обычно требуется несколько «попыток» из разных начальных точек, чтобы посетить все вершины графа: это будет показано при обсуждении алгоритма DFS.

Для начала сосредоточимся на конкретном варианте использования: будем рассматривать начальную точку как заданную (выбранную извне, алгоритму ничего о ней не известно) и выполним обход графа из нее. Основываясь на этих предположениях, обсудим две наиболее распространенные стратегии обхода графов.

14.3.1. Оптимизация маршрутов доставки

Пришло время вернуться к задаче, с которой началась эта глава: у нас есть один заказ, который нужно доставить из исходящей точки, склада или фабрики, где хранятся товары, во входящую, соответствующую адресу клиента.

Гипотеза о том, что доставка заказов осуществляется по одному, является очевидным упрощением. В общем случае такое обслуживание заказов было бы слишком дорого, и компании, занимающиеся доставкой, стараются собирать заказы с одного склада в близлежащие пункты назначения, чтобы распределить затраты (топливо, рабочее время сотрудников и т. д.) между несколькими заказами.

Однако поиск наилучшего пути, проходящего через несколько точек, является вычислительно сложной задачей[1]; и наоборот, поиск наилучшего пути для случая с одним пунктом отправления и одним пунктом назначения можно выполнить эффективно.

В этом и нескольких следующих разделах мы постепенно разработаем общее решение этой задачи, начав с максимально упрощенного сценария и удаляя упрощения шаг за шагом, по мере знакомства со все более сложными алгоритмами решения.

---

[1] Слышали о «задаче коммивояжера»? Это одна из наиболее изученных сложных задач.

Итак, начнем обсуждение с того, что обдумаем нашу цель: какой путь считать «лучшим»? Можно принять решение, например, что лучший путь — это кратчайший путь, но также можно предпочесть определение «самый быстрый» или «самый дешевый», в зависимости от требований.

На данный момент предположим, что нам нужен кратчайший путь. Если упростить сценарий и проигнорировать такие факторы, как интенсивность движения, дорожные условия и ограничения скорости, то можно условиться, что чем короче расстояние, тем быстрее его можно преодолеть.

Но даже этот упрощенный сценарий можно сделать еще проще. На рис. 14.9, например, показана часть города, где дороги образуют регулярную сетку. Это обычная ситуация для многих городов США, в то время как в других странах мира она не так распространена. В Европе многие городские центры имеют план, изначально разработанный в Средние века или даже раньше, и дороги там гораздо менее регулярно упорядочены.

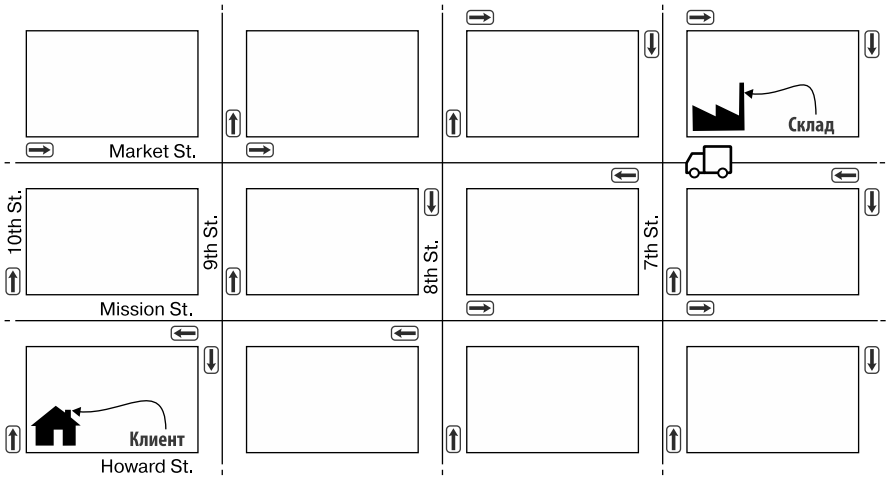


Рис. 14.9. Пример карты: часть центра Сан-Франциско, где кварталы образуют регулярную сетку

На данный момент можно ограничиться идеальной ситуацией. Почему? Потому что это облегчит нам работу. Если все блоки одинаковы и их можно аппроксимировать квадратами (или прямоугольниками, близкими к квадратам), то не придется беспокоиться о реальных расстояниях; можно просто подсчитать количество пройденных блоков, чтобы вычислить длину маршрута. Такая задача решалась бы тривиально просто... если бы не улицы с односторонним движением! Как только будет разработано минимальное жизнеспособное решение для этого упрощенного сценария, мы сможем подумать о его дальнейшем развитии, чтобы охватить более реалистичные ситуации.

На рис. 14.10 показан граф, который можно построить по карте, изображенной на рис. 14.9. На ней добавлены вершины для каждого перекрестка. И я рекомендую учитывать, что ребро, идущее от вершины v к вершине u, означает, что от перекрестка, представленного вершиной v, к перекрестку, представленному вершиной u, есть дорога, по которой можно пройти в этом направлении (но не обязательно в обратном направлении, от u к v).

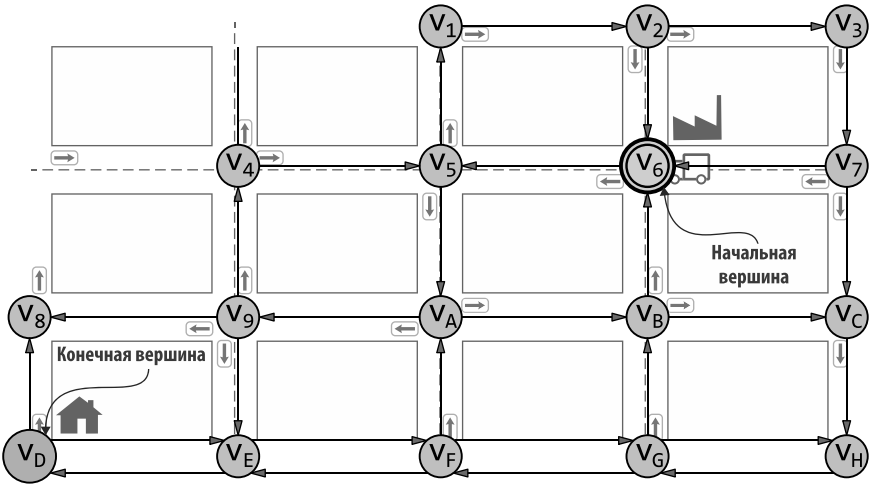


Рис. 14.10. Построение графа по карте на рис. 14.8. Мы добавили вершины для всех перекрестков (некоторые вершины, однако, опущены, чтобы не загромождать схему) и ребра, представляющие улицы с односторонним движением (улицы с двусторонним движением моделируются парами ребер)

Если бы по всем улицам можно было проехать в обоих направлениях, мы бы просто поехали на запад по Market St. от склада до пересечения с 10th St., а затем на юг по 10th St. до места назначения.

Однако, учитывая дорожные знаки (их можно видеть на рис. 14.9), это невозможно, и нужно выбирать объезд. В следующем подразделе мы увидим, как это сделать.

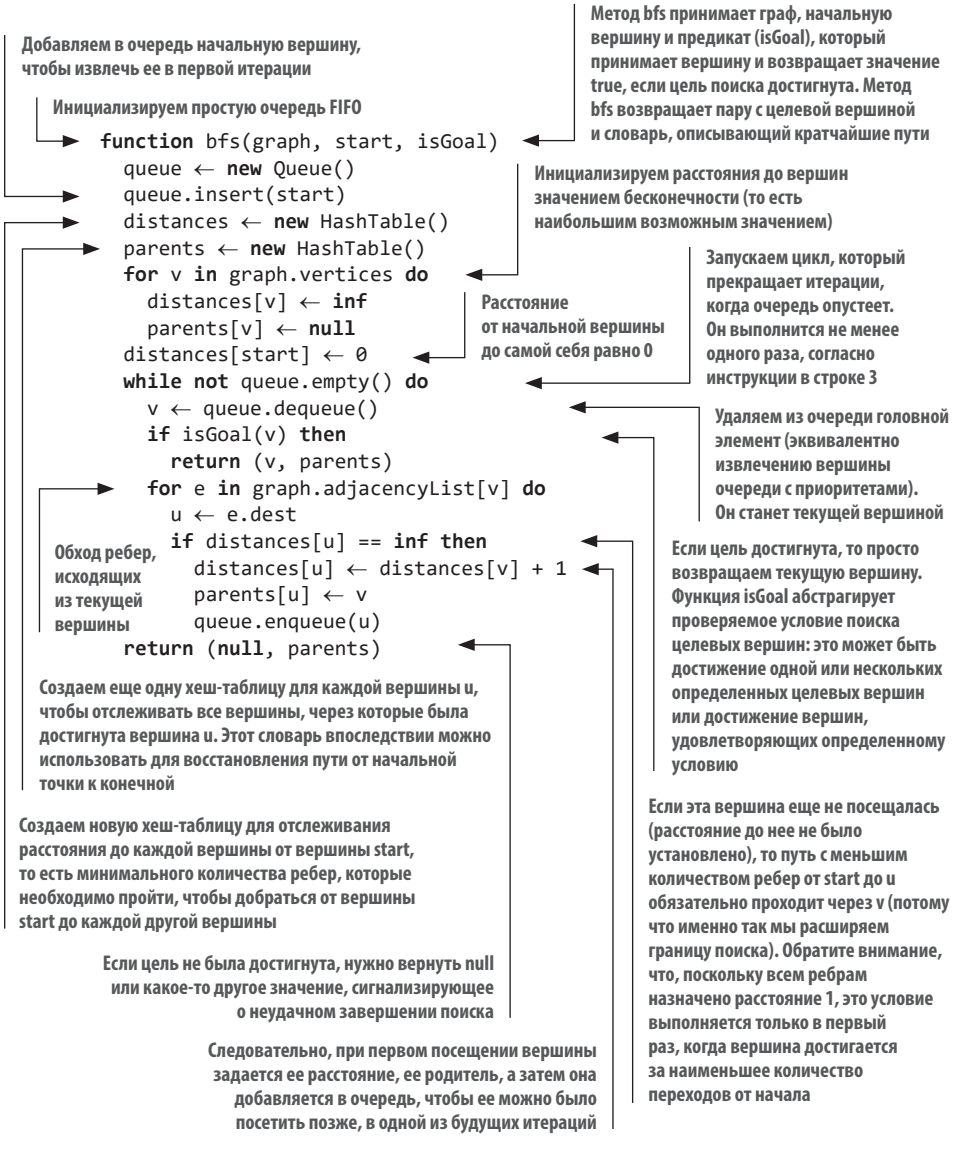
14.3.2. Поиск в ширину

В листинге 14.2 показан псевдокод, реализующий алгоритм *поиска в ширину* (Breadth First Search, BFS), целью которого, как следует из названия, является максимально возможное расширение поиска, сохранение периметра уже посещенных вершин и расширение этого периметра до соседних вершин. Процесс показан на рис. 14.11, где для демонстрации первых нескольких шагов алгоритма используется граф, изображенный на рис. 14.6, *B*.

По сути, этот алгоритм хранит очередь вершин, которые будут посещаться следующими, так называемую *границу*. Вершины хранятся в определенном порядке и соответственно обрабатываются от самой близкой к источнику до самой дальней. Роль

контейнера для отслеживания вершин на границе могла бы играть очередь с приоритетами, но это было бы излишним. Поскольку метрика, которая используется для вычисления расстояния до каждой вершины от источника, — это просто количество пройденных ребер, алгоритм естественным образом обнаруживает вершины в том же порядке, в каком они должны обрабатываться. Кроме того, расстояние до каждой вершины можно вычислить в момент ее посещения (когда каждая вершина посещается в первый раз при обходе другого списка смежности вершин).

Листинг 14.2. Метод bfs



Таким образом, помимо инициализации в строках 2–9, ядром алгоритма BFS является цикл в строке 10: вершина v удаляется из очереди и обрабатывается. После (необязательной) проверки достижения цели поиска (в этом случае уже можно вернуть v) алгоритм начинает исследовать окрестности v (список смежности, то есть все ребра, исходящие из нее). Если обнаружится вершина u, которая еще не посещалась, то расстояние от нее до начальной вершины будет равно расстоянию до $v + 1$; поэтому можно установить расстояние до u и добавить u в конец очереди.

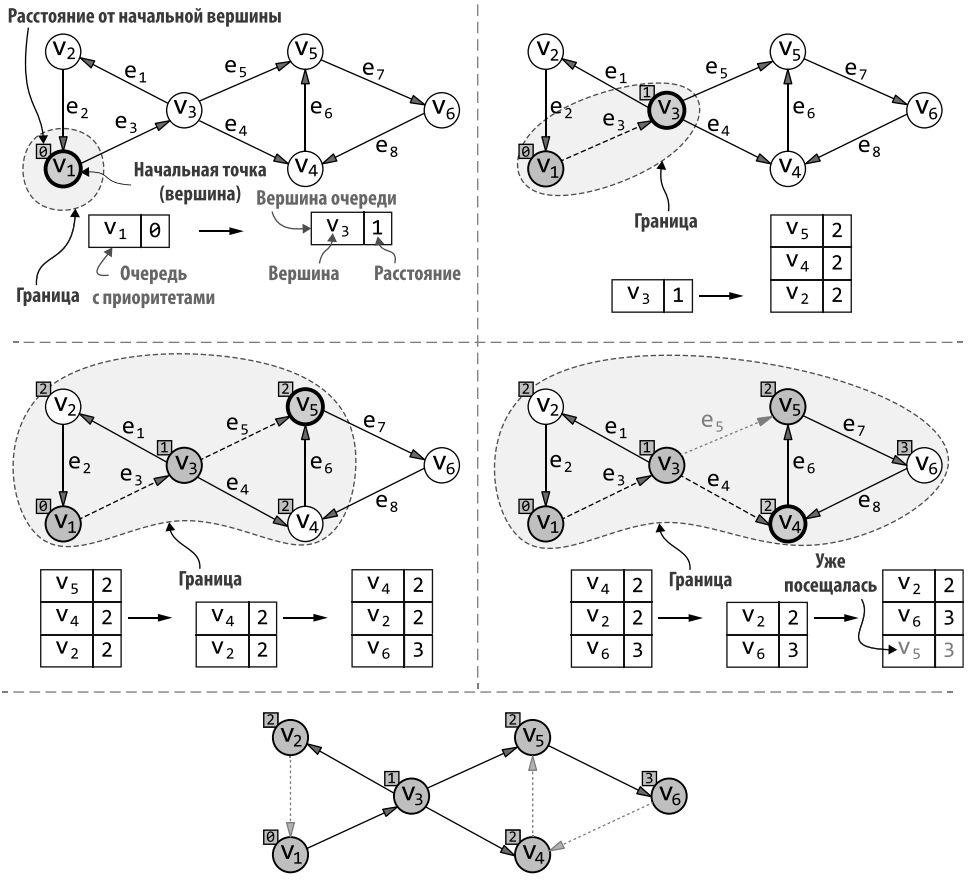


Рис. 14.11. BFS в действии. Этот пример показывает выполнение алгоритма с одной начальной вершиной, он вычисляет расстояние до всех вершин в графе. Алгоритм явно поддерживает очередь с вершинами на границе, которые будут исследоваться далее. Вершины естественно упорядочены по их расстоянию от источника, поэтому можно применять обычную очередь, хотя концептуально алгоритм ведет себя так, как если бы он использовал очередь с приоритетами. Пунктирные ребра на первых четырех схемах — это ребра, пройденные от источника к текущей вершине (той, что в начале очереди вверху слева), а ребра, изображенные точками на нижней схеме, — это ребра, которые не были включены в кратчайшие пути от начальной вершины до остальной части графа

На каком основании можно уверенно заявлять, что u будет посещена в правильном порядке, то есть после всех вершин, находящихся ближе к начальной, но перед находящимися дальше?

Это легко доказать индукцией по расстоянию между вершинами. Базой индукции является начальный случай. Добавляем начальную вершину с расстоянием 0. Эта вершина — единственная на расстоянии 0, поэтому любая другая вершина, добавленная позже, не может быть ближе к начальной.

Выполняя индуктивный переход, мы предполагаем, что наша гипотеза верна для всех вершин на расстоянии $d - 1$, и хотим доказать ее для вершины v на расстоянии d. Следовательно, v извлекается из очереди после всех вершин на расстоянии $d - 1$ и перед всеми вершинами на расстоянии $d + 1$. Отсюда можно заключить, что ни одна из вершин в очереди не может иметь расстояние $d + 2$, так как ни одна вершина на расстоянии $d + 1$ еще не была посещена, а мы добавляем вершины в очередь только при просмотре списка смежности каждой посещенной вершины (то есть расстояние добавленной вершины может увеличиться только на одну единицу по отношению к ее родителю). В свою очередь, это гарантирует, что u будет посещена раньше любых вершин на расстоянии $d + 2$ (или дальше) от начальной.

Кроме того, согласно индуктивной гипотезе можно утверждать, что все вершины на расстоянии $d - 1$ были посещены до v, поэтому все вершины на расстоянии d уже находятся в очереди и, следовательно, будут посещены до u.

Это свойство позволяет использовать простую очередь вместо очереди с приоритетами. Неплохо, учитывая, что время добавления и удаления элемента для первой в худшем случае равно $O(1)$ (тогда как для куч, например, эти операции выполняются за $O(\log(n))$), что обеспечивает линейное время работы BFS. В худшем случае оно будет линейно зависеть от наибольшего числа вершин и ребер, то есть $O(|V| + |E|)$. Для связных графов эту оценку можно упростить до $O(|E|)$.

Выше упоминалось, что проверка достижения цели поиска (строка 12) и вся концепция целевой вершины необязательны. Алгоритм BFS также можно использовать для простого вычисления пути и расстояния от одной вершины до всех других вершин графа[1]. Стоит отметить, что в настоящее время ни один из известных алгоритмов поиска кратчайшего пути к одной вершине не является более эффективным, чем BFS. Другими словами, вычисление кратчайшего пути «от одной вершины до другой» и кратчайшего пути «от одной вершины до всех целевых вершин» асимптотически одинаково затратны[2].

---

[1] Либо удалив проверку достижения цели, либо передав в аргументе isGoal функцию, которая всегда возвращает false.

[2] Просто для ясности: этот вариант отличается от вычисления оптимального пути через несколько или все вершины. В случае «от одной вершины до всех целевых вершин» вычисляются только оптимальные пути от начальной вершины до каждой другой вершины по отдельности.

Если бы нас интересовали все расстояния между всеми парами вершин, можно было бы использовать более эффективные решения, чем запуск BFS $|V|$ раз (но это выходит за рамки нашего текущего обсуждения).

14.3.3. Реконструкция пути к цели

Довольно часто помимо вычисления минимального расстояния до определенной вершины также бывает нужно найти кратчайший путь к этой вершине, то есть перечислить, какие ребра и в какой последовательности следует пройти, чтобы добраться от начальной вершины до конечной.

Как показано в нижней части рис. 14.11, есть возможность получить дерево, учитывая отношение «родитель — потомок» между посещенной и вновь найденной вершинами: каждая вершина u имеет ровно одного родителя, вершину v, которая уже посещалась к моменту, когда u была обнаружена (строка 15 в листинге 14.2).

Это дерево содержит все кратчайшие пути от начальной вершины к другим вершинам. Но результат алгоритма BFS — простой словарь, как же можно реконструировать эти пути, имея возвращаемый контейнер `parents`?

Требуемый алгоритм реконструкции показан в листинге 14.3. Она начинается с целевой вершины (последней вершины в пути), и весь путь восстанавливается в обратном направлении, когда на каждом шаге отыскивается родитель текущей вершины. В простых графах существует только одно ребро между упорядоченной парой вершин, поэтому, если известно, что мы переместились из вершины v в вершину u, пройденное ребро становится неявным. В мультиграфах необходимо отслеживать фактическое ребро, выбранное на каждом шаге.

Листинг 14.3. Метод реконструкции пути

```
Проверяем, достижима ли целевая вершина
из начальной вершины, если нет, то возвращаем null

    Метод reconstructPath принимает словарь родителей, созданный
    методом bfs, и целевую вершину. Возвращает путь от начальной
    вершины (неявный, определяемый родителями) до целевой
  function reconstructPath(parents, destination)
    if parents[destination] == null then
      return null
    current ← destination
    path ← [destination]
    while parents[current] != null do
      current ← parents[current]
      path.insert(current)
    return reverse(path)
```

Инициализируем возвращаемый путь как список вершин, которые будут посещены (в обратном порядке, начиная с целевой)

Выполняем итерации, пока не будет достигнута начальная вершина — первая, чей родитель будет равен null (найдется как минимум одна такая вершина; это гарантируется особенностями работы bfs)

Добавляем этого родителя в путь

По завершении цикла, прежде чем вернуть путь, его нужно перевернуть в обратном порядке, потому что сейчас список содержит вершины от конечной вершины к начальной

Перемещаемся на шаг назад, от текущей вершины к ее родителю, то есть к вершине, которая была посещена алгоритмом bfs в момент обнаружения текущей вершины

Код в листинге 14.3 предполагает, что словарь `parents` сформирован правильно, граф связный и существует путь от начальной вершины к конечной. Если все эти условия выполняются, то есть только два случая, когда текущая вершина может иметь пустую ссылку (`null`) на родителя: либо текущая вершина является конечной (и тогда это означает, что она недостижима из начальной вершины), либо текущая вершина является начальной.

Если выполняется условие `parent[destination]!=null`, то это означает, что целевая вершина была достигнута прохождением некоторого пути от начальной вершины (это обусловлено особенностями работы алгоритма BFS) и можно доказать по индукции, что должна существовать цепочка вершин между этими двумя вершинами.

Разберемся, как действует алгоритм, глядя на схему на рис. 14.11. Здесь `bfs`, используя v_1 в качестве начальной вершины, вернул следующий словарь `parents`:

$$[v_1 \to \textbf{null}, v_2 \to v_3, v_3 \to v_1, v_4 \to v_3, v_5 \to v_3, v_6 \to v_5]$$

Начав, например, с v_5, заметим, что `parent[v`$_5$`]==v`$_3$, а это значит, что нужно добавить v_3 в `path` после v_5, а затем посмотреть его родителя и т. д.

Непосредственно перед строкой 9 имеется список `path==[v`$_5$`, v`$_3$`, v`$_1$`]`, а после его переупорядочения получается искомая последовательность вершин.

14.3.4. Поиск в глубину

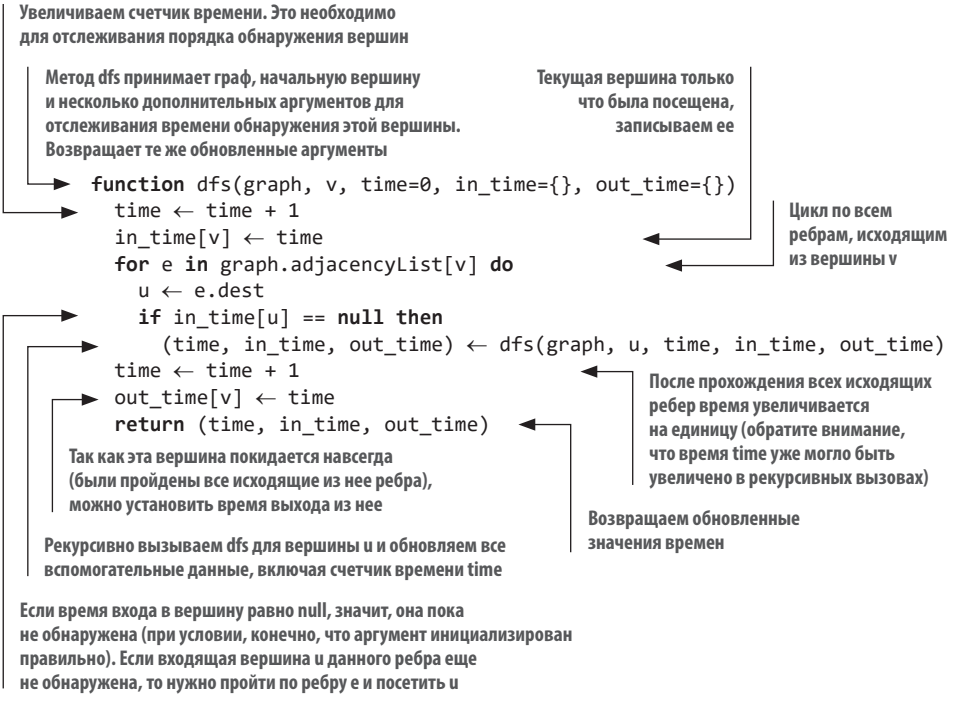
Алгоритм BFS, как мы видели, использует четкую стратегию обхода графа. Он начинается с ближайших к исходной точке вершин, а затем распространяется во все стороны, как волна, концентрическими кольцами: сначала охватывает все вершины на расстоянии 1, затем все вершины на расстоянии 2 и т. д.

Это весьма эффективная стратегия, когда нужно найти кратчайшие пути между вершинами графа, но она не является единственно возможной.

Другой подход заключается в обходе путей в глубину. Он напоминает поиск выхода из лабиринта: вы продолжаете идти как можно дальше от начальной вершины, выбирая направление на каждом перекрестке, пока не зайдете в тупик (в терминах графа: пока не доберетесь до вершины, не имеющей не пройденных прежде исходящих ребер). В этот момент вы возвращаетесь к последней точке бифуркации (предыдущей вершине, имеющей непройденные исходящие ребра) и выбираете другой путь.

Эта идея *поиска в глубину* (Depth First Search, DFS), по сути, противоположна стратегии BFS. Алгоритм DFS, представленный в листинге 14.4, нельзя использовать для поиска кратчайших путей, но он имеет множество важных вариантов применения в теории графов и на практике.

Листинг 14.4. Метод dfs обхода вершин (и их соседей)



На рис. 14.12 показан пример обхода алгоритмом DFS того же графа, что применялся для иллюстрации работы алгоритма BFS, и с использованием той же вершины v_1 в качестве отправной точки. Как видите, вершины посещаются в совершенно другом порядке (и это обусловлено не только условием выбора ребра, которое нужно пройти первым). Пожалуй, наиболее заметной деталью является использование стека вместо очереди для отслеживания следующих вершин, которые нужно посетить.

DFS, как и BFS, не дает гарантий, что за один прогон сумеет посетить все вершины графа. Это показано на рис. 14.13, где обход начинается не с v_1, а с v_4. Поскольку граф имеет две сильно связанные компоненты и первая из них недостижима из v_4, это неизбежно означает, что вершины с v_1 по v_3 не могут быть посещены в этом прогоне.

Чтобы завершить обход графа, нужно выполнить еще один прогон, начав с одной из оставшихся вершин, которые не посещались ранее: это показано на рис. 14.14, где новый обход начинается с вершины v_3 и охватывает три вершины, которые не были посещены при первом прогоне.

В завершение обсуждения обратимся к листингу 14.5. Тут мы найдем код, выполняющий полный обход графа с использованием алгоритма DFS и перезапусками.

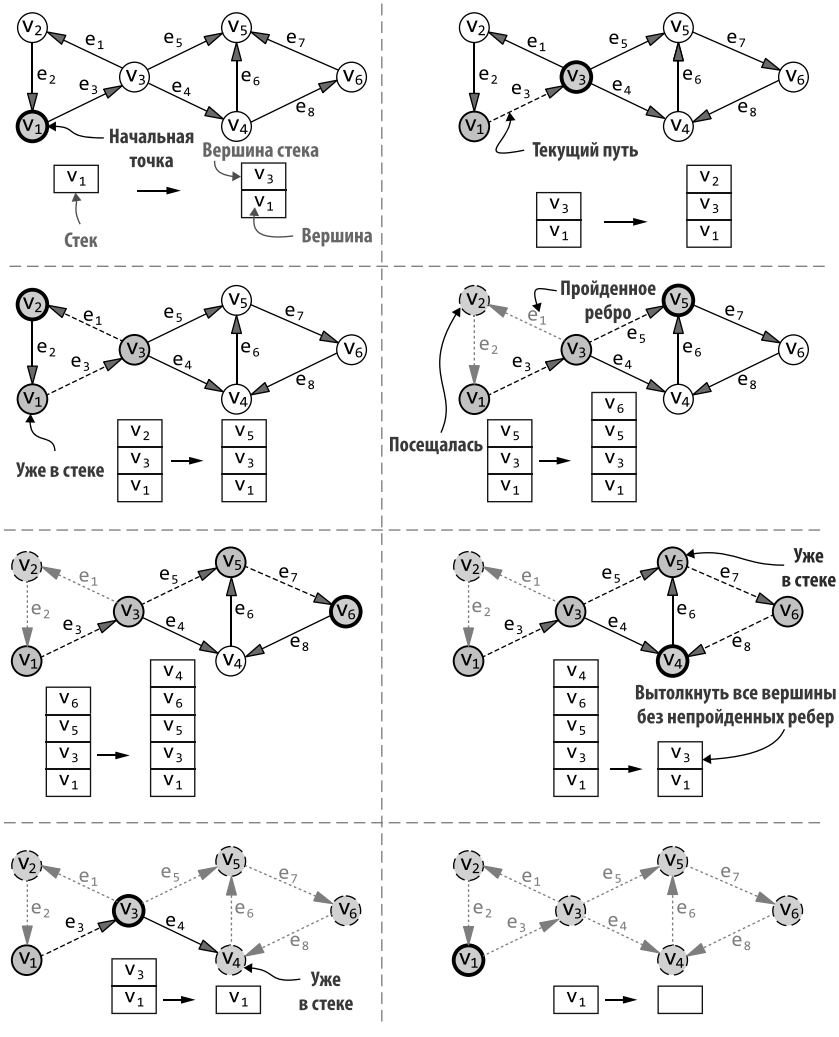


Рис. 14.12. DFS в действии. Алгоритм неявно (в рекурсивных реализациях, в противном случае — явно) поддерживает стек с вершинами, которые необходимо обойти следующими. Если стек хранится явно, то также необходимо запоминать, какие ребра были пройдены

Можно также предусмотреть передачу методу `dfs` функции обратного вызова, чтобы к ней можно было обращаться во время обхода для обработки каждой посещенной вершины. Эту функцию можно использовать для разных целей, от обновления графа (изменения меток вершин или любого другого связанного с ним атрибута) до выполнения определенных произвольных операций с графами. Нужно лишь предусмотреть в листингах 14.4 и 14.5 дополнительный параметр для передачи функции обратного вызова и вызвать ее в первой же инструкции в листинге 14.4.

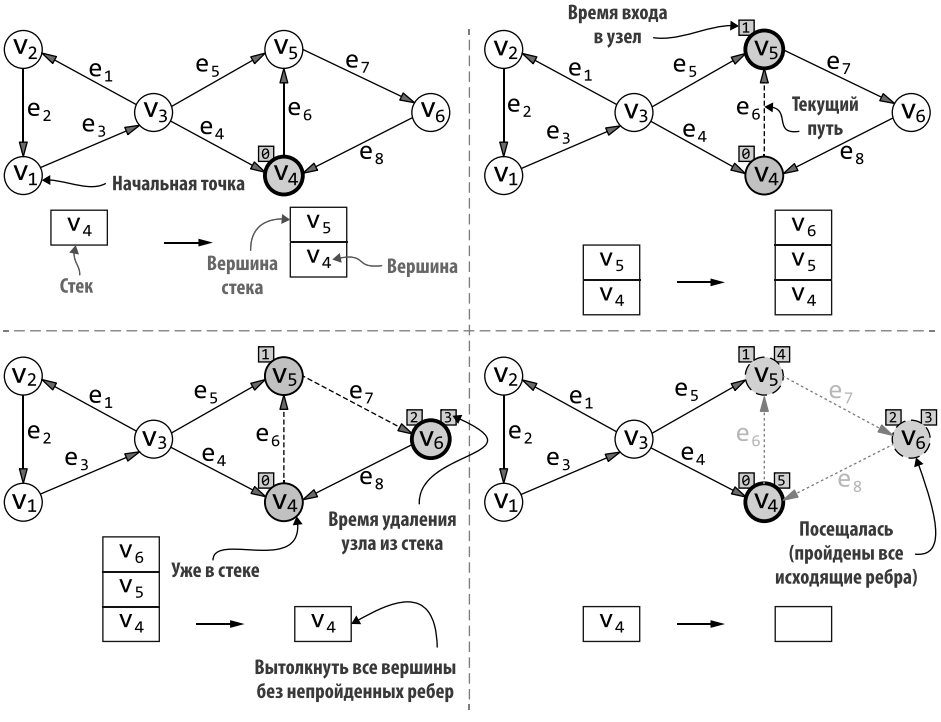


Рис. 14.13. DFS в действии. Выбор начальной точки играет важную роль в обходе графа; оба алгоритма, DFS и BFS, не смогут посетить все вершины за один прогон, например, если обход начинается с вершины v_4 (или, что важно, v_5 или v_6). На этом рисунке вершины также отмечены временем их обработки и временем удаления из стека. Эти значения актуальны для нескольких алгоритмов

Листинг 14.5. Метод dfs, выполняющий полный обход графа

```
Инициализируем счетчик времени. Это необходимо
для отслеживания порядка обнаружения вершин

Метод dfs принимает граф, выполняет обход всех его
вершин и возвращает два словаря с временами входа
и выхода для каждой вершины

function dfs(graph)
  time ← 0
  in_time ← null (∀ v ∈ graph)
  out_time ← null (∀ v ∈ graph)
  for v in graph.vertices do
    if in_time[v] == null then
      (time, in_time, out_time) ←
          dfs(graph, v, time, in_time, out_time)
  return (in_time, out_time)
```

Инициализируем время входа и выхода для каждой вершины значением null

Цикл по всем вершинам в графе

И в этом случае нужно вызвать dfs, чтобы выполнить обход v

После посещения всех вершин возвращаем время входа и выхода

На рис. 14.13 и 14.14 также можно видеть, что добавлены моменты времени, в которые происходит вход в вершины и выход из них (когда будет полностью пройден их список смежности). Эти значения являются фундаментальными для нескольких вариантов применения, таких как вычисление топологической сортировки (для DAG достаточно просто упорядочить вершины в обратном порядке по времени выхода), поиск циклов (если соседняя посещаемая в данный момент вершина имеет время входа, но не имеет времени выхода) или вычисление связанных компонент.

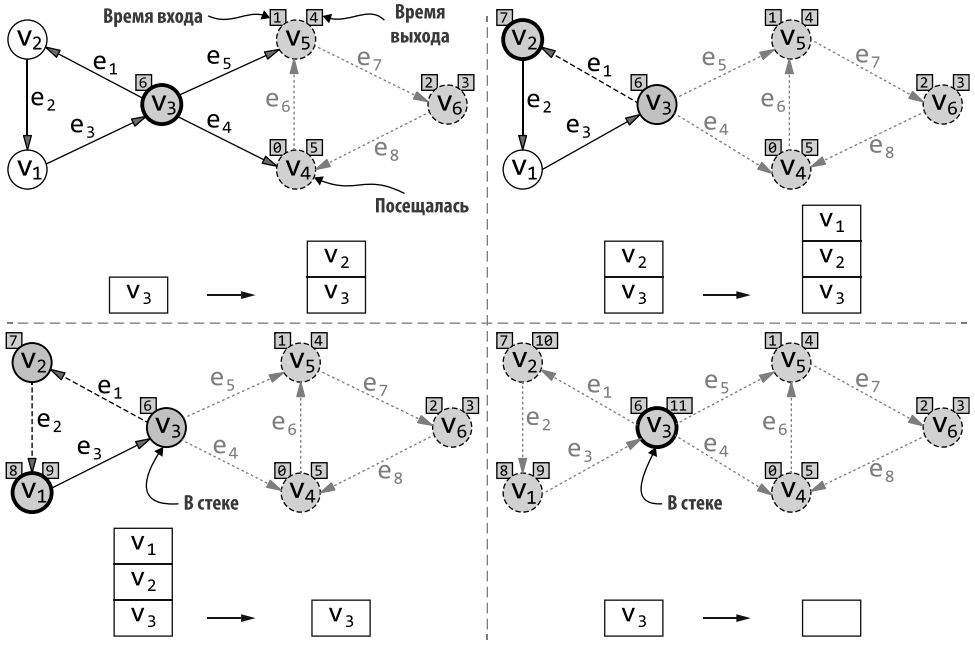


Рис. 14.14. DFS в действии. Вернувшись к примеру на рис. 14.13, можно продолжить обход и добраться до всех вершин, случайно выбирая одну из оставшихся и перезапуская алгоритм DFS для нее

Заключительное замечание о производительности: как и BFS, этот алгоритм тоже имеет линейное время работы $O(|V| + |E|)$ и требует $O(|V|)$ рекурсивных вызовов (или, что то же самое, $O(|V|)$ дополнительного пространства в стеке).

14.3.5. И снова о выборе между очередью и стеком

Рассматривая эти алгоритмы обхода, важно четко понимать, что их цели и контекст применения принципиально различны.

BFS используется, когда известна начальная вершина *S* и нужно найти кратчайший путь к другой определенной цели (конкретной вершине, всем вершинам, до которых можно добраться из *S*, или до достижения определенного условия).

DFS в основном используется, когда нужно обойти все вершины и не имеет значения, с какой начать. Этот алгоритм основан на глубоком понимании структуры графа и применим как основа для нескольких алгоритмов, включая поиск топологической сортировки для DAG или вычисление сильно связанных компонент ориентированного графа.

Интересная особенность обоих алгоритмов состоит в том, что их базовая версия, выполняющая только обход, может быть переписана как тот же шаблонный алгоритм, в котором вновь обнаруженные вершины добавляются в контейнер, а из него в каждой итерации извлекается следующий элемент. В BFS роль такого контейнера играет очередь, и вершины обрабатываются в порядке их обнаружения, а в DFS — стек (неявный в рекурсивной версии алгоритма), и каждый прогон будет пытаться пройти как можно дальше, прежде чем вернуться и посетить соседние вершины.

14.3.6. Лучший маршрут доставки посылки

Теперь, зная, как работает BFS и как реконструировать путь от начальной до целевой вершины, можно вернуться к нашему примеру и применить BFS к графу на рис. 14.10; результат показан на рис. 14.15.

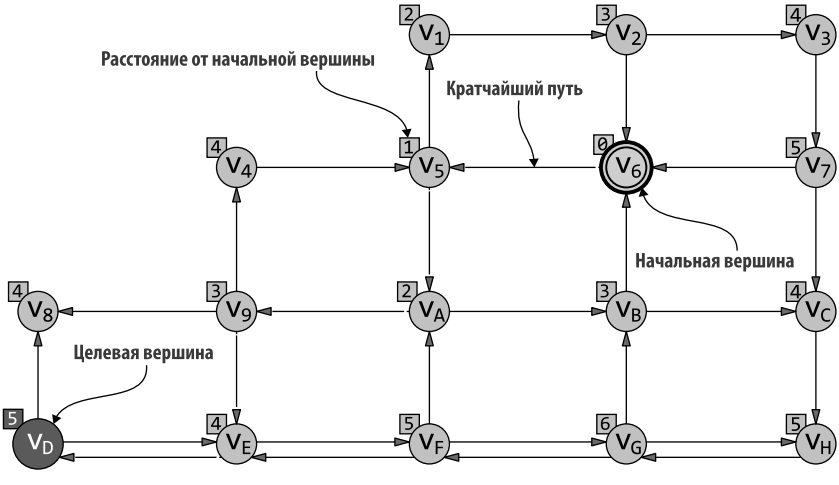


Рис. 14.15. Кратчайший путь к целевой вершине и кратчайшие расстояния до всех вершин, вычисленные с использованием BFS для графа на рис. 14.10. Кратчайший путь показан пунктирными стрелками, а расстояния указаны рядом с каждой вершиной

В этом случае кратчайший путь очевиден — это путь, ближайший к прямой между источником и пунктом назначения, а также путь с минимальным *манхэттенским расстоянием*[1].

[1] Манхэттенское расстояние, также известное как расстояние городских кварталов, — это сумма абсолютных разностей декартовых координат двух точек. Название связано с уличной планировкой Манхэттена, где большинство улиц образуют сетку, поэтому длина кратчайшего пути между двумя перекрестками равна сумме сторон квартала.

И все же после удаления ребра между вершинами v_9 и v_E результат должен полностью измениться. Попробуйте в качестве упражнения найти решение для этого модифицированного случая (вручную или написав свою версию BFS и запустив ее).

14.4. КРАТЧАЙШИЙ ПУТЬ ВО ВЗВЕШЕННЫХ ГРАФАХ: АЛГОРИТМ ДЕЙКСТРЫ

Упрощение сценария позволило нам использовать простой и быстрый алгоритм BFS поиска приблизительного кратчайшего пути для доставки заказов. Это упрощение подходит для современных городских центров, таких как центр Сан-Франциско, но оно неприменимо к более общим сценариям. Если понадобится оптимизировать доставку в пригороды Сан-Франциско или в другие города, где нет регулярной структуры дорог, то аппроксимация расстояний количеством пройденных кварталов уже не будет работать.

Если переместиться, например, из Сан-Франциско (или из Манхэттена) в центр Дублина, схема которого показана на рис. 14.16, то обнаружится, что улицы не имеют регулярной планировки, а кварталы могут сильно различаться по размеру и форме, поэтому нужно учитывать фактическое расстояние между каждой парой перекрестков, которое больше не будет их манхэттенским расстоянием.

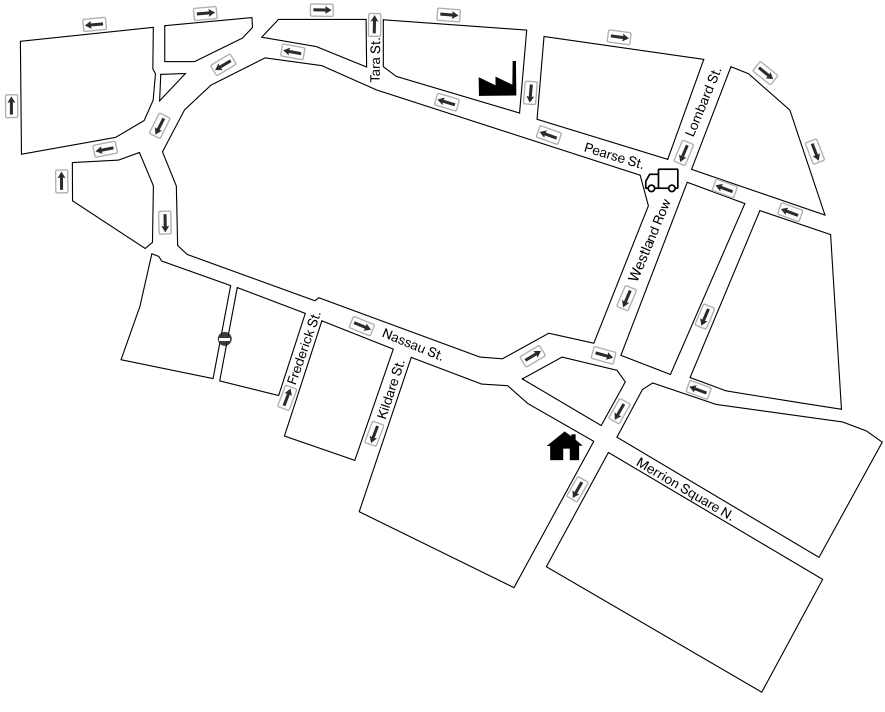


Рис. 14.16. Пример карты города (центр Дублина), где упрощения, использованные при реализации BFS для поиска кратчайших путей, более невозможны

14.4.1. Отличия от BFS

Подобно BFS, алгоритм Дейкстры принимает граф и начальную вершину (и необязательную целевую вершину) и вычисляет минимальное расстояние от начальной вершины до цели (или, что то же самое, с тем же асимптотическим временем выполнения, до всех остальных вершин). Однако, в отличие от BFS, в алгоритме Дейкстры расстояние между двумя вершинами задается весом ребра. Рассмотрим рис. 14.17, где показан ориентированный взвешенный граф, моделирующий карту, показанную на рис. 14.16.

В этом контексте минимальное расстояние между двумя вершинами u и v является минимальной суммой весов ребер по всем путям от u до v. Если такого пути нет, то есть если нет пути из u в v, то расстояние между ними считается бесконечным.

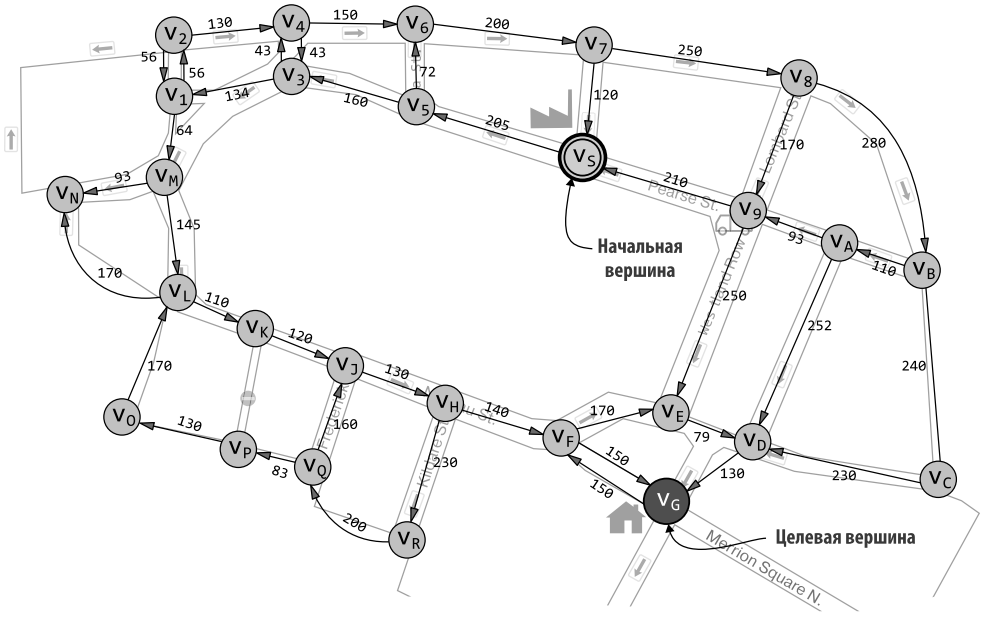


Рис. 14.17. Наложение ориентированного взвешенного графа на карту на рис. 14.16. Веса ребер — это расстояния (в метрах) между пересечениями, смоделированными вершинами ребра

На рис. 14.18 (с более простым примером графа) показано, как работает алгоритм Дейкстры. Он похож на BFS, за исключением двух основных отличий:

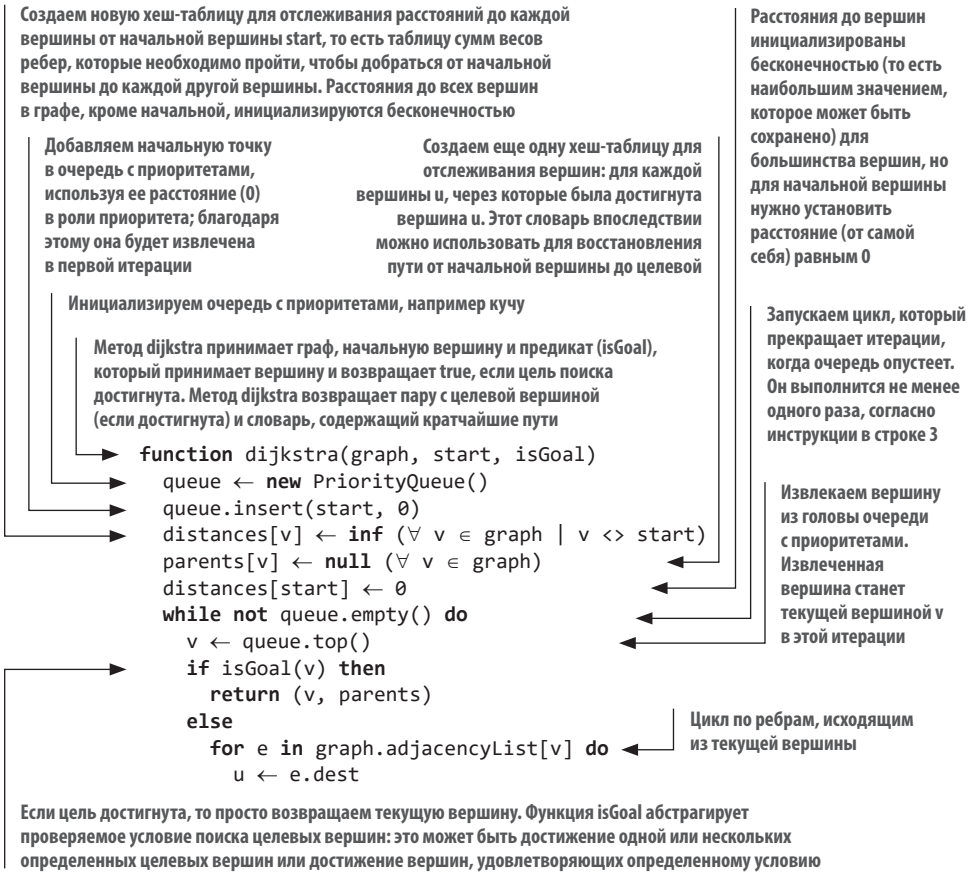
- вместо длин путей используется метрика — сумма весов;
- следовательно, для отслеживания вершин, которые нужно посетить следующими, необходимо использовать другой контейнер: простой очереди недостаточно, нужна очередь с приоритетами.

Все остальное, логика алгоритма и вспомогательные данные, аналогично BFS. Это очень удобно, потому что можно написать алгоритм Дейкстры, опираясь на код в листинге 14.2, внеся лишь небольшие изменения. Однако, если вы думаете, что это сходство — совпадение, то задержите дыхание и подождите с выводами до раздела 14.5[1].

14.4.2. Реализация

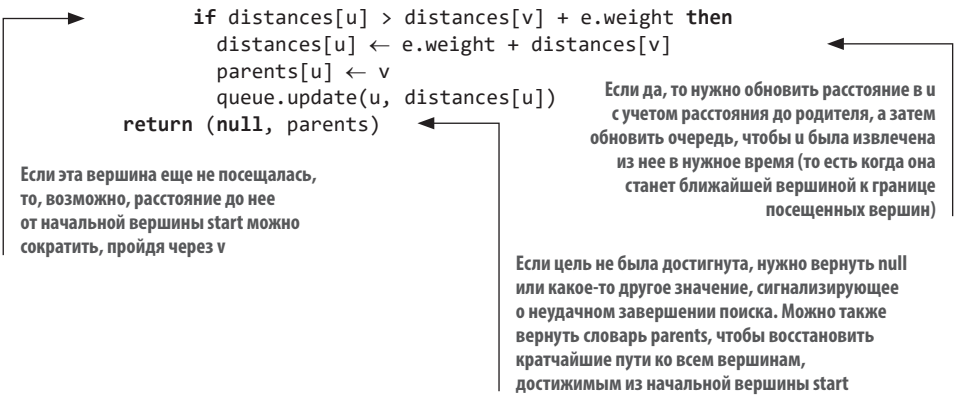
Листинг 14.6 описывает детали алгоритма Дейкстры. Сравнив его с листингом 14.2, можно заметить, как он похож на алгоритм BFS: настолько, что в алгоритме Дейкстры можно использовать тот же алгоритм восстановления кратчайшего пути, который показан в листинге 14.3. Тем не менее необходимо быть еще более осторожными в отношении производительности.

Листинг 14.6. Алгоритм Дейкстры




```
function dijkstra(graph, start, isGoal)
  queue ← new PriorityQueue()
  queue.insert(start, 0)
  distances[v] ← inf (∀ v ∈ graph | v <> start)
  parents[v] ← null (∀ v ∈ graph)
  distances[start] ← 0
  while not queue.empty() do
    v ← queue.top()
    if isGoal(v) then
      return (v, parents)
    else
      for e in graph.adjacencyList[v] do
        u ← e.dest
```

[1] Пожалуйста, не задерживайте дыхание буквально! Даже если вы читаете очень быстро.



14.4.3. Анализ

В реализации BFS каждая вершина добавлялась в (простую) очередь и никогда не обновлялась. Алгоритм Дейкстры, напротив, для отслеживания ближайших обнаруженных вершин использует очередь с приоритетами, при этом приоритет[1] вершины может измениться уже после добавления ее в очередь.

Это обусловлено принципиальной разницей между алгоритмами. В качестве метрики BFS использует только количество пройденных ребер, а если использовать веса ребер, то может обнаружиться путь, включающий больше ребер, но имеющий меньший суммарный вес, чем другой путь, включающий меньше ребер. Например, на рис. 14.18 видно, что между v_1 и v_2 есть два пути. Путь $v_1 \to v_3 \to v_2$ пролегает всего через два ребра, но его суммарный вес равен 8, в то время как другой путь, $v_1 \to v_3 \to v_5 \to v_2$, пролегает через три ребра с суммарным весом всего 5. Второй путь пролегает через большее количество ребер и включает еще одну вершину между v_3 и v_2, поэтому расстояние до v_2 будет первоначально установлено равным 8 (когда будет посещена v_3), а затем обновлено до 5, когда алгоритм посетит вершину v_5.

Из-за такого поведения при каждом посещении новой вершины могут обновляться приоритеты всех ее соседей.

Это, в свою очередь, влияет на асимптотическую производительность алгоритма Дейкстры, которая зависит от эффективности реализации операции обновления приоритета. В частности, для алгоритма Дейкстры можно реализовать очередь с приоритетами как:

- массив (сортированный или несортированный, как описано в приложении B);
- кучу;
- кучу Фибоначчи.

[1] Будем использовать неубывающую кучу для хранения вершин на границе, а их расстояния от начальной вершины будут играть роль приоритетов.

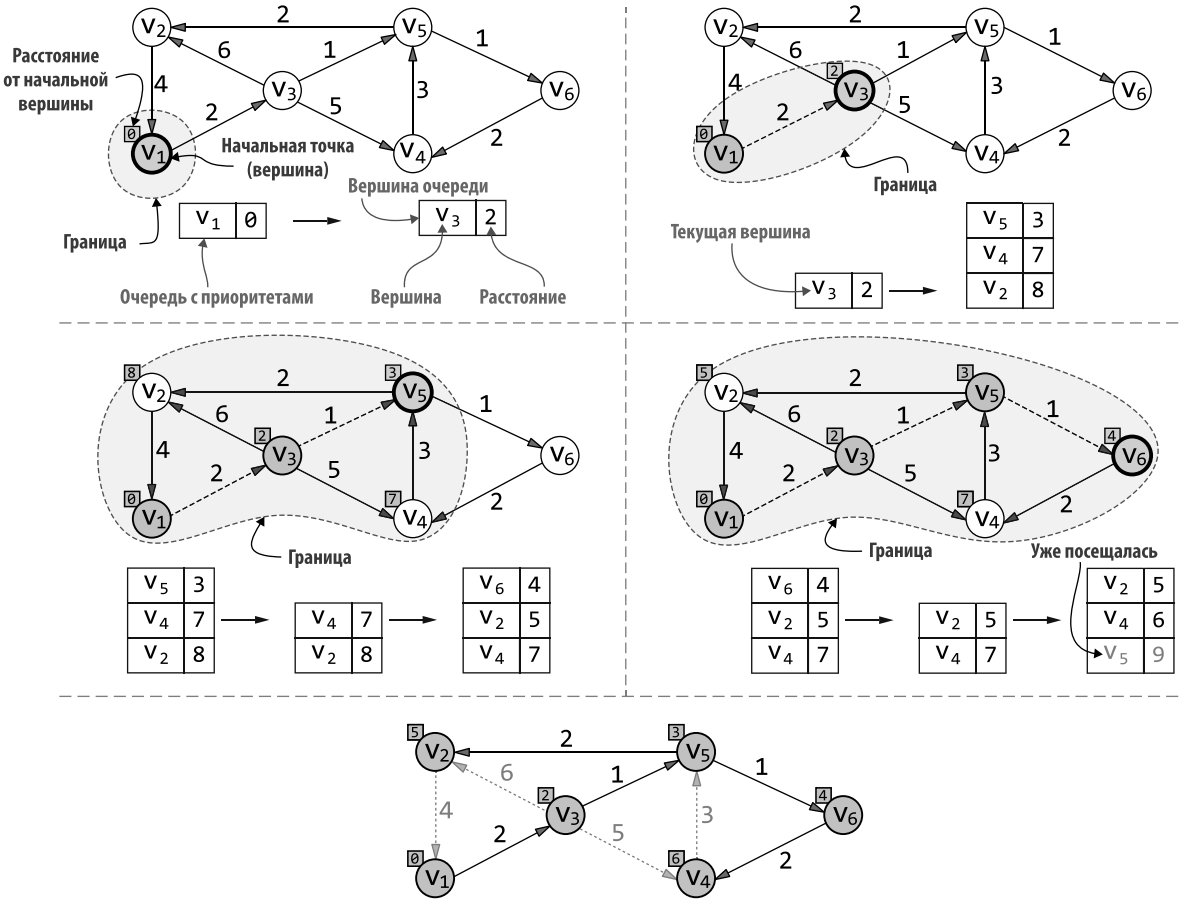


Рис. 14.18. Работа алгоритма Дейкстры на ориентированном графе (из примера на рис. 14.10)

Время выполнения с очередью на основе массива будет равно $O(|E| \times |V|)$, потому что для обновления приоритета или корректировки очереди после извлечения потребуется выполнить $O(|V|)$ операций с каждой вершиной.

С оставшимися двумя вариантами время выполнения равно $O(|V| \times \log(|V|) + |E| \times DQ(|V|))$, где:

- $|B|$ — количество вершин на графе;
- $|E|$ — количество ребер;
- $DQ(|V|)$ — (среднее) время выполнения каждой операции «обновления приоритета».

В табл. 14.2 приводятся обобщенные сведения о времени работы алгоритма Дейкстры в зависимости от реализации очереди с приоритетами.

Таблица 14.2. Время работы алгоритма Дейкстры на связном графе G=(V,E)

	Массив	**Куча**	**Куча Фибоначчи**
Время выполнения	$O(\|E\| \times \|V\|)$	$O(\|E\| \times \log(\|V\|))$	$O(\|V\| \times \log(\|V\|) + \|E\|)^*$

* Амортизированное.

Лучшее теоретическое время получается с кучей Фибоначчи, для которой амортизированное время уменьшения приоритета элемента равно $O(1)$. Однако эта структура данных сложна в реализации, и на практике она неэффективна, поэтому лучше всего использовать кучи. Как уже было показано в разделе 2.9, на практике d-ичные кучи имеют более эффективные реализации.

14.4.4. Кратчайший путь доставки

Выше в этом разделе обсуждалась работа алгоритма Дейкстры, его реализация и производительность.

Теперь осталось только решить, как применить этот алгоритм к рассматриваемому примеру и найти кратчайший путь для доставки заказа покупателю.

Хорошая новость заключается в том, что на самом деле это просто. Создав граф, показанный на рис. 14.17, можно просто применить алгоритм к нему (или к примеру на рис. 14.10, как было сделано при обсуждении BFS) и реконструировать кратчайший путь.

Результат для примера из этого раздела показан на рис. 14.19, здесь вычислены и показаны кратчайшие расстояния от начальной вершины (v_S) до каждой другой вершины.

Обратите внимание, что некоторые ребра, нарисованные линиями из точек, не принадлежат ни одному кратчайшему пути, в то время как кратчайший путь к пункту назначения, v_G, обозначен жирными пунктирными линиями.

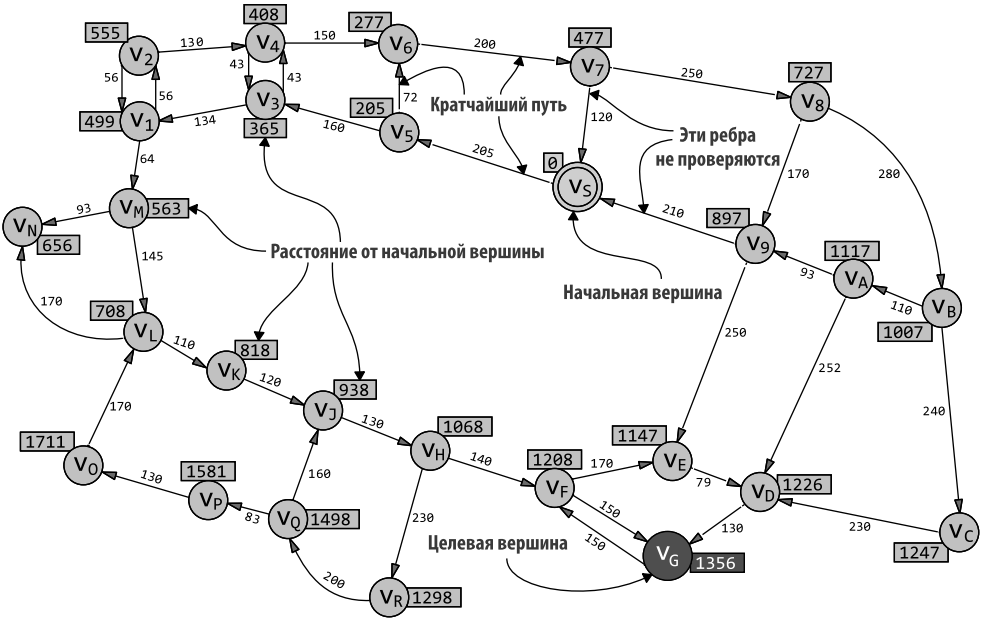


Рис. 14.19. Результат применения алгоритма Дейкстры к примеру на рис. 14.17. Обратите внимание, что от начальной вершины (v_S) до конечной (v_G) есть два пути, имеющих близкие длины; посмотрите, как сложно было бы определить самый короткий (но при этом нелогичный) вариант. Пунктирные линии обозначают ребра на кратчайшем пути, сплошные линии обозначают ребра, которые было просмотрены, но не попали в кратчайший путь, и, наконец, пунктирные линии, изображенные точками, обозначают ребра, которые даже не рассматривались (потому что в какой-то момент становилось ясно, что любой путь, пролегающий через них, будет длиннее найденного кратчайшего пути)

Этот пример идеально подходит для иллюстрации необходимости алгоритмов, таких как алгоритм Дейкстры, потому что от v_S до v_G можно добраться двумя путями, которые имеют почти одинаковые суммарные расстояния, и наша интуиция, скорее всего, выберет самый длинный, потому что он выглядит более линейным.

И последнее замечание: как уже упоминалось, применить алгоритм оказалось простым делом, но только благодаря одному свойству этого графа: у него нет ребер с отрицательным весом. Любое такое ребро фактически нарушило бы предположение, лежащее в основе алгоритма: при расширении границы посещенных вершин и выборе ближайшей непосещенной вершины в каждой итерации, в момент посещения вершины становится известно ее минимальное расстояние от начальной вершины.

Причина в том, что алгоритм Дейкстры (как и BFS) — *жадный* алгоритм, способный найти решение задачи, делая локально оптимальный выбор. Фактически, выбирая, какую вершину посетить следующей, нужно только принять во внимание исходящие ребра вершин, которые мы уже посетили. Жадные алгоритмы можно применять

только к определенным задачам: наличие отрицательных ребер делает задачу непригодной для решения с помощью любого жадного алгоритма, потому что локально-оптимальный выбор становится невозможным.

Ребра с отрицательными весами могут показаться нелогичными, но на самом деле они довольно распространены. Если измерять расстояния количеством топлива, израсходованного на перемещение между двумя вершинами, и цель состоит в том, чтобы не остаться с пустым баком, то ребро, соответствующее дороге с заправочной станцией, может иметь отрицательный вес. Точно так же, если связать стоимость с топливом, преимущество, позволяющее выполнить вторую доставку или получение, может иметь отрицательную стоимость, потому что появляется возможность попутно заработать дополнительные деньги.

Для решения задач с отрицательными ребрами нужно использовать алгоритм *Беллмана — Форда* (Bellman — Ford) — оригинальный алгоритм, использующий прием динамического программирования для получения решения, учитывающего ребра с отрицательным весом. Алгоритм Беллмана — Форда обходится дороже, чем алгоритм Дейкстры; его время работы равно $O(|V| \times |E|)$. Зато его можно применять к более широкому множеству графов. Хотя он также имеет некоторые ограничения: не может работать с графами, имеющими циклы с отрицательным весом[1] (в то же время его можно использовать для поиска таких циклов).

14.5. ЗА ПРЕДЕЛАМИ АЛГОРИТМА ДЕЙКСТРЫ: A*

Алгоритмы BFS и Дейкстры очень похожи друг на друга, и оба они, как оказывается, являются частным случаем алгоритма A* (произносится как «A-звездочка»).

Этот алгоритм, показанный в листинге 14.7, не просто более общий, он улучшает производительность алгоритма Дейкстры как минимум в двух разных ситуациях. Однако, прежде чем углубляться в эти ситуации, рассмотрим различия между обоими этими алгоритмами и A*.

Как можно видеть в строке 1 в листинге 14.7, приведенная обобщенная реализация алгоритма A* принимает два дополнительных аргумента: функцию расстояния и эвристику. Оба участвуют в вычислении так называемой f-оценки в строке 18. Рассчитываемое значение сочетает стоимость достижения текущего узла *u* из начальной вершины и ожидаемую стоимость достижения цели из *u*.

Управляя этими двумя аргументами, можно получить либо алгоритм BFS, либо алгоритм Дейкстры (или ни тот ни другой). Для обоих этих алгоритмов эвристика должна быть функцией, тождественно равной 0, то есть в этом аргументе можно передать `heuristic = lambda(v)` $\rightarrow 0$. Оба алгоритма, по сути, полностью игнорируют любое понятие расстояния от вершин до цели или информацию о нем.

---

[1] Если граф имеет цикл с отрицательным весом — цикл, сумма весов ребер которого отрицательна, — то обсуждение кратчайших путей может оказаться бессмысленным. На самом деле, многократно проходя через цикл, можно получить сколь угодно малую общую стоимость.

Листинг 14.7. Алгоритм A*

Для метрики расстояния ситуация иная:

- алгоритм Дейкстры использует вес ребра как функцию расстояния, поэтому нужно передать что-то вроде `distance = lambda(e) → e.weight`;
- BFS учитывает только количество пройденных ребер, что эквивалентно случаю, когда все ребра имеют одинаковый вес, тождественно равный 1! Как следствие, можно передать `distance = lambda(e) → 1`.

На практике в 99,9 % случаев лучше напрямую реализовать алгоритм Дейкстры или BFS, а не как частный случай A*. Это подводит нас к золотому правилу упрощения, которому я научился у одного великого инженера, когда работал с ним.

ПРИМЕЧАНИЕ

Не обобщайте код без необходимости. Не следует думать о написании универсальной версии чего-либо, пока не появится хотя бы три разных варианта, которые можно реализовать путем незначительной модификации одного и того же универсального кода[1].

Обобщенный код, такой как универсальная версия A* в листинге 14.7, обычно отягощен некоторыми накладными расходами, например обусловленными вызовами методов или лямбда-выражений, таких как `distance`, вместо простого получения длины ребра, или (например, в случае с BFS) использованием очереди с приоритетами вместо более быстрой простой очереди. Универсальный код также труднее поддерживать и рассуждать о порядке его выполнения.

Из этих соображений должно быть ясно, что алгоритм A* не предназначен для использования в роли универсального метода, поведение которого настраивается параметрами. И все же он может быть чрезвычайно полезен.

Как уже упоминалось, есть по крайней мере две веские причины для реализации A* — два сценария, в которых A* дает преимущество перед алгоритмом Дейкстры.

Для начала уточню: это не всегда так. В общем случае алгоритм Дейкстры асимптотически так же быстр, как A* (иногда последний может быть даже неприменим по смыслу). A* дает преимущество только в некоторых сценариях, когда есть дополнительная информация, которую можно каким-то образом использовать.

Первый сценарий, когда можно использовать A* для ускорения поиска пути к цели, — если имеется информация о расстоянии от всех или некоторых вершин до цели (целей). Схема на рис. 14.20 объясняет эту ситуацию лучше, чем тысяча слов! Обратите внимание, что в данном конкретном случае ключевым фактором является

[1] Конечно, использование таких шаблонов проектирования, как «Макет» или «Стратегия», поможет избавить код от повторений и сделать его более простым в обслуживании, но дело в том, что реализация универсальных методов или классов сделает код менее чистым и удобным для сопровождения, поэтому старайтесь сбалансировать оба аспекта.

наличие в вершинах, моделирующих физическое местоположение в реальном мире, дополнительной информации, которая может помочь оценить расстояние от них до конечной цели. Это не всегда верно и обычно не относится к общим графам.

Другими словами, дополнительная информация исходит не из графа, а из знания предметной области.

Однако такое преимущество довольно сомнительно, потому что нет априорной гарантии, что A* будет работать лучше алгоритма Дейкстры. Наоборот, легко создать пример, в котором A* всегда будет работать так же плохо, как алгоритм Дейкстры. Взгляните на рис. 14.21, чтобы понять, как можно изменить предыдущий пример, чтобы обмануть A*! Ключевым моментом является качество дополнительной информации, получаемой эвристической функцией: чем надежнее и ближе оценка к реальному расстоянию, тем лучше работает A*.

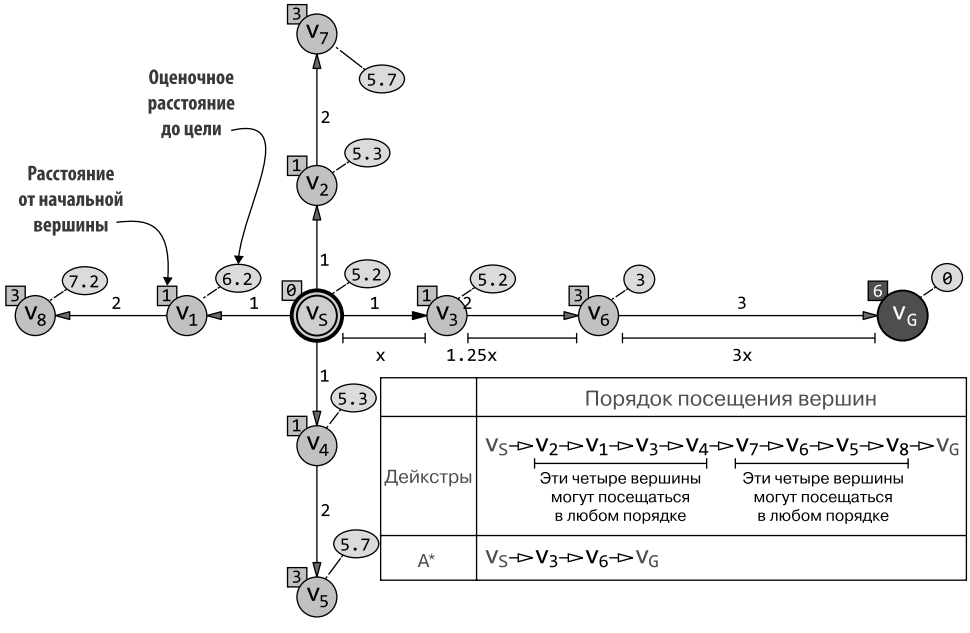


Рис. 14.20. Пример графа, где A* обеспечивает значительное ускорение по сравнению с алгоритмом Дейкстры. Конечно, это крайний случай, но часто имеются некоторые знания предметной области, которые A* может использовать для исключения ветвей из поиска и тем самым значительно ускорить его. Обратите внимание, что здесь прямолинейное расстояние между вершинами, используемое в качестве эвристики, а также веса ребер выражаются в единицах, кратных общей единице, обозначенной как x (это могут быть метры, километры и т. д.). Числа в квадратах рядом с каждой вершиной — это расстояние от данной вершины до начальной, а числа в пунктирных эллипсах — предполагаемые расстояния до цели, вычисленные как расстояние по прямой линии от данной вершины до вершины v_G

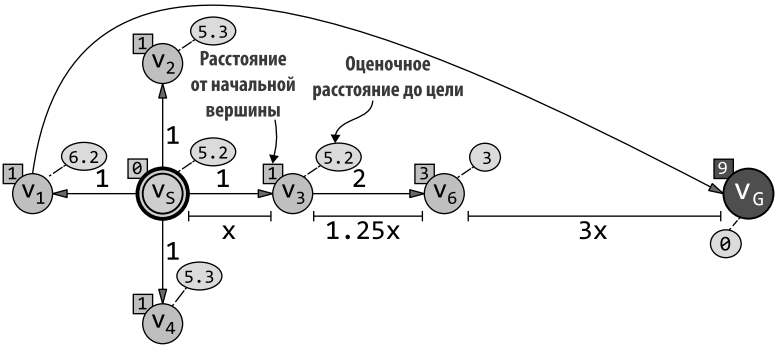
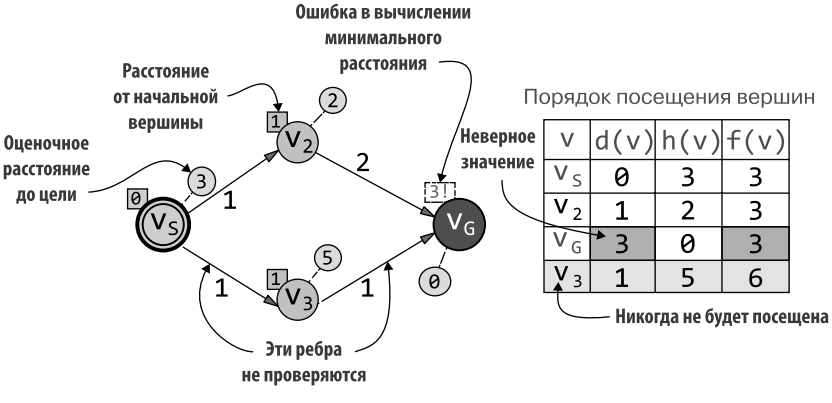


Рис. 14.21. Крайний случай, когда A* не сможет превзойти алгоритм Дейкстры. Оба алгоритма посетят все вершины (A* всегда в одном и том же порядке, Дейкстры — в частично случайном порядке), прежде чем им наконец удастся пройти по ребру из v_1 в v_G

14.5.1. Насколько хорош алгоритм A*?

Если бы вы слушали лекцию на эту тему в аудитории, то сейчас могли бы задать дополнительный вопрос: можно ли создать пример, в котором A* работает постоянно хуже алгоритма Дейкстры?

Как оказывается, легко: если взять для примера граф на рис. 14.21 и изменить вес ребра, соединяющего v_1 и v_G, установив любое значение меньше 2, но сохранив те же оценки, то можно быть уверенными, что A* посетит все остальные вершины, прежде чем доберется до цели, тогда как алгоритм Дейкстры никогда не посетит v_6 и, в зависимости от порядка обработки ребер из v_S, может также пропустить вершины с v_2 по v_4.



v	d(v)	h(v)	f(v)
v_S	0	3	3
v_2	1	2	3
v_G	3	0	3
v_3	1	5	6

Рис. 14.22. Другой крайний случай для A*, когда алгоритм возвращает неоптимальное решение. Поскольку оценка для вершины v_3 раздута, цель достигается другим путем, и даже до посещения вершины v_3. Как только цель достигнута, поиск останавливается и возвращается неверный (неоптимальный) путь

Ключевым для этого примера является завышение эвристики расстояния до цели для v_1 (и еще нескольких вершин): A* стремится минимизировать расчетную стоимость, которая может отличаться от фактической, и всякий раз, когда эта оценка оказывается пессимистичной, алгоритм ошибается.

Фактически предыдущий пример показывает, как использование неправильных оценок может сделать поиск излишне медленным. Конечно, неудобно, но иногда приемлемо. Но и это еще не все. Могло бы быть намного хуже, потому что также можно найти примеры, когда A* возвращает неоптимальное решение. На рис. 14.22 оценка для вершины v_3 раздута, и, следовательно, вершина v_G достигается другим путем, еще до посещения вершины v_3. Напомню, что как только цель достигнута, алгоритм прекращает поиск и возвращает неправильный (неоптимальный) путь.

Иногда это можно считать приемлемым, но на практике такое поведение обычно нежелательно, и, к счастью, есть способ его избежать.

Фактически можно доказать, что A* является *полным*[1] и *оптимальным*, когда эвристическая функция удовлетворяет двум условиям: она должна быть *допустимой* и *непротиворечивой*. Поясню:

- *допустимая* (или *оптимистичная*) эвристика никогда не завышает стоимость достижения цели;
- *непротиворечивой* считается эвристика, которая для данной вершины v и любой из последующей за ней вершиной u дает расчетную стоимость u, не превышающую расчетной стоимости v плюс стоимость перехода из v в u: `heuristic(u)` ≤ `distance(v, u)` + `heuristic(v)`.

Есть еще более простой способ запомнить разницу между этими двумя условиями. Его предложил один из рецензентов этой книги: *допустимость* означает невозможность завышения стоимости пути, а последовательность — невозможность завышения стоимости ребра.

Планируя использовать поиск A*, первое, что нужно сделать, — убедиться в наличии эвристической функции, которая одновременно допустима и непротиворечива. Такое условие необходимо и достаточно, чтобы найти оптимальное решение[2].

Что все это означает для нашей задачи доставки заказов покупателям? Для ускорения поиска наилучшего маршрута в качестве эвристики можно использовать «расстояние до клиента по прямой». Она будет направлять поиск, отдавая предпочтение путям, приближающим к цели, а не удаляющим от нее; в свою очередь, потребуется посетить меньшее количество вершин, прежде чем цель будет достигнута.

---

[1] Полнота гарантирует, что для достижения цели потребуется посетить только конечное число узлов.

[2] Для деревьев достаточным условием является допустимость. Сможете ли вы объяснить почему? Подсказка: сколько в дереве путей от корня к целевой вершине проходит через данный узел u?

На рис. 14.23 показано, как A* находит лучший маршрут для предыдущего примера, к которому мы ранее применили алгоритм Дейкстры (рис. 14.19). Алгоритму Дейкстры пришлось бы посетить все вершины, расстояние которых от начальной вершины меньше 1356 (общая длина кратчайшего пути от начальной вершины до цели). Алгоритм A* способен достичь цели быстрее, и, хотя он по-прежнему посещает некоторые вершины, находящиеся не на кратчайшем пути, он не посещает вершины v_A, v_B, v_C, v_F, v_N, v_R, которые посещает алгоритм Дейкстры.

В этом конкретном примере применение A* может сэкономить время за счет отказа от прохождения 6 ребер из 23, что довольно хорошо (экономия 25 %), особенно если учесть, что есть два пути с очень близкими весами, отличающимися всего на 2 метра.

Рассматривая расстояние, преодолеваемое по дороге, как вес ребра и расстояние по прямой как эвристику, заметим, что расстояние по прямой по своей природе оптимистично, потому что расстояние, преодолеваемое по дороге, в принципе не может быть меньше расстояния по прямой и в лучшем случае будет совпадать с ним.

Кроме того, расстояние по прямой является последовательной эвристикой. На самом деле, если применить наш выбор к условию непротиворечивости, то получится:

```
straight_line(u, goal) ≤ road_distance(v, u) + straight_distance(v)
```

что, безусловно, верно, так как `straight_line(v,u)` ≤ `road_distance(v,u)`, а расстояние по прямой, являющееся евклидовым расстоянием, определенно удовлетворяет неравенству треугольника[1].

Хотя это условие выполняется всегда, значение эвристики для вершины u может быть больше значения, присвоенного ее родителю v. Например, на рис. 14.23 взгляните на вершины v_E и v_D: расстояние по прямой между вершиной v_D и целевой вершиной больше, чем расстояние от ее родителя v_E. Причина в том, что не все вершины соединены ребрами (точнее, двумя направленными ребрами), и поэтому, скажем, из v_E (находящейся ближе к цели) попасть в v_G можно только обходным путем — через v_D.

Можно доказать, что если бы граф был полносвязным, то вершины посещались бы в соответствии с их расстоянием до цели по прямой. Другими словами, значение эвристики будет монотонно уменьшаться при обходе.

С другой стороны, большинство графов, встречающихся на практике, не являются полносвязными и, к счастью, это строгое условие не нужно, чтобы A* мог найти оптимальное решение. Но требование допустимости и непротиворечивости эвристики должно выполняться.

Эти свойства гарантируют, что алгоритм найдет наилучшее решение, но стоит ли стремиться получить максимально точную оценку? Можно придумать множество допустимых и непротиворечивых эвристик, но найдет ли алгоритм лучший маршрут быстрее, если будет выбирать маршрут с более точной оценкой?

---

[1] Для трех точек A, B и C, принадлежащих евклидову пространству, всегда верно условие distance(A,C) ≤ distance(A,B) + distance(B,C).

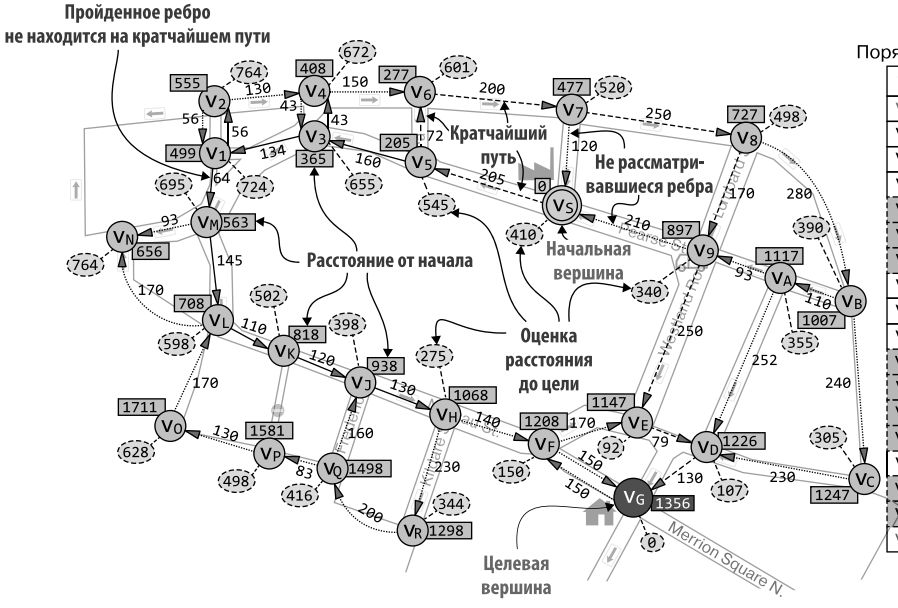


Рис. 14.23. Применение A* к графу на рис. 14.19: алгоритму Дейкстры пришлось посетить все вершины, расстояние которых от начальной вершины меньше 1356, а алгоритм A* достигает цели быстрее, и хотя он все еще посещает некоторые вершины, не находящиеся на кратчайшем пути (выделены заливкой в таблице справа), он не просматривает вершины v_A, v_B, v_C, v_F, v_N, v_R

Излишне говорить, что, когда приходится вычислять тысячи маршрутов в час, использование более быстрого поиска может сэкономить на вычислениях, позволить отправлять заказы с завода быстрее и в конечном счете экономить деньги.

И снова можно получить теоретическую гарантию производительности A*: если зафиксировать эвристику и расстояние, то для любой непротиворечивой эвристики A* будет не просто оптимальным алгоритмом, но и оптимально-эффективным. Это означает, что никакой другой алгоритм не сможет гарантировать посещение меньшего количества узлов, чем A*.

При этом чем ближе оценка к фактическому расстоянию от вершины до цели, тем быстрее алгоритм достигнет цели. В качестве примера попробуйте придумать лучшую эвристику для графика на рис. 14.23. (Подсказка: что может быть точнее расстояния по прямой?)

Итак, на данный момент мы радуемся найденному способу гарантировать, что A* найдет оптимальное решение, но также осознаем прозвучавшее здесь упоминание, что иногда можно согласиться с неоптимальным решением. Когда стоимость обхода высока или есть ограничения на время отклика, может быть желательно выбрать недопустимую эвристику, гарантирующую более быструю сходимость.

14.5.2. Эвристика как способ балансировки в реальном времени

На этом мы заканчиваем обсуждение оптимальности поиска. Выше упоминалось, что существует как минимум еще один сценарий, в котором A* может оказаться полезным. Рассмотрим его кратко.

Огромный потенциал эвристической функции заключается в возможности использовать другую метрику расстояния и передавать больше информации о предметной области. Эвристика может даже объединять несколько данных, если возвращаемое значение соответствующим образом масштабируется, чтобы иметь смысл по сравнению с длиной ребра. Например, бессмысленно (и это привело бы к огромной потере производительности) использовать метры (или даже миллиметры) в роли весов ребер и секунды в качестве оценок эвристики. Однако всегда можно масштабировать миллиметры в метры или, если эвристика возвращает информацию о среднем времени, необходимом для достижения цели из каждой вершины, умножить ее на среднюю или минимальную скорость движения по дорогам, чтобы получить величину, которую затем прибавить к весу любого ребра.

Но мы также можем подняться на следующий уровень, еще больше отделив расстояния и эвристики. Представьте, что лучший маршрут вычисляется на ходу, а не заранее, как это делает автомобильный навигатор.

Для этого в первую очередь следует переключить метрику с расстояния на время в пути, а время, необходимое для достижения цели, менять в зависимости от наличия пробок, погодных условий, районов, закрытых для проезда в определенные часы, и т. д.

Однако среднее время, необходимое для следования по определенному маршруту, известно заранее, и его возможно использовать в качестве компаса для взвешивания решений, принимаемых в реальном времени.

Предположим, например, что вы планируете расширить услуги доставки и теперь вам нужно организовать переправку каких-то товаров из Неаполя во Флоренцию. Самый быстрый маршрут до Рима пролегает по автомагистрали, проходящей к востоку от города, и обычно для его преодоления требуется чуть меньше трех часов. Однако наш навигатор замечает, что следующие 10 миль на этом маршруте встретится интенсивное движение, а при обходе Рима с западной стороны движения практически нет. Если бы навигатор просто воспользовался алгоритмом Дейкстры, то следующим он бы выбрал более быстрое ребро и привел вас на запад.

К сожалению, это может увеличить продолжительность поездки как минимум на час. Алгоритм A* может прийти на помощь и сбалансировать краткосрочное преимущество более свободного участка автомагистрали с долгосрочным преимуществом более короткого и, как правило, более быстрого маршрута.

Это чрезвычайно упрощенный сценарий, но суть, я думаю, понятна. A* способен лучше балансировать долгосрочные затраты с локальным выбором, и именно поэтому он был самым современным в области ИИ, например, для поиска пути в видеоиграх[1].

Следующий шаг в улучшении этих алгоритмов навигации — переоценка обработки запросов на поиск кратчайшего пути в вакууме. Это особенно важно в часы пик, когда многие пользователи запрашивают поиск кратчайшего пути между похожими местами, в результате чего отправка всем одного и того же маршрута может привести к появлению пробок на одних дорогах при почти полном отсутствии движения на других дорогах. Было бы здорово, если бы система маршрутизации могла учитывать все запросы, связанные с одними и теми же сегментами, распределять трафик по нескольким маршрутам и минимизировать пробки из-за того, что маршруты прокладываются по одним и тем же дорогам.

Надо сказать, весьма амбициозная цель. Ее обсуждение выходит за рамки книги. К тому же достигнуть ее на классических компьютерах невозможно. Именно поэтому ею занимаются разработчики квантовых технологий[2] и получают обнадеживающие результаты.

---

[1] Другие примеры практического применения алгоритмов A*, Дейкстры и BFS варьируются от IP-маршрутизации до теории графов и даже сборки мусора.

[2] Подробности можно найти здесь: http://mng.bz/w99B. Теория графов — одна из тех областей, где квантовые вычисления могут произвести революцию; для знакомства с практикой квантовых вычислений я рекомендую книги Quantum Computing for Developers Джона Вос (Johan Vos), Manning Publications, 2021; и Learn Quantum Computing with Python and Q# Сары С. Кайзер (Sarah C. Kaiser) и Кристофера Е. Гранда (Christopher E. Granade), Manning Publications, 2021 (*Кайзер С., Гранад К.* Изучаем квантовые вычисления на Python и Q#).

РЕЗЮМЕ

- Граф — это структура данных, пригодная для моделирования многих задач, она в целом подходит для работы с объектами, связанными некоторым отношением близости.
- Обычно представление графа в виде списка смежности лучше подходит для большинства случаев, но иногда для решения задач с плотными графами предпочтительнее использовать матрицу смежности.
- Существует много возможных способов исследования графа, но наиболее распространенными являются поиск в ширину (Breadth First Search, BFS) и поиск в глубину (Depth First Search, DFS).
- Алгоритм Дейкстры расширяет BFS и используется для поиска кратчайшего пути, минимизируя суммарный вес ребер, а не их количество.
- Алгоритм A* работает лучше метода Дейкстры, когда есть дополнительная информация, кроме веса ребер, и ее можно передать в эвристику, оценивающую расстояние от каждой вершины до цели.

15 *Представление графов и планарность: рисование графа с минимальным числом пересечений ребер*

В этой главе

- ✓ Представление графов на двумерной плоскости.
- ✓ Определение планарности графа.
- ✓ Полные графы и полные двудольные графы.
- ✓ Обсуждение алгоритмов определения планарности графа.
- ✓ Определение минимального числа пересечений для непланарных графов.
- ✓ Реализация алгоритмов поиска пересекающихся ребер.

После знакомства с основными свойствами графов в главе 14 пришло время сделать следующий шаг: нарисовать граф. Уже говорилось о графах с абстрактной точки зрения, но, чтобы описать работу алгоритмов поиска кратчайшего пути, все равно приходилось визуализировать их. В главе 14 мы рисовали графы вручную и считали это само собой разумеющимся. А можно ли как-то автоматизировать представление этих структур данных в евклидовом пространстве и, в частности, на двумерной плоскости?

Это не всегда необходимо и не всегда возможно, однако есть много приложений, где порядок размещения вершин и ребер графа на плоскости играет важную роль. Возьмем, к примеру, проектирование *печатной платы*, изображенной на рис. 15.1. Расположение на плате электронных компонент (вершин) и проводящих дорожек (ребер) очень важно не только для нормального функционирования схемы, но и для оптимизации производственного процесса, уменьшения количества используемой меди и снижения общих расходов.

В этой главе я буду постепенно знакомить вас с основными идеями представления графов, сосредоточив внимание преимущественно на двумерных плоскостях, планарных графах и на минимизации пересечений ребер, когда граф не является планарным.

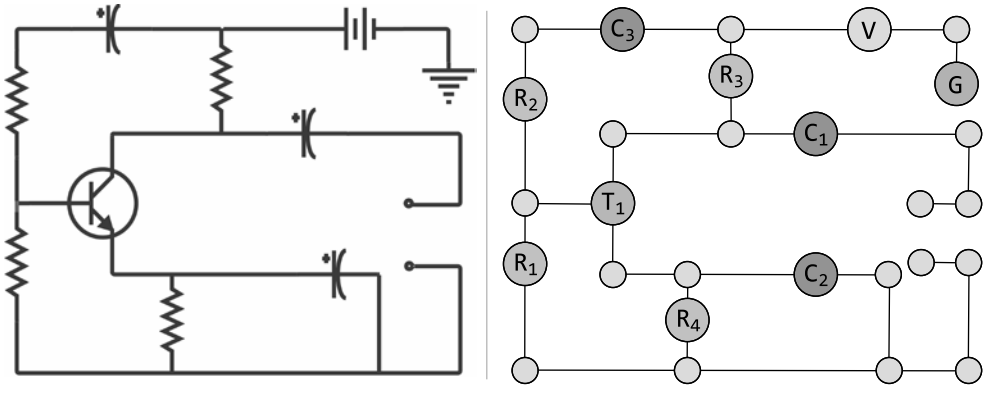


Рис. 15.1. Пример электронной схемы и ее представление в виде графа, из которого можно получить рисунок печатной платы. Как можно видеть на графе, все электронные компоненты имеют вершины (окрашены разными цветами в зависимости от типа) с двумя концами и соединения, смоделированные на графе как пары вершин с перемычкой (чтобы можно было ограничиться горизонтальными и вертикальными сегментами, соответствующими токопроводящим дорожкам)

В процессе объяснения этих понятий будет заложена основа для создания приложения, принимающего граф и отображающего его на экране (или, что то же самое, на бумаге).

15.1. ПРЕДСТАВЛЕНИЕ ГРАФОВ

Графы — удивительная структура данных. В главе 14 мы рассмотрели их лишь в самых общих чертах, затронув алгоритмы Дейкстры и A*, а также коснувшись некоторых интересных сфер применения графов. Но вы, должно быть, слышали еще и о графах знаний[1] или графовых базах данных[2], таких как Neo4J, и это лишь некоторые из тех областей применения графов, что на слуху в наши дни.

---

[1] Граф знаний — чрезвычайно сложная структура данных, объединяющая в одном графе и сами данные, и способ их понимания. В Google, например, граф знаний используется для уточнения семантики поиска.

[2] Графовые базы данных опираются на сильные взаимосвязи, присутствующие в современных данных, что позволяет организовывать и запрашивать информацию с учетом семантики, используя ребра графа для моделирования динамических отношений между данными (вершинами графа). Их можно рассматривать как классические реляционные БД, превращенные в более гибкие и даже более мощные. Вспомните, например, Neo4J или Fullstack GraphQL.

Но графы также используются для моделирования более осязаемых вещей; например, печатную плату можно представить в виде графа с электронными компонентами в виде вершин и токопроводящих дорожек (обычно они сделаны из меди) в виде ребер[1]. Кроме того, нам, людям, часто требуется иметь визуальное представление графа перед глазами, чтобы лучше понять его. Рассмотрим, например, блок-схему (которая, что неудивительно, является графом), изображенную на рис. 15.2.

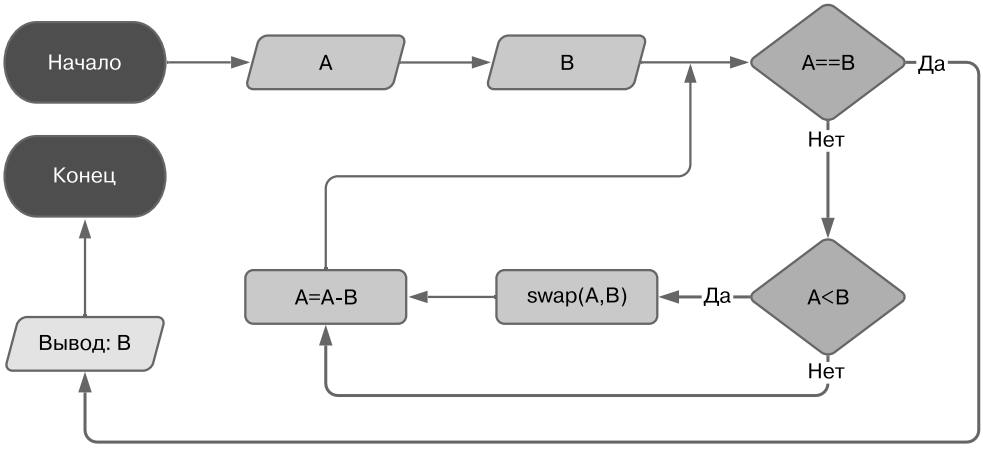


Рис. 15.2. Блок-схема (в данном примере для алгоритма вычисления наибольшего общего делителя (НОД) двух чисел) — это особый вид графа

Имея наглядное изображение, легко можно проследить поток выполнения и получить представление о его структуре. Но взгляните на формальное определение в терминах вершин и ребер графа:

```
G = (
V = [Начало, A, B, A==B, A<B, swap(A,B), A=A-B, Вывод: B, Конец],
E = [Начало -> A, A -> B, A==B -[Yes]-> Output: B, A==B -[No]-> Вывод: B,
➥ A<B -[Да]-> swap(A,B), A<B -[Нет]-> A=A-B, swap(A,B) -> A=A-B, A=A-B ->
➥ A==B, Вывод: B -> Конец]
)
```

Что проще понять, это определение или его визуальное представление?

Я думаю, все согласятся с тем, что визуальное представление графов незаменимо, по крайней мере когда приходится изучать и обрабатывать их вручную. Хотя это требуется не всегда[2], но в реальной жизни немало примеров, когда без визуали-

[1] В случае с печатными платами ребра изображаются как ломаные линии, состоящие из перпендикулярных отрезков.

[2] Например, едва ли кто-то сможет понять структуру графа знаний в Google, взглянув на него (учитывая, что он имеет гигантское количество вершин и ребер): такие графы и графовые базы данных не предназначены для осмысления человеком и обрабатываются исключительно с применением алгоритмов.

зации графов не обойтись, например: блок-схемы алгоритмов, диаграммы UML, диаграммы PERT и т. д.

Следующее, с чем нужно согласиться (или не согласиться), — не все визуальные представления одинаково ценны. Посмотрите на рис. 15.3 и сравните его с рис. 15.2. Не знаю, как у вас, но у меня после взгляда на диаграмму на рис. 15.3 возникает желание посмотреть также формальное определение графа: настолько запутанной выглядит диаграмма.

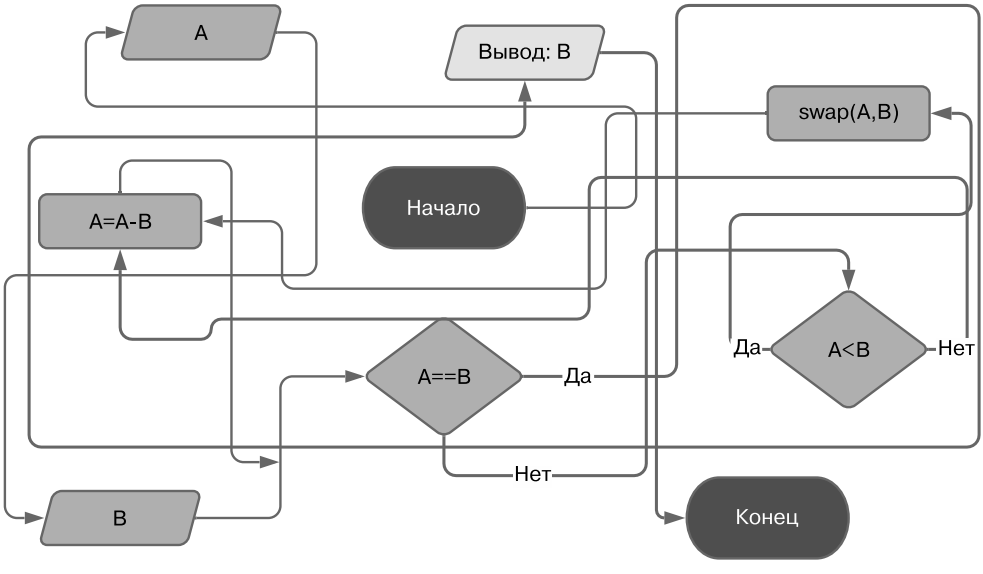


Рис. 15.3. Та же блок-схема, что и на рис. 15.2, но с другим расположением элементов. Вы все еще способны понять ее?

Ключевое различие между двумя представлениями в том, что на рис. 15.3 ребра пересекаются друг с другом несколько раз, что затрудняет следование взглядом по ним. На рис. 15.2 ребра не пересекаются — это *планарное представление планарного (плоского) графа*! Не волнуйтесь, я дам определение этих терминов чуть ниже. А перед этим хочу поделиться еще одним соображением: схема может стать еще менее понятной, если ребра будут пересекать вершины.

15.1.1. Некоторые основные определения

В предыдущем разделе было показано, что изображение графа с непересекающимися ребрами делает визуализацию намного более простой и понятной. Но всегда ли можно избежать пересечений?

Запомните этот вопрос, мы еще вернемся к нему. А пока рассмотрим пару определений, которые будем использовать в этой и последующих главах.

Рисование графа на плоскости можно представить как размещение вершин в двумерном евклидовом пространстве. Неформально каждую вершину можно представить как точку в $\mathbb{R}^2$ (множество всех пар действительных чисел), а каждое ребро — как дугу (или ломаную линию) между двумя вершинами.

Более формально планарное представление можно определить как изоморфизм (отображение 1:1) между абстрактным графом G и планарным (плоским) графом G'.

Планарный граф, в свою очередь, определяется как пара конечных множеств (V, E) (они обозначают вершины и ребра соответственно), таких что:

- V является подмножеством $\mathbb{R}^2$;
- каждое ребро $e \in E$ является участком *жордановой кривой*, проходящей через две вершины;
- никакие два ребра не имеют одинаковых пар концов;
- никакое ребро не пересекает вершину (кроме его конечных точек) или любое другое ребро.

Здесь появился новый термин — жорданова кривая (или кривая Жордана), — требующий определения. Жорданова кривая — это плоская, простая и замкнутая кривая, то есть непрерывная петля, не пересекающаяся с самой собой. Несколько примеров можно увидеть на рис. 15.4.

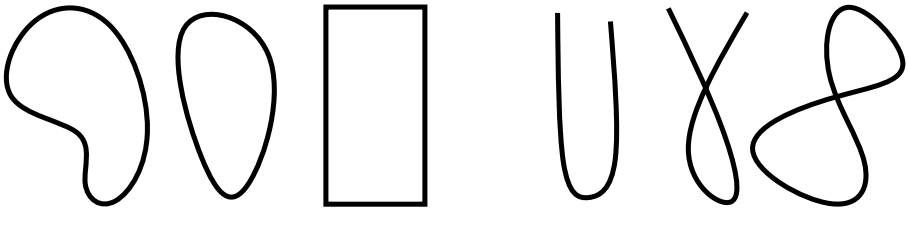


Рис. 15.4. Примеры жордановых и нежордановых кривых. Жорданова кривая — это замкнутая кривая, не имеющая точек пересечения с самой собой (она не «скручена», как в последних двух примерах). Обратите внимание, что прямоугольник (или любой другой многоугольник) является допустимой кривой Жордана. Для представления ребер графов мы будем использовать фрагменты кривых Жордана, поэтому единственным ограничением останется требование к отсутствию их пересечений с самими собой

То есть планарный граф определяется как абстрактный граф G, имеющий планарное представление.

Теперь вернемся к нашему вопросу, переформулировав его с использованием наших определений: все ли графы планарные?

К сожалению, нет, не все графы планарные. Первый алгоритм проверки планарности графа был предложен польским математиком Казимежем Куратовским (Kazimierz

Kuratowski). Его теорема характеризует планарность в терминах *запрещенных графов* (forbidden graphs). Фактически в ней говорится, что планарный граф не может содержать два конкретных непланарных графа в виде своих подграфов.

Простейшими двумя непланарными графами являются: полный граф K_5 и полный двудольный граф $K_{3,3}$. Теорема Куратовского (теорема КК) утверждает, что «граф является планарным тогда и только тогда, когда он не содержит в виде подграфа ни K_5, ни $K_{3,3}$, ни какую-либо *разновидность* этих двух графов».

Это удивительный результат, но чтобы лучше его оценить, нужно рассмотреть еще несколько определений.

15.1.2. Полные и двудольные графы

Полный граф — это граф, каждая вершина которого соединена ребром с каждой другой вершиной графа. В этих графах количество ребер максимально для простых графов и находится в квадратичной зависимости от количества вершин: $|E| = O(|V|^2)$.

Однако обратите внимание, что полный граф не содержит петель; следовательно, точное количество ребер полного графа с n вершинами равно $n \times (n-1) / 2$, где $|V| == n$.

Полные графы обозначаются буквой K (от фамилии Куратовский) и нижним индексом, сообщающим количество вершин в графе; поэтому K_5 (рис. 15.5) обозначает полный граф с пятью вершинами, а K_n — полный граф с n вершинами.

Двудольный граф — это связный граф, вершины которого можно разбить на две группы, назовем их A и B, так, что вершины в группе A соединены только с вершинами в группе B (другими словами, ни одна вершина в группе A не имеет ребра, связывающего ее с другой вершиной в группе A; то же относится к группе B).

Полный двудольный граф — это граф, в котором две группы вершин связаны всеми возможными ребрами (петли при этом не допускаются).

$K_{n,m}$ — общий случай полного двудольного графа с двумя разделами, включающими n и m вершин, а $K_{3,3}$ (рис. 15.6) — полный двудольный граф с двумя разделами по три вершины в каждом.

Полный двудольный граф, разделы которого имеют размеры n и m, имеет ровно $n \times m$ ребер.

Общее *представление* (не обязательно плоское) определяется подобно тому, как было показано в предыдущем разделе; это изоморфизм Γ между графом G и подмножеством $G' = (V, E)$ в $\mathbb{R}^2$, такой что:

1) V является подмножеством $\mathbb{R}^2$;
2) каждое ребро $e \in E$ является отрезком на жордановой кривой между двумя вершинами;
3) никакие два ребра не имеют одинаковых пар концов;
4) никакое ребро не имеет пересечений с вершинами (кроме своих конечных точек).

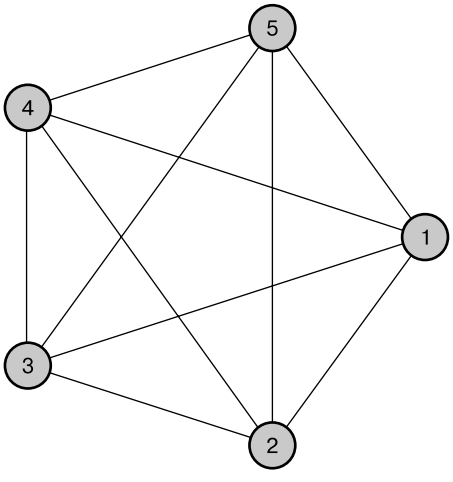


Рис. 15.5. Представление полного графа с пятью вершинами. Как вы считаете, это лучший способ нарисовать его?

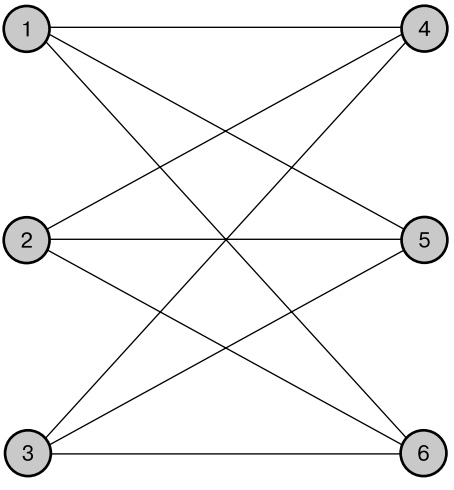


Рис. 15.6. Представление $K_{3,3}$. Как вы думаете, можно ли изобразить этот граф более наглядно?

По сути, по сравнению с определениями планарного (плоского) графа и планарного представления, данными в подразделе 15.1.1, здесь мы отказываемся только от требования отсутствия пересечений ребер.

15.2. ПЛАНАРНЫЕ ГРАФЫ

Теорема Куратовского может показаться нелогичной, так как она определяет планарный граф, перечисляя то, чего он не может содержать, однако она является важным инструментом, так как:

1) выделяет две категории графов (полные и полные двудольные), не являющиеся плоскими (за исключением их наименьших экземпляров), которые нельзя нарисовать на плоскости так, чтобы избежать пересечения их ребер;
2) математически доказывает, что граф не является плоским, когда он таковым не является.

Итак, это отличный инструмент для математических доказательств, но его использование в алгоритмах, автоматически проверяющих планарность графа, — это совсем другая история.

Далее в этом разделе я покажу, как реализовать алгоритм проверки планарности. Но прежде закончим обсуждение графов Куратовского.

Если посмотреть на рис. 15.5, то можно заметить, что представление K_5 имеет пять точек пересечения ребер; однако для каждого абстрактного графа G существует

бесконечно много возможных представлений. Как вы понимаете, существует бесконечно много способов нарисовать G, перемещая каждую вершину немного (или намного) в любом направлении и даже используя разные кривые для ребер (например, бесконечное множество кривых вместо отрезков прямых).

Очевидно, что это справедливо и для K_5. Теперь возникает вопрос: эквивалентны ли все эти представления в отношении пересечений ребер?

Нам уже известен один способ нарисовать K_5 с пятью пересечениями ребер, поэтому, найдя другой способ, где число пересечений ребер больше или меньше, мы получим доказательство, что не все представления одинаковы.

Короче говоря, на рис. 15.7 показано представление K_5 с единственной точкой пересечения двух ребер.

Теперь можно уверенно ответить на вопрос: нет, не все представления эквивалентны. На самом деле работа над поиском «наилучшего»[1] возможного представления графов будет нашей задачей до конца книги[2].

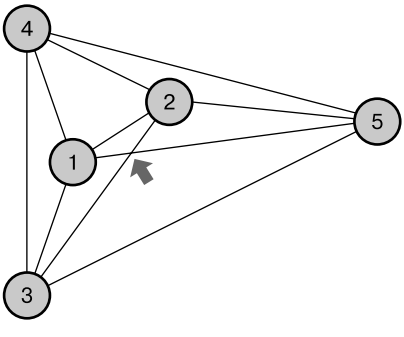


Рис. 15.7. Лучшее представление для K_5. В этом случае есть только одна точка пересечения, обозначенная жирной стрелкой. Обратите внимание, что существует бесконечно много реализаций, эквивалентных этому представлению

15.2.1. Практическое применение теоремы Куратовского

Теорема Куратовского утверждает, что K_5 и $K_{3,3}$ являются простейшими графами, не имеющими планарного представления. Что он имел в виду под словом «простейшими»? В данном случае это означает, что не существует графа меньшего размера (имеется в виду с меньшим количеством вершин или ребер), который не был бы планарным, и поэтому каждый подграф K_5 или $K_{3,3}$ имеет планарное представление.

---

1 Нам также придется обсудить понятие «наилучшего» представления: для начала примем, что под «наилучшим» понимается представление с «наименьшим числом пересечений ребер».

2 Изучая пути решения этой задачи, вы одновременно познакомитесь с новыми алгоритмами и методами, которые также могут применяться в других областях, не только к графам.

Мне всегда казался занимательным факт наличия двух базовых случаев. Единого базового графа найти не удалось, потому что эти два принципиально анизоморфны, но в то же время весьма примечательно, что любой другой непланарный граф можно свести именно к этим двум.

У кого-то из вас мог возникнуть вопрос: откуда известно, что нельзя нарисовать эти графы без пересечений и что нет более простого графа, который не был бы планарным. Все просто: Куратовский это доказал, поэтому можно довериться его теореме.

Но если у вас все еще остались сомнения, попробуйте поперемещать вершины на рис. 15.7 и найти планарное представление. Устраивайтесь поудобнее, потому что может потребоваться некоторое время, прежде чем вы поймете, что это невозможно!

Другую часть утверждения об отсутствии меньшего непланарного графа показать проще. Сосредоточимся на K_5 и сначала посмотрим на граф с меньшим количеством вершин, например на K_4, показанный на рис. 15.8.

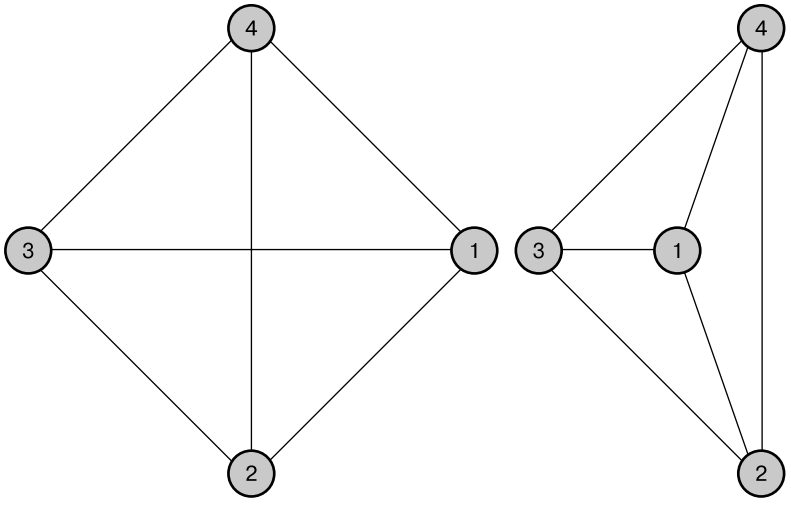


Рис. 15.8. Два представления K_4. Этот граф может показаться хорошим кандидатом на непланарность, но, переместив всего одну вершину, можно найти планарное представление, изображенное справа

На первый взгляд кажется, что этот граф имеет пару пересекающихся ребер. Однако стоит переместить одну из вершин за точку пересечения, и получится плоское представление.

Другая возможность, которую следует исключить, — существование непланарного графа с меньшим количеством ребер, чем у K_5. Однако, если посмотреть на рис. 15.7, сразу станет очевидно, что, удалив ребро $1 \rightarrow 5$ или $2 \rightarrow 3$, мы также избавимся от одного пересечения, это показано на рис. 15.9. Поскольку полный граф симметричен

и инвариантен к перемаркировке[1], можно получить эквивалентное представление (без учета меток) независимо от того, какое ребро удалить из K_5.

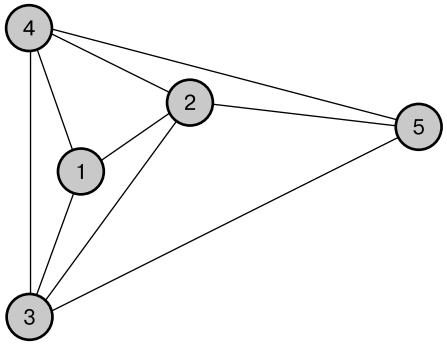


Рис. 15.9. Любой граф, полученный удалением ребра из K_5, можно представить на плоскости без пересечений его ребер

Итак, самые большие подграфы в K_5 планарны, и, следовательно, любой другой граф с пятью или менее вершинами и менее чем с девятью ребрами является планарным.

То же самое можно показать на примере $K_{3,3}$; исследование его подграфов может послужить хорошим упражнением для лучшего понимания двудольных графов и их представлений.

15.2.2. Проверка планарности

Проверить планарность графа сложнее, чем может показаться. Даже проверить планарность представления не так просто: это одна из задач, которую наш мозг легко решает, но определить ход решения в виде алгоритма — нетривиальная задача. Чтобы не прибегать к компьютерному зрению (к которому часто обращаются для помощи в такого рода операциях[2]), нужно формализовать способ рисования ребер, например ограничившись прямолинейными отрезками или кривыми Безье, чтобы можно было использовать математические формулы для определения наличия пересечений. Но в любом случае для проверки количества пересечений на большом графе потребуется много вычислений.

И это только для одного представления. Определить непланарность графа означает доказать, что в любом возможном представлении, которое мы можем придумать, существует по крайней мере одно пересечение.

Мы уже упоминали работу Куратовского о планарных графах, представляя первый метод определения планарности.

---

[1] Неважно, как помечаются вершины, потому что вершины изоморфны: эквивалентны друг другу и каждая связана со всеми остальными.

[2] Помимо вычислительной сложности работы с современным компьютерным зрением, ему требуется большой набор данных и много времени для обучения, но и оно не дает детерминированного алгоритма, что очевидно.

Однако планарность изучается довольно давно, и фактически Эйлер в XVIII веке придумал инвариант (доказанный только в 1811 году Коши), определяющий необходимые условия планарности графа.

Этих условий недостаточно, поэтому их нельзя использовать для доказательства планарности, но они легко вычисляются, а при нарушении исключают планарность.

Вот два условия, которые легко реализовать:

- данный простой связный граф $G = (V, E)$, имеющий не менее трех вершин, является планарным, только если $|E| \leq 3|B| - 6$;
- если $|V| > 3$ и G не имеет петли длиной 3, то G — планарный, только если $|E| \leq 2|V| - 4$.

Итак, на первом шаге в алгоритме проверки планарности можно проверить оба условия за линейное время $O(V + E)$[1]: если одно из них не выполняется, то можно уверенно утверждать, что граф непланарный.

Существует несколько алгоритмов проверки планарности. Ни один из них не является особенно простым в реализации, и многие к тому же неэффективны: первый эффективный алгоритм с линейным временем работы в наихудшем случае был получен только в 1974 году Хопкрофтом (Hopcroft) и Тарьяном (Tarjan).

Далее я покажу, что неэффективные алгоритмы, разработанные ранее, требовали $O(|V|^3)$ времени и даже больше.

Один из способов улучшить ситуацию — использовать стратегию «разделяй и властвуй» для деления исходных графов на более мелкие подграфы, которые можно проверять по отдельности.

Это возможно благодаря следующим двум леммам:

- граф является планарным тогда и только тогда, когда планарны все его связанные компоненты;
- граф является планарным тогда и только тогда, когда планарны все его двусвязные компоненты.

В главе 14 мы уже дали определение *связного графа*: G считается связным, если из любой вершины $v \in G$ можно найти путь в любую другую вершину $u \in G$. Если граф несвязный, мы можем определить связность его компонент как максимальных непересекающихся подграфов в G, которые являются связными.

Двусвязный граф — это связный граф, обладающий дополнительным свойством: не существует ни одной вершины $v \in G$ такой, что удаление v из G приведет к нарушению связности графа. Можно дать эквивалентное определение двусвязного графа: G считается двусвязным, если между любой парой вершин $u, v \in G$

[1] Первое условие можно проверить за постоянное время, если есть информация о размере графа.

существует два непересекающихся пути, то есть не имеющих других общих ребер или вершин, кроме u и v.

Доказательство первой леммы тривиально. Для несвязного графа, поскольку между его связными компонентами нет ребер, достаточно нарисовать каждую компоненту так, чтобы он не пересекался с другими.

Как следствие из двух лемм, любой граф G можно разбить на его двусвязные компоненты и применить проверку планарности к каждой из них в отдельности.

15.2.3. Наивный алгоритм проверки планарности

Выше неоднократно упоминалось, что существует (неэффективный) алгоритм, основанный на теореме Куратовского, который довольно просто реализовать. Давайте начнем именно с него. В листинге 15.1 показан шаблонный метод, служащий оберткой для любого алгоритма проверки планарности, обеспечивающий разделение графа на связные (или, что еще лучше, двусвязные) компоненты и выполняющий проверку каждой из них.

В главе 14 было показано, как найти связные компоненты графа, используя поиск в глубину; поиск двусвязных компонент немного сложнее, но его все же можно сделать с помощью модифицированной версии DFS[1].

Листинг 15.1. Шаблон для применения алгоритма проверки планарности

Метод planarityTesting — это шаблонная метафункция. Она принимает граф и алгоритм проверки планарности и применяет алгоритм ко всем двусвязным компонентам графа. Если какая-либо из них не проходит проверку на планарность, возвращается признак непланарности графа

```
function planarityTesting(graph, isPlanar)
  components ← biconnectedComponents(graph)
  for G in components do
    if not isPlanar(G) then
      return false
  return true
```

Разбиваем граф graph на двусвязные компоненты (для простоты можно использовать связные компоненты)

Цикл по компонентам графа

Если какая-то компонента непланарная, то и весь граф непланарный

Если все компоненты планарные, то вернуть true

Теперь определим фактический метод, выполняющий проверку планарности каждой двусвязной (или связной) компоненты. В листинге 15.2 показан метод, основанный на теореме Куратовского.

Алгоритм использует индуктивное определение графа. Деревья можно определить индукцией по количеству вершин (мы строим более крупные деревья, добавляя потомков к корню), для данного графа $G = (V, E)$ его можно индуктивно вырастить двумя разными способами.

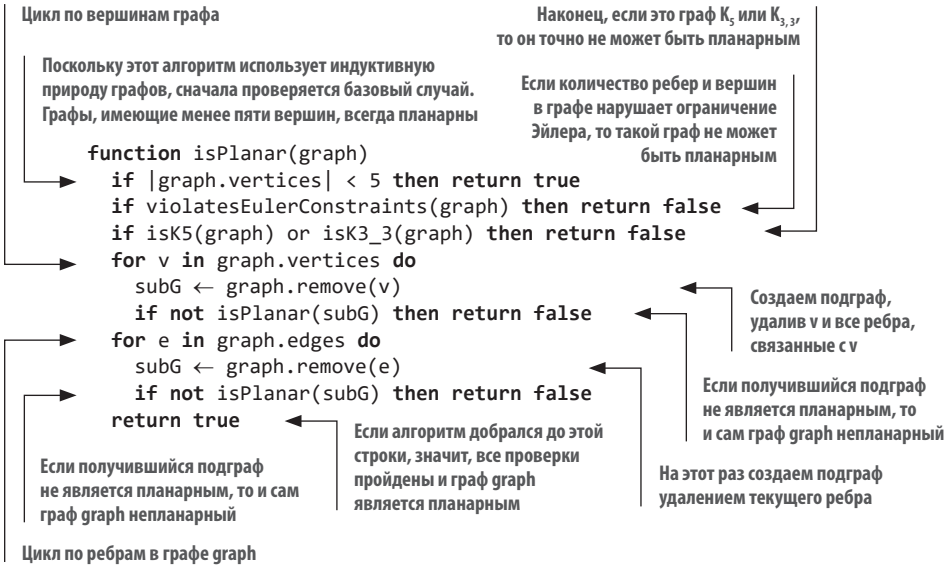
---

[1] Я не буду приводить описание этого алгоритма для экономии места, но вы можете найти его в Интернете: https://en.wikipedia.org/wiki/Biconnected_component.

Как было показано в главе 14, на самом деле можно сделать так:

- $G' = (V + \{v\}, E)$: добавляется новая вершина;
- $G' = (V, E + \{(u, v)\}) \mid u, v \in V$: добавляется новое ребро.

Листинг 15.2. Проверка планарности на основе теоремы КК



Когда дело доходит до разложения G, нужно рассмотреть два набора подграфов:

- *индуцированные подграфы* — все графы, которые можно получить, удаляя из G каждую вершину по отдельности (правило индукции 1) и все ребра, связанные с удаляемой вершиной;
- *остовные подграфы* — все графы, которые можно получить, удаляя из G каждое ребро по отдельности (правило индукции 2).

Эти два множества графов рекурсивно проверяются в строках 5–7 и 8–10 соответственно. Поскольку алгоритм является рекурсивным, нужен некоторый базовый случай: можно, конечно, в роли такого случая использовать пустой граф, но мы знаем, что все графы с четырьмя вершинами или менее являются планарными, и имеем полное основание остановить рекурсию раньше (строка 2) и сэкономить вычислительные ресурсы.

Осталось только проверить, является ли текущий входной граф непланарным. Для этого используется другой базовый случай (на самом деле их два) в строке 4, проверяющий — привела ли рекурсия к появлению K_5 или $K_{3,3}$ (мы знаем, что, согласно теореме Куратовского, это означает непланарность). В данном примере, однако, добавлена еще одна проверка в строке 3, в ней использовано неравенство Эйлера: как

мы видели в подразделе 15.2.2, если исследуемый граф имеет слишком много ребер для своих вершин, он не может быть планарным.

Реализации служебных методов, выполняющих эти проверки, приводятся в листингах 15.3, 15.4 и 15.5.

Листинг 15.3. Вспомогательные методы проверки планарности: инварианты Эйлера

Проверяет соблюдение графом условий планарности

Временные переменные с числом вершин и ребер в графе graph

```
function violatesEulerConstraints(graph)
  (n,m) ← (|graph.vertices|, |graph.edges|)
  if m > 3 × n - 6 then
    return true
  if not hasCycleOfLength3(graph) and m > 2 × n - 4 then
    return true
  return false
```

Первое ограничение: планарные графы соответствуют условию $|E| \leq 3|V| - 6$

Второе (более строгое) условие: планарные графы, не имеющие петель длиной 3, соответствуют условию $|E| \leq 2|V| - 4$

Метод проверки ограничений Эйлера в листинге 15.3 напрямую выводится из формул в подразделе 15.2.2. Самое сложное — проверить отсутствие в графе петель длиной 3: это можно сделать с помощью модифицированной версии DFS, возвращающей все петли за линейное время $O(V + E)$. Поскольку это довольно дорого и требует нетривиальных усилий для разработки и сопровождения кода, польза от включения этой второй проверки является спорной, и — особенно в первой попытке — она может того не стоить: проще ограничиться проверкой только первого условия и убрать инструкцию `if` в строке 5.

Листинг 15.4. Вспомогательные методы проверки планарности: проверка K5

Метод isK5 принимает граф и проверяет, является ли он полным графом с пятью вершинами

```
function isK5(graph)
  if |graph.vertices| == 5 and |graph.simpleEdges| == 10 then
    return true
  return false
```

Граф K5 должен иметь пять вершин и ровно десять ребер (не считая петель). Здесь предполагается, что graph.simpleEdges возвращает все ребра в графе, за исключением возможных петель

В листинге 15.4 показан метод, проверяющий, является ли граф (изоморфным) графом K_5. Очевидно, что у него должно быть пять вершин (!), а также ровно десять ребер. Обратите внимание, что здесь мы говорим о простых ребрах, когда исходящая и входящая вершины различны (поэтому нет петель).

В листинге 15.5 показан метод, проверяющий, является ли граф (изоморфным) графом $K_{3,3}$. Это немного сложнее, потому что недостаточно проверить количество вершин (шесть) и ребер (девять), также нужно убедиться, что граф двудольный и что оба раздела имеют размер 3.

Листинг 15.5. Вспомогательные методы проверки планарности: проверка $K_{3,3}$

Метод isK3_3 принимает граф и проверяет, является ли он полным двудольным графом, в каждом из двух разделов которого имеется по три вершины

Временные переменные для хранения количества вершин и ребер

Граф $K_{3,3}$ имеет шесть вершин и ровно девять ребер (не считая петель): здесь предполагается, что graph.simpleEdges возвращает все ребра в графе, исключая петли

```
function isK3_3(graph)
  (n,m) ← (|graph.vertices|, |graph.simpleEdges|)
  if n == 6 and m == 9 then
    if isBipartite(graph) and partitionsSize(graph) == (3,3) then
      return true
  return false
```

Однако этого недостаточно. Нужно также убедиться, что граф является двудольным и два его раздела имеют правильные размеры

Это подводит нас к последнему шагу, который нужно реализовать: выяснить, является ли граф двудольным, и получить анализ двух его разделов.

Используем свойство двудольных графов: граф является двудольным тогда и только тогда, когда возможно раскрасить его вершины в два разных цвета и при этом не окажется пары смежных вершин одного цвета. На рис. 15.10 показано несколько примеров, поясняющих это определение.

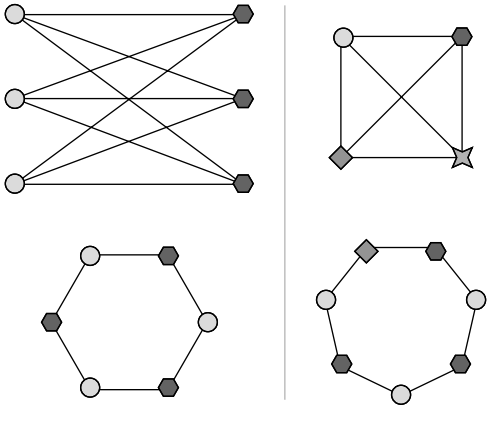


Рис. 15.10. Двудольные графы и раскраска. Все графы раскрашены минимально возможным количеством цветов. *Слева* два примера двудольных графов; *справа* недвудольные графы. Как видите, для недвудольных графов двух цветов недостаточно (здесь также использованы разные формы для представления вершин, чтобы ярче выразить разницу)

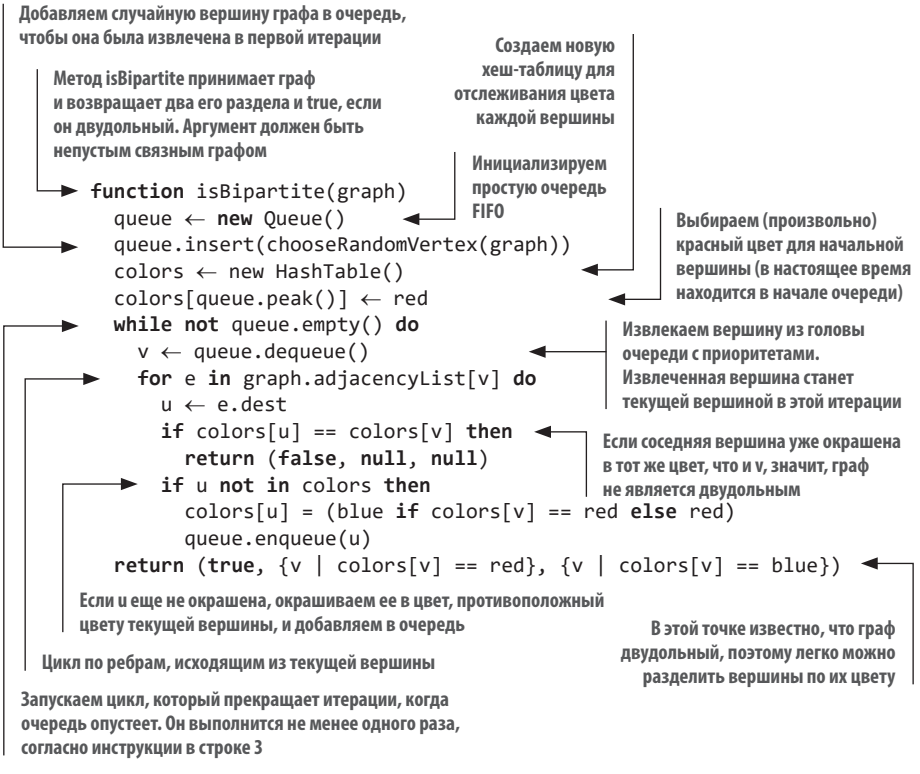
Мы легко можем раскрасить граф, изменив алгоритм BFS.

1. Представим начальную вершину в виде красного шестиугольника.
2. После извлечения очередной вершины из очереди окрашиваем всех ее соседей в противоположный цвет: например, в синие квадраты, если текущая вершина — красный шестиугольник, и наоборот.
3. Если любая из соседних вершин уже окрашена в тот же цвет, что и текущая вершина, то граф не является двудольным.

В листинге 15.6 показан псевдокод метода, возвращающего два раздела при проверке двудольности графа. Желающие могут взглянуть на его реализацию на Java,

доступную в репозитории книги на GitHub[1], или на реализацию на JavaScript в библиотеке JsGraph, которая использовалась для рисования большинства примеров представления графов в этой главе.

Листинг 15.6. Проверка двудольности связного графа



15.2.4. Увеличение производительности

Выше приведена основная часть простейшего алгоритма, проверяющего планарность графа. Мы знаем, что он неэффективен, но насколько именно?

Это рекурсивный алгоритм, в котором каждый вызов метода сопровождается несколькими рекурсивными вызовами; глубина рекурсии линейна и зависит от размера исходного графа $G = (V, E)$, потому что за один раз удаляется одна вершина или одно ребро.

Ширина дерева рекурсии (количество рекурсивных вызовов) тоже линейна и зависит от размера графа, который в данный момент обрабатывается, назовем его $G' = (V', E')$. Такое явление связано с тем, что два цикла `for` обходят все вершины и все ребра в G'.

[1] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#graph.

Запишем формулу, отражающую время работы. Если $|V| = n$ и $|E| = m$, то:

```
T(n,m) = n × T(n-1, m) + m × T(n, m-1)
T(0,0) = 1
T(0,×)=1
T(×,0)=1
```

Для рекуррентного соотношения[1] вида $T(n) = n \times T(n-1)$ решение зависит от значения базового члена. Если, как в данном случае, $T(0) = 1$, то $T(n) = n!$ — это, по крайней мере, когда есть одна переменная. В нашем конкретном случае с двумя переменными значение растет еще быстрее.

Факториал — функция, которая растет быстрее, чем экспоненциальная функция, — крайне нежелательное явление в формуле времени работы алгоритма. Ведь его присутствие означает, что алгоритм, представленный в листинге 15.2, можно использовать только для очень небольших графов.

Встречая факториалы в своих алгоритмах, осознавайте, что они выполняют одни и те же вычисления снова и снова несколько раз.

Это относится и к нашему алгоритму. Рассмотрим небольшой начальный граф, насчитывающий всего три вершины. Пусть он будет таким $G = (\{1, 2, 3\}, E)$. Ради примера не будем заморачиваться с его ребрами, а сосредоточимся на рекурсии по вершинам. Цикл `for` в строке 4 листинга 15.2 выполнит три вызова[2] для следующих графов[3]:

```
(V',E')=({2,3}, E-{1}), ({1,3}, E-{2}), ({1,2}, E-{3})
```

Для каждого из этих графов будет выполнено два вызова:

```
({3}, E-{1,2}), ({2}, E-{1,3}),
({3}, E-{1,2}), ({1}, E-{2,3}),
({2}, E-{1,3}), ({1}, E-{2,3})
```

Как видите, во втором раунде вызовов каждый граф появляется дважды. Если также рассмотреть рекурсию по ребрам, картина станет намного хуже.

Обычно такой рост заканчивается сбоем из-за нехватки памяти задолго до того, как рекурсия приблизится к базовым случаям.

---

[1] Рекуррентное соотношение — это уравнение, рекурсивно определяющее последовательность значений, где каждый член последовательности определяется как функция предыдущих членов. Дополнительную информацию о решении рекуррентных соотношений ищите в приложении Б.

[2] Представьте для простоты, что условие в строке 2 останавливает рекурсию, только если получен пустой граф.

[3] Под сокращенной записью $E - \{1\}$ подразумевается $E - \{(1, v) \mid v \in \{2, 3\}, (1, v) \in E\}$. То же относится к другим вершинам.

Наиболее распространенная стратегия решения этой проблемы состоит в том, чтобы избежать повторяющихся вычислений одним из следующих способов:

- определить лучшую рекурсию без повторных вычислений (не всегда возможно);
- сократить дерево поиска, чтобы избежать повторной работы (алгоритмы *ветвей и границ*);
- вычислять и сохранять результаты решения меньших задач и обращаться к ним при вычислении более крупных задач (алгоритмы *динамического программирования*).

Для нашего случая разумнее выбрать третий вариант и использовать прием *мемоизации*[1], чтобы обеспечить кэширование результатов решения меньших задач.

Такой прием даст некоторое улучшение, но, как будет показано далее, наиболее эффективные алгоритмы проверки планарности вместо него предполагают добавление или удаление ребра на каждом шаге так, чтобы гарантировать линейное число шагов (соответственно, они используют первую из стратегий в списке).

Избегая повторных вычислений, мы гарантируем, что проверим каждый отдельный подграф не более одного раза, поэтому количество шагов будет ограничено количеством возможных подграфов. Для графа с n вершинами и m ребрами существует 2^n *порожденных подграфов* (поскольку существует 2^n подмножеств вершин) и 2^m остовных подграфов (с учетом подмножества ребер). Таким образом, общее количество возможных подграфов ограничено величиной 2^{n+m}, что лучше, чем факториал, но все же слишком велико, чтобы считать алгоритм пригодным для обработки графов с числом вершин больше 20.

Есть и другие небольшие оптимизации, которые можно применить. Они не настолько эффективны, как избавление от повторного выполнения одной и той же работы, но все же способствуют ускорению алгоритма. Например, можно оптимизировать условие остановки. Не нужно ждать, пока опустимся до уровня K_5 или $K_{3,3}$; поводом для остановки может служить любой полный граф с пятью и более вершинами или любой полный двудольный граф с двумя разделами, имеющими три вершины или более, заведомо непланарный.

Однако большинство этих случаев уже улавливаются инвариантами Эйлера.

В листинге 15.7 показан псевдокод оптимизированного метода; реализацию на Java можно найти в репозитории книги[2], а реализацию на JavaScript — в библиотеке JsGraph.

---

[1] Использование рекурсии и приема мемоизации для предотвращения переполнения стека описывается в приложении Д.

[2] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#graph.

Листинг 15.7. Проверка планарности с кэшированием

```
function isPlanar(graph, cache={})
  if graph in cache then return (cache[graph], cache)
  if |graph.vertices| < 5 then return (true, cache)
  if violatesEulerConstraints(graph) then
    cache[graph] ← false
    return (false, cache)
  if isComplete(graph) or isNonPlanarCompleteBipartite(graph) then
    cache[graph] ← false
    return (false, cache)
  for v in graph.vertices do
    subG ← graph.remove(v)
    (planar, cache) ← isPlanar(subG, cache)
    if not planar then
      cache[graph] ← false
      return (false, cache)
  for e in graph.edges do
    subG ← graph.remove(e)
    (planar, cache) ← isPlanar(subG, cache)
    if not planar then
      cache[graph] ← false
      return (false, cache)
  cache[graph] ← true
  return (true, cache)
```

Вместо проверки на соответствие случаю K_5 или $K_{3,3}$ проверяется только полнота (поскольку полный граф, имеющий не менее пяти вершин, определенно непланарный) или полнота и двудольность (в этом случае также нужно убедиться, что наименьший из двух разделов имеет размер 3)

Если количество ребер и вершин нарушает ограничения Эйлера для планарных графов, то можно с уверенностью утверждать, что граф не является планарным. Перед возвратом нужно обновить кэш, чтобы этот граф не проверялся снова в другой ветке вычислений

Сначала проверяем — есть ли граф в нашем кэше. Роль кэша может играть, например, такая простая структура данных, как словарь, если графы сериализуемы

Выполняя рекурсивный вызов, необходимо передать накопленный кэш. Во всем остальном, кроме кэша, алгоритм остается прежним

Цикл по вершинам графа

Цикл по всем ребрам в графе. Здесь применяется тот же шаблон, что и при обходе вершин

Оказавшись в этой точке, можно быть уверенными, что граф планарный; теперь нужно обновить кэш и вернуть true

15.2.5. Эффективные алгоритмы

Алгоритм, представленный в листинге 15.7, слишком медленный, чтобы его можно было применять к большим графам. Тем не менее он может оказаться приемлемым и недорогим[1] вариантом, когда требуется обрабатывать небольшие графы.

Однако существуют гораздо лучшие алгоритмы проверки планарности. Они используют разные подходы к получению ответа (и планарного представления) за линейное время, но у всех у них есть одно общее качество: они довольно сложны.

Например, «научные работы, занимающие десятки страниц» сложны. Их подробное описание выходит за рамки этой главы, но мы кратко коснемся некоторых наиболее важных из них и дадим ссылки заинтересованным читателям[2].

---

[1] В смысле усилий, необходимых для реализации и сопровождения кода.

[2] Полный обзор можно найти в статье: *Patrignani M.* Planarity Testing and Embedding // Handbook of Graph Drawing and Visualization. Chapman and Hall/CRC, 2013. — P. 1–42.

Имейте в виду, что для реализации этих алгоритмов может потребоваться приложить значительные усилия. Учитывайте также, что если предъявляемые ограничения можно ослабить и принять алгоритм, создающий более или менее разумное представление, пусть и не планарное, то можно использовать более простые эвристики для создания представлений (подробнее об этом поговорим ниже).

Как уже упоминалось, первый линейный алгоритм проверки планарности был разработан в 1974 году Хопкрофтом (Hopcroft) и Тарьяном[1] (Tarjan) с целью оптимизации ранее разработанного[2] варианта $O(|V|^2)$. Их идея основана на добавлении вершин, поэтому алгоритм начинает работу снизу вверх, сохраняя возможные планарные вложения для каждого индуцированного подграфа[3], построенного инкрементно.

Как упоминалось выше, эта стратегия определяет другой подход к рекурсии: восходящий, а не нисходящий, пошаговый, а не по принципу «разделяй и властвуй». Но прежде всего реконструируется исходный граф по одной вершине за раз, в этом подходе не требуется анализировать все подграфы, поэтому он выполняет только линейное количество шагов.

Суть в том, что при добавлении вершин алгоритм отслеживает возможные представления подграфа.

Перенесемся в 2004 год, когда Бойер (Boyer) и Мирвольд (Myrvold) разработали совершенно новый подход[4]: метод сложения ребер, постепенно добавляющий ребра вместо вершин. Он по-прежнему имеет линейное время выполнения, $O(|V| + |E|)$, но у него есть большое преимущество: этот метод позволяет избежать каких-либо требований к структурам данных, используемым для хранения представлений-кандидатов. Упомянутый алгоритм в настоящее время является одним из самых современных решений для поиска планарных представлений планарных графов; самое замечательное, что на boost.org можно найти его реализацию с открытым исходным кодом: http://mng.bz/5jj7.

Наконец, я хочу упомянуть еще один современный алгоритм из области проверки планарности[5] — Фрайссейкса (Fraysseix), де Мендеса (de Mendes) и Розенштиля (Rosenstiehl). Он характеризует плоские графы с точки зрения упорядочения ребер слева направо в дереве поиска в глубину, и его реализация основана на поиске в глубину, хотя очевидно, что этот метод поиска нуждается в соответствующей модификации. А еще он использует *деревья Тремо*, специальные остовные деревья, описывающие порядок посещения вершин в графе алгоритмом DFS.

---

1 *Hopcroft J., Tarjan R. E.* Efficient planarity testing // Journal of the Association for Computing Machinery, 1974. — № 21 (4). — P. 549–568.

2 *Tarjan R. E.* Implementation of an efficient algorithm for planarity testing of graphs. Неопубликованная реализация, декабрь 1969.

3 Как мы уже видели, индуцированный подграф получается удалением одной или нескольких вершин из исходного графа.

4 *Boyer J. M., Myrvold W. J.* On the cutting edge: simplified O(n) planarity by edge addition (PDF) // Journal of Graph Algorithms and Applications, 2004. — № 8 (3). — P. 241–273.

5 *Fraysseix de H., Ossona de Mendez P.; Rosenstiehl P.* Trémaux trees and planarity // International Journal of Foundations of Computer Science, 2006. — № 17 (5). — P. 1017–1029.

15.3. НЕПЛАНАРНЫЕ ГРАФЫ

Теперь, имея по крайней мере один алгоритм проверки планарности, можно с бóльшим оптимизмом смотреть на задачу визуализации графов на экране. Некоторые из алгоритмов проверки планарности также выводят планарное представление, что дает хорошую отправную точку.

И все же впереди у нас с вами еще долгий путь.

Начнем с двух вопросов.

- Можно ли считать уменьшение количества пересекающихся ребер единственным критерием, которому необходимо следовать?
- Как известно, не все графы плоские... достаточное ли это основание, чтобы просто отказаться от визуализации непланарных графов?

Сосредоточимся на первом вопросе. Что вы думаете? Потратьте минуту, чтобы вообразить, какие еще характеристики делают визуализацию хорошей и наоборот.

Затем взгляните на рис. 15.11, чтобы подтвердить свои мысли.

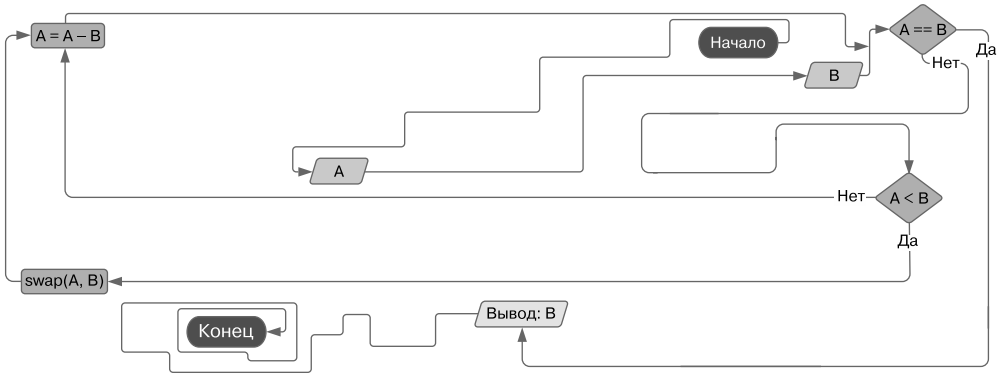


Рис. 15.11. Еще одно безумное представление блок-схемы с рис. 15.2

Здесь можно заметить как минимум три разных аспекта, не позволяющих считать эту визуализацию хорошей.

1. Наиболее очевидный недостаток — невозможно прочитать текст надписей. Это связано с тем, что элементы расположены излишне далеко друг от друга, и нужно уменьшить масштаб, чтобы увидеть всю диаграмму.
2. Некоторые ребра искривлены и извилисты, что затрудняет движение по ним взглядом.
3. По сравнению с рис. 15.2 относительное расположение элементов затрудняет определение направления потока. Смежные узлы находятся далеко друг от друга, а между ними находятся другие узлы, представляющие не связанные с ними шаги.

Это были основные недостатки диаграммы на рис. 15.11. Можно ли формализовать эти соображения в требования для улучшения визуализации? Давайте попробуем.

1. Смежные вершины (вершины, соединенные ребром) должны располагаться как можно ближе друг к другу на плоскости. Конечно, здесь нужен компромисс, потому что располагать вершины слишком близко друг к другу тоже нехорошо (иначе они будут перекрывать друг друга и ребро, соединяющее их). Кроме того, если вершина v смежна с множеством других вершин, нельзя допускать размещения слишком большого количества таких вершин вокруг v.
2. Ребра должны изображаться самым простым способом: либо прямыми отрезками, либо дугами эллипсов.
3. Количество пересечений ребер должно быть минимальным. Ребра планарных графов вообще не должны пересекаться, а для непланарных графов таких пересечений должно быть как можно меньше.

В следующих главах я покажу, как эти требования преобразовать в математические выражения для моделирования *функции стоимости*, отражающей качество представления графа.

А в оставшейся части этой главы подробно разберем третий пункт.

Как вы видели выше, существуют графы, которые невозможно изобразить без пересечения ребер.

ПЕРЕСЕЧЕНИЯ РЕБЕР В МНОГОМЕРНЫХ ПРОСТРАНСТВАХ

Интересно отметить, что если перейти из плоского пространства в трехмерное, то найти такое представление в $\mathbb{R}^3$, в котором ребра не пересекаются, становится тривиальной задачей. Рассмотрим следующую жорданову кривую $C(t)$ — поверхность, определяемую как:

$$C(t) = \begin{cases} x = t; \\ y = t^2, t \geq 0; \\ z = t^3. \end{cases}$$

Здесь каждую вершину можно отобразить в отдельную точку на кривой и нарисовать ребра как отрезки, соединяющие вершины. Можно также доказать невозможность выбора четырех точек из $C(t)$, которые лежат в одной плоскости, и, следовательно, отрезки между парами точек не могут пересекаться.

Однако для непланарных графов G можно определить значение, называемое *числом пересечений (ЧП)*, то есть значение, отражающее наименьшее возможное количество пересечений ребер из всех возможных представлений графа G в $\mathbb{R}^2$.

Планарные графы, очевидно, имеют число пересечений, равное 0; оба непланарных графа, которые мы видели выше в этой главе, K_5 и $K_{3,3}$, имеют число пересечений, равное 1.

15.3.1. Поиск числа пересечений

Теорема Куратовского перечисляет необходимые и достаточные условия планарности графа, но она мало чем может помочь при вычислении минимального числа пересечений непланарных графов. Задача поиска числа пересечений для непланарного графа исследована гораздо хуже, чем проверка планарности. Несмотря на существование нескольких эффективных алгоритмов построения планарных представлений для планарных графов, в настоящее время не существует эффективного алгоритма, способного найти минимальное число пересечений графа в общем случае.

На самом деле доказано, что определение числа пересечений в общем случае является *NP-полной* задачей.

Однако есть заметные исключения[1], работающие в случаях, если сузить поле. Например, недавно было доказано[2], что существует простой алгоритм проверки числа пересечений (равного 1) в непланарном графе.

Если предположить, что G — непланарный граф (и, значит, имеет не менее пяти вершин, как следствие теоремы Куратовского), то для каждой пары несмежных[3] ребер $a \to b$, $c \to d$ можно удалить оба ребра и добавить новую вершину v и четыре новых ребра: $a \to v$, $v \to b$, $c \to v$, $v \to d$.

Если вновь созданный граф является планарным, то число пересечений исходного графа равно 1.

Некоторые из наиболее интересных результатов в этой области сосредоточены на полных графах и полных двудольных графах. Гипотезы Гая и Заранкевича определяют формулу для числа пересечений этих графов, но они пока не доказаны.

Гипотеза Гая предполагает, что минимальное число пересечений некоторого полного графа с n вершинами определяется выражением:

$$Z(K_n) = \frac{1}{4}\left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor \left\lfloor \frac{n-2}{2} \right\rfloor \left\lfloor \frac{n-3}{2} \right\rfloor.$$

---

1 *Clancy K., Haythorpe M., Newcombe A.* A survey of graphs with known or bounded crossing numbers. arXiv preprint arXiv:1901.05155 (2019).

2 *Haythorpe M.* QuickCross — Crossing Number Problem.

3 Два ребра называют смежными, если они имеют хотя бы одну общую вершину; здесь говорится о паре несмежных ребер, соответственно, все четыре вершины различны.

Гипотеза Заранкевича дает оценку для полных двудольных графов с двумя разделами, имеющими n и m вершин соответственно:

$$Z\left(K_{n,m}\right)=\frac{1}{4}\left\lfloor\frac{n}{2}\right\rfloor\left\lfloor\frac{n-1}{2}\right\rfloor\left\lfloor\frac{m}{2}\right\rfloor\left\lfloor\frac{m-1}{2}\right\rfloor.$$

В настоящее время доказано, что обе формулы определяют верхние границы, а это означает, что число пересечений для этих графиков не больше значения, вычисленного с использованием формул, но предположение о том, что они определяют нижние границы, пока не опровергнуто.

Применив эти формулы к двум уже знакомым нам графам, получим:

$$Z\left(K_5\right)=\frac{1}{4}\cdot\left\lfloor\frac{5}{2}\right\rfloor\cdot\left\lfloor\frac{4}{2}\right\rfloor\cdot\left\lfloor\frac{3}{2}\right\rfloor\cdot\left\lfloor\frac{2}{2}\right\rfloor=1;$$

$$Z\left(K_{3,3}\right)=\left\lfloor\frac{3}{2}\right\rfloor\cdot\left\lfloor\frac{2}{2}\right\rfloor\cdot\left\lfloor\frac{3}{2}\right\rfloor\cdot\left\lfloor\frac{2}{2}\right\rfloor=1.$$

Итак, для K_5 и $K_{3,3}$ ожидания согласуются с упомянутым выше и с нашим опытом. На самом деле доказано, что гипотеза Гая верна как точное значение для $n \le 12$, а гипотеза Заранкевича — для $n, m \le 7$.

Если взять для примера граф K_6, показанный на рис. 15.12, то ожидаемое (и доказанное) его число пересечений равно 3.

И все же получить представление с минимальным количеством пересечений ребер значительно сложнее. Пример представления показан на рис. 15.13.

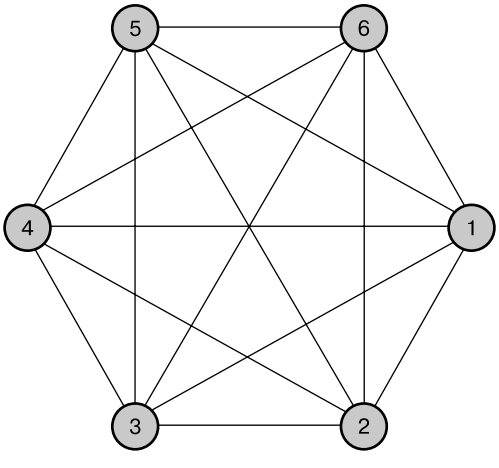


Рис. 15.12. Наивное представление K_6: здесь имеется 15 пересекающихся пар ребер

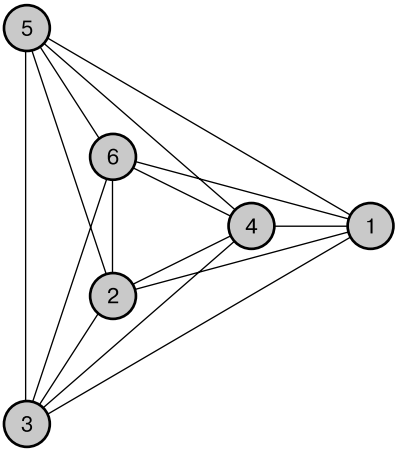


Рис. 15.13. Представление K_6 с минимальным числом пересечений: здесь имеется три пересечения

15.3.2. Количество прямолинейных пересечений

Заметили ли вы, что до сих пор мы рисовали графы только с использованием прямых отрезков? Этот подход прекрасно служил нашей цели, и можно было рисовать графы с минимальным количеством пересечений, но это не всегда верно.

Фактически сейчас стоит ввести новое определение: *число прямолинейных пересечений графа G* (ЧПП) — минимальное число ребер, пересекающихся на представлении графа *G* с ребрами, изображенными в виде прямых отрезков.

Ограничиваясь ребрами в виде прямых отрезков, мы ограничиваем число возможных представлений. Поэтому неудивительно, что число прямолинейных пересечений ребер графа никогда не меньше его числа пересечений.

А может ли быть иначе? Доказано, что любой граф *G* с числом пересечений, меньше или равным 3, можно изобразить с прямолинейными ребрами, количество которых совпадает с минимальным числом пересечений: другими словами, всякий раз, когда число пересечений `cr(G)` равно 3 или меньше, оно соответствует числу прямолинейных пересечений, `rcr(G)`.

Это, конечно, здорово, потому что означает, что планарные графы можно изобразить в виде прямых или кривых линий. Но результат нельзя обобщить, на самом деле существуют графы, для которых `cr(G)` = 4 < `rcr(G)`. На рис. 15.14 показан пример, использованный для доказательства в статье[1] 1993 года. Слева можно видеть прямолинейный рисунок графа с 12 пересечениями ребер. Это число совпадает с числом прямолинейных пересечений графа, и, как бы ни перемещали вершины, мы не сможем получить меньше пересечений при условии, что ребра изображаются только как прямые отрезки.

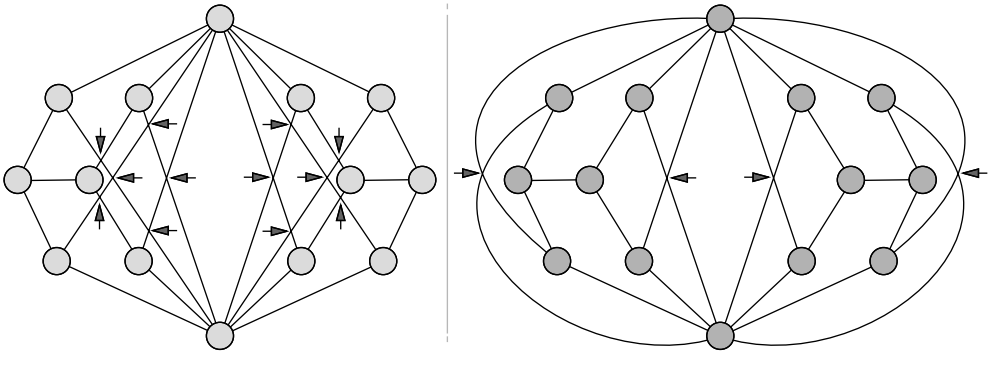


Рис. 15.14. Простейший граф, для которого число прямолинейных пересечений больше числа пересечений. Пересечения отмечены маленькими стрелками

[1] *Bienstock D., Dean N.* Bounds for rectilinear crossing numbers // Journal of Graph Theory. 1993. — 17.3 — P. 333–348.

С другой стороны, *справа* на рис. 15.14 показано, что, использовав кубические кривые Безье и переместив несколько ребер к внешней грани первого представления, можно уменьшить количество пересечений до 4, что совпадает с числом пересечений для этого графа.

Интересно отметить, что, используя такой граф в качестве отправной точки, можно строить графы с числом пересечений, равным 4, но с произвольно большим числом прямолинейных пересечений (потенциально вплоть до бесконечности). Доказательство и правила построения можно найти в оригинальной статье.

Для полных графов известно, что при $n = 10$ величина $\texttt{rcr}(K_n) > \texttt{cr}(K_n)$. К сожалению, мы не можем улучшить примеры для K_5 и K_6: даже используя жордановы кривые, лучшее, чего можно добиться, — создать представление для K_6, изображенное на рис. 15.15, которое по-прежнему имеет три пересечения.

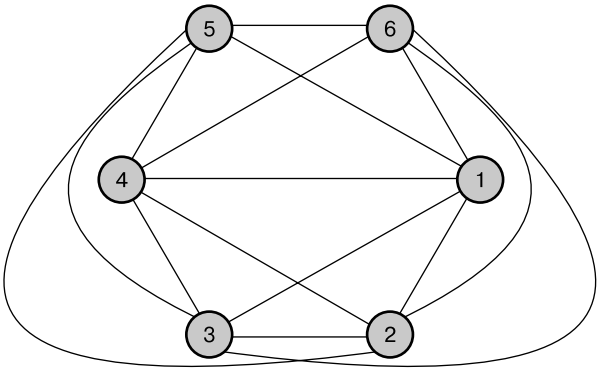


Рис. 15.15. Альтернативное представление K_6 с минимальным пересечением: использование кривых для изображения ребер позволяет сохранить более правильное расположение вершин

Тем не менее использование дуг позволяет выбрать более простое и правильное расположение вершин.

Проще говоря, есть вероятность, что, если рисовать большие графы, используя только прямолинейные отрезки, получим большее количество пересечений ребер, чем это реально возможно, потому что для многих графов число прямолинейных пересечений больше числа пересечений. Означает ли это, что необходимо отказаться от такого способа рисования графов? Ничуть!

Во-первых, как было показано выше, для изображения всех планарных графов и всех графов с числом пересечений меньше 4 можно использовать прямолинейные ребра без каких-либо ограничений; кроме того, не все остальные графы с $\texttt{cr}(G) = 4$ имеют худшее значение $\texttt{rcr}(G)$, чем $\texttt{cr}(G)$.

Оказывается, на практике нас чаще интересует изображение планарных или почти планарных графов, потому что блок-схемы, диаграммы PERT, схемы рабочих

процессов и т. д. обычно имеют вид довольно разреженных графов, которые, в свою очередь, с меньшей вероятностью будут иметь высокое число пересечений. В конце концов, из инвариантов Эйлера известно, что планарные графы должны быть разреженными (поскольку число ребер линейно зависит от числа вершин).

Во-вторых, даже притом что представление с использованием кривых может иметь меньшее количество точек пересечения, это не значит, что его легко найти. На самом деле найти такие представления с помощью алгоритма гораздо сложнее, потому что требуется оптимизировать гораздо больше параметров. Помимо положения вершин, необходимо найти идеальные кривые, уменьшающие количество пересечений. Это добавит по крайней мере один или два параметра на ребро, если использовать квадратичные или кубические кривые Безье, что сделает пространство поиска намного больше и усложнит сам поиск. На самом деле количество параметров увеличивается от $O(V)$ до $O(V + E)$, что, в свою очередь, равно $O(V^2)$ в худшем случае.

В-третьих, использование кривых требует большей вычислительной мощности. Алгоритм проверки пересечения двух прямых отрезков значительно проще, чем проверка пересечения двух кривых (и его можно сделать еще проще, наложив некоторые ограничения на положение вершин). Это означает, что даже вычисление количества пересечений в решении-кандидате обходится дороже при использовании кривых линий.

В следующих главах сосредоточимся на линейных представлениях, но также покажем, как расширить их до криволинейных.

15.4. ПЕРЕСЕЧЕНИЯ РЕБЕР

Последний фрагмент кода, который нужен, чтобы закончить первоначальный полностью рандомизированный алгоритм поиска представлений с минимальным пересечением, — метод, проверяющий, пересекаются ли два ребра.

В этом разделе начнем обсуждать вариант для представлений с прямолинейными ребрами, потому что, когда ребра изображаются прямолинейными отрезками, проще проверить, пересекаются ли они.

Затем перейдем к кривым Безье и кратко поговорим о том, как они работают, какое подмножество кривых допустимо и как проверять наличие пересечений кривых из этого подмножества.

15.4.1. Прямолинейные отрезки

Начнем с пересечения двух отрезков. Выше мы решили сосредоточиться на представлениях с прямолинейными ребрами, поэтому уделим больше времени обсуждению этого случая.

Первоначальная стратегия недорогого определения, пересекаются ли два ребра, показана на рис. 15.16. Если вокруг каждого из двух отрезков нарисовать прямоугольник со сторонами, параллельными осям, то очевидно, что два отрезка не могут пересечься,

если прямоугольники не пересекаются. В свою очередь, два прямоугольника не могут пересекаться, если проекции двух отрезков на декартовы оси не пересекаются на обеих осях *X* и *Y*.

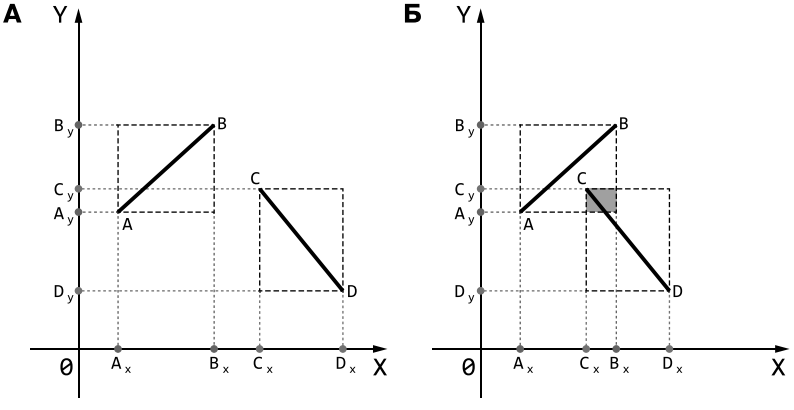


Рис. 15.16. Примеры непересекающихся отрезков: проверка ограничивающих прямоугольников может только исключить пересечение, но не подтвердить его

Это условие отсутствия пересечения проекций отрезков является достаточным, но не необходимым, чтобы можно было утверждать, что два отрезка не пересекаются. На рис. 15.16, *Б* показан случай, когда проекции отрезков пересекаются, а сами они — нет.

Итак, имеем асимметричную ситуацию:

- если проекции отрезков на оси координат не пересекаются, то можно утверждать, что отрезки не пересекаются;
- в противном случае нельзя делать никаких предположений и необходимы дополнительные проверки.

Выполнить их можно разными способами, например проверить, лежат ли крайние точки отрезка по разные стороны от линии, проходящей через другой отрезок, как показано на рис. 15.17.

Это, однако, требует слишком много возни с крайними случаями (например, с параллельными отрезками), поэтому я предпочитаю другой, более элегантный метод, который обнаружил недавно.

Суть его показана на рис. 15.18. Он связан не с ограничивающими прямоугольниками, а с поиском точки пересечения линий, проходящих через отрезки и последующей проверкой, лежит ли она внутри обоих отрезков. На самом деле возможны три случая:

- точка пересечения двух прямых *(P)* лежит вне обоих отрезков;
- *P* лежит внутри одного из двух отрезков, но не обоих;
- *P* лежит внутри обоих отрезков.

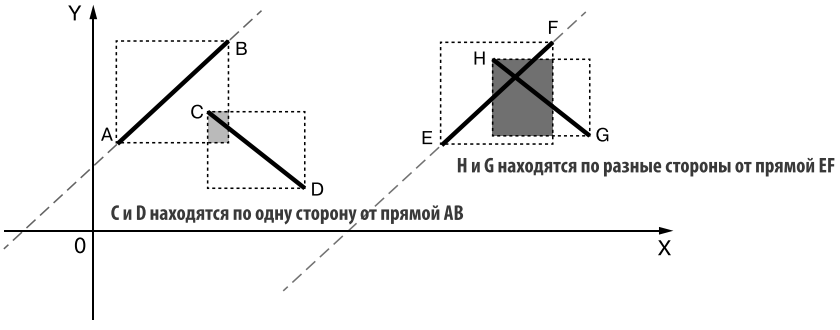


Рис. 15.17. Пересечение отрезков: если ограничивающие прямоугольники пересекаются, то необходимо проверить следующий критерий — находятся ли вершины отрезка CD (или HG) по разные стороны от прямой, проходящей через отрезок AB (EF), и наоборот

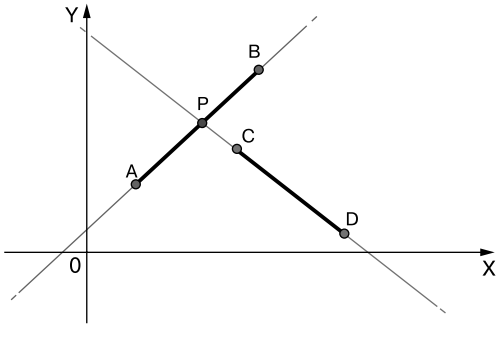


Рис. 15.18. Проверка попадания точки пересечения двух прямых в оба отрезка

Третий случай — единственный, когда пересечение имеет место.

Как найти *P*? Ну, прежде всего, использовать векторы и лучи, а не просто прямые.

Определим векторы $\mathbf{v} = BA = (B_x - A_x, B_y - A_y)$ и $\mathbf{w} = DC = (D_x - C_x, D_y - C_y)$. Эти векторы начинаются во второй точке отрезка и заканчиваются в первой. Теперь вспомните, что векторы легко можно перемещать (выполнять параллельный перенос), например, можно переместить **v** так, чтобы он начинался там, где заканчивается **w**, или наоборот, и вычислить их сумму или произведение. Рассмотрим в качестве возможного варианта вектор **u**, показанный на рис. 15.19, *A* (который мы определим чуть ниже). Здесь для удобства показано, что он имеет общую точку *A* с **v**, благодаря чему очевидно, что эти два вектора ортогональны.

Если предположить, что векторы **v** и **w** не параллельны (что легко проверить, вычислив их векторное произведение), то прямые, проходящие через них, пересекутся в одной точке *P*, которая может лежать, а может и не лежать внутри отрезков. В любом случае должны быть два действительных числа, h и g, такие, что масштабированные векторы $h \times \mathbf{v}$ и $g \times \mathbf{w}$ с начальными точками *B* и *D* соответственно заканчиваются точно в точке *P*, как показано на рис. 15.19, *Б*.

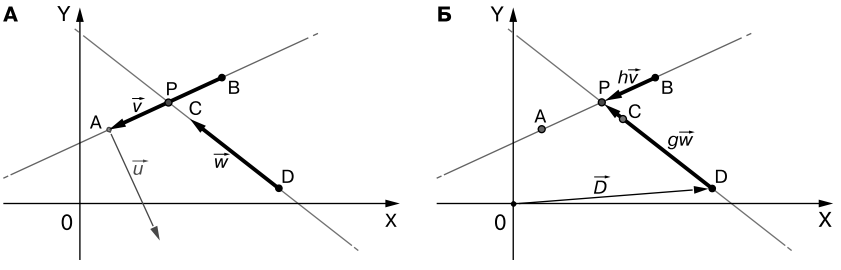


Рис. 15.19. Подход на основе векторов для проверки попадания точки пересечения двух линий в оба отрезка

Другими словами, координаты точки P могут быть выражены через оба вектора, и должно выполняться следующее уравнение:

$$B + h \times \mathbf{v} = D + g \times \mathbf{w}. \quad (1)$$

Обратите внимание, что B и D тоже можно рассматривать как векторы: в частности, векторы, идущие от начала координат к этим двум точкам.

Мы упоминали, что определим новый вектор **u**. А теперь самое время подумать:

$$\mathbf{u} = (-v_y, v_x) = (A_y - B_y, B_x - A_x).$$

Этот вектор имеет особое свойство, его скалярное произведение с вектором **v** равно нулю:

$$\mathbf{u} \times \mathbf{v} = \mathbf{v} \times \mathbf{u} = v_x v_y - v_y v_x = 0.$$

Возвращаясь к уравнению (1), мы можем умножить обе части на **u**, которое не равно `null` (пока A и B являются разными точками, и мы принимаем это в качестве гипотезы, потому что разные вершины не могут быть назначены одной и той же точке). В результате получаем:

$$B \times \mathbf{u} + h \times \mathbf{v} \times \mathbf{u} = D \times \mathbf{u} + g \times \mathbf{w} \times \mathbf{u} \Rightarrow B \times \mathbf{u} = D \times \mathbf{u} + g \times \mathbf{w} \times \mathbf{u}.$$

Таким образом, исключаем неизвестное h и решаем уравнение относительно g:

$$g = \frac{(B-D)\vec{u}}{\vec{w}\vec{u}} = \frac{(B_x - D_x, B_y - D_y)(A_y - B_y, B_x - A_x)}{(D_x - C_x, D_y - C_y)(A_y - B_y, B_x - A_x)} =$$

$$= \frac{B_x A_y - B_x B_y - D_x A_y + D_x B_y + B_y B_x - B_y A_x - D_y B_x + D_y A_x}{D_x A_y - D_x B_y - C_x A_y + C_x B_y + D_y B_x - D_y A_x - C_y B_x + C_y A_x} =$$

$$= \frac{D_x(B_y - A_y) + D_y(A_x - B_x)}{(D_x - C_x)(A_y - B_y) + (D_y - C_y)(B_x - A_x)}.$$

Аналогично можно определить вектор $\mathbf{z} = (-w_y, w_x)$, такой, что $\mathbf{z} \times \mathbf{w} = 0$, и вывести формулу для h:

$$h = \frac{(D-B)\vec{z}}{\vec{v}\vec{z}} = \frac{B_x(D_y - C_y) + B_y(C_x - D_x)}{(B_x - A_x)(C_y - D_y) + (B_y - A_y)(D_x - C_x)}.$$

Остается только рассуждать о значении этих двух решений. Можно ли, глядя на рис. 15.19, сказать, какое из значений, h или g в данном примере, больше 1?

Если вы заметили, в этом конкретном примере вектор BP меньше вектора BA, и поэтому h должно иметь значение от 0 до 1. Скаляр g должен быть больше 1, потому что вектор DP длиннее, чем DC.

В первом случае очевидно, что P находится внутри отрезка BA, а во втором — вне DC.

Когда одно из значений равно 0 или 1, это означает, что P совпадает с одной из конечных точек отрезка. Если и h, и g равны 0 или 1, то мы имеем крайний случай, то есть два отрезка имеют общую вершину. Эти два пограничных случая показаны на рис. 15.20.

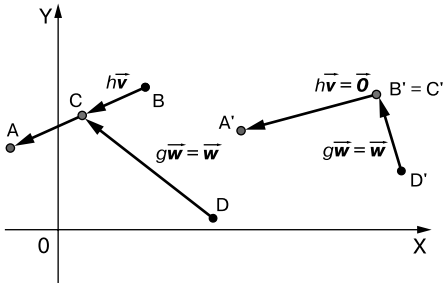


Рис. 15.20. Два крайних случая для метода на основе векторов. В первом случае для отрезков AB и CD мы имеем 0 ≤ h ≤ 1 и g == 1. Следовательно, вершина C лежит на отрезке AB. Второй пример, для отрезков A'B' и C'D' имеем h == 0 и g == 1. В этом случае две конечные точки должны быть одинаковыми и фактически B' == C'

В заключение, если предположить, что все четыре вершины различны, то два отрезка пересекаются тогда и только тогда, когда выполняются оба условия: $0 \le h \le 1$ и $0 \le g \le 1$ (но только одно из этих значений равно 0 или 1).

В листинге 15.8 показан код вычисления масштабного коэффициента (h или g, в зависимости от выбранного порядка обхода точек) для одного отрезка по отношению к другому.

Листинг 15.8. Метод vectorScalingFactor

```
function vectorScalingFactor(A, B, C, D)
  v = (B.x-A.x, B.y-A.y)
  w = (D.x-C.x, D.y-C.y)
  u = (-v.y, v.x)
  return ((B.x-D.x) × u.x + (B.y-D.y) × u.y) / (w.x × u.x + w.y × u.y)
```

Метод vectorScalingFactor принимает четыре точки, которые, как предполагается, являются конечными точками отрезков AB и CD, и возвращает коэффициент масштабирования, который необходимо применить к вектору CD, чтобы он заканчивался точно на отрезке AB

Мы повторно используем этот коэффициент в листинге 15.9, где проверяется пересечение двух ребер графа.

Листинг 15.9. Метод edgesIntersection

Метод edgesIntersection принимает два ребра и возвращает true, только и если только отрезки, представляющие ребра, пересекаются

Предполагается, что свойства source и destination возвращают точки, которые, в свою очередь, имеют поля x и y, представляющие координаты

```
function edgesIntersection(e1, e2)
  h = vectorScalingFactor(
      e1.source, e1.destination, e2.source, e2.destination)
  g = vectorScalingFactor(
      e2.source, e2.destination, e1.source, e1.destination)
  return 0 ≤ h ≤ 1 and 0 ≤ g ≤ 1 and not (h in {0,1} and g in {0,1})
```

Как мы знаем, ребра пересекаются, если оба значения, h и g, находятся в диапазоне от 0 до 1, но если оба значения равны 0 или 1, то это означает, что ребра всего лишь смежные

15.4.2. Ломаные линии

В главе 15 мы нарисовали несколько примеров блок-схем, используя ломаные линии для обозначения ребер, и, в частности, получили подмножество всех ломаных линий, все отрезки которых параллельны декартовым осям. В этом случае проверка на пересечение становится несколько проще: несмотря на то что для каждого ребра нужно проверять все образующие его отрезки, сама проверка пересечений двух отрезков, которые могут быть либо вертикальными, либо горизонтальными, становится намного проще и сводится к проверке координат их концов.

Одно важное отличие этого представления состоит в том, что количество пересечений между двумя ребрами больше не равно 0 или 1; они могут пересекаться несколько раз — еще одна причина предпочесть другие стили ломаным линиям.

15.4.3. Кривые Безье

Интересную и гибкую альтернативу ломаным линиям представляют кривые Безье. Эти кривые — ценное решение, потому что их можно строить с переменной степенью точности, в зависимости от доступных вычислительных ресурсов. Кривые Безье определяются красивой математической формулой и отличаются гибкостью и точностью. Подробное изучение этих линий выходит за рамки этой книги, но я попытаюсь дать краткое введение, которого должно быть достаточно для начала. Тем, кого заинтересует эта тема, я могу порекомендовать два бесплатных онлайн-ресурса:

- *Sederberg T. W.* Computer aided geometric design. — 2012;
- A Primer on Bézier Curves, https://pomax.github.io/bezierinfo/.

Начнем формирование нашего восприятия кривых Безье с их геометрического определения, примеры показаны на рис. 15.21.

Чтобы получить квадратичную кривую, нужны три точки: две конечные точки (S и E на рис. 15.21) и одна контрольная (C_1). Сначала нужно нарисовать два отрезка между конечными точками и C_1. Затем выбрать две точки на этих отрезках так, что

если точка на SC_1 (назовем ее A) такова, что $SA / SC = t$, то B, конечная точка на EC_1, должна удовлетворять условию $EB / EC_1 = (1 - t)$. Эти две точки, A и B, будут концами другого отрезка AB, на котором нужно выбрать точку P_t такую, что $AP_t / AB = t$.

Если $t == 0$, то $P_t == A$, а когда $t == 1$, то $P_t == B$. Совокупность всех этих точек P_t для всех значений t от 0 до 1 образует квадратичную кривую Безье между A и B.

Обратите внимание, что отрезок A_tB_t всегда касается кривой.

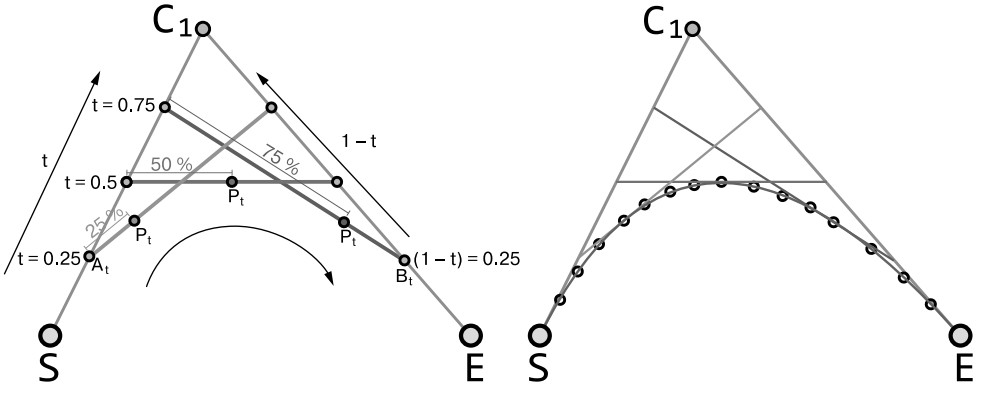


Рис. 15.21. Как рисуется квадратичная кривая Безье. Чтобы получить квадратичную кривую, нужны три точки: две конечные точки (S, E) и одна контрольная (C_1, или просто C для ясности в дальнейшем). Сначала нужно нарисовать два отрезка между конечными точками и C. Затем нарисовать отрезки, конечные точки которых лежат на SC и EC, при этом, если первая конечная точка, назовем ее A, такова, что SA / SC = t, то B, вторая конечная точка, должна удовлетворять условию EB / EC = (1 – t). Наконец, на отрезке AB выбираем точку P так, чтобы выполнялось условие AP / AB = t

Процедура построения кубических кривых Безье несколько сложнее, как показано на рис. 15.22. В этом случае есть две контрольные точки, C_1 и C_2, и сначала нужно нарисовать отрезки C_1C_2 и выбрать на этом отрезке точку C_t такую, чтобы выполнялось соотношение между двумя отрезками, созданными этой точкой, $t = C_1C_t / C_1C_2$.

Затем выбираются две точки, A_t и B_t, на отрезках между конечными точками и ближайшей контрольной точкой такие, что $SA_t / SC_1 = t$ и $C_2B_t / C_2E = t$.

Наконец, для трех точек, A_t, B_t и C_t, нужно выполнить те же действия, что и для квадратичной кривой, и выбрать точку P_t.

Технически кривая Безье формируется в итеративном процессе линейной интерполяции. Выбирая точки A_t, B_t (и C_t), мы применяем линейную интерполяцию (изменяя отношение t), и то же самое можно сделать для отрезков между сгенерированными точками.

Итак, начав с двух отрезков (квадратичная кривая), можно дважды применить линейную интерполяцию, чтобы найти точки кривой. С тремя начальными

отрезками (кубическая кривая) выбираются три точки, которые, в свою очередь, определяют два новых отрезка и т. д.; таким образом, линейная интерполяция применяется три раза.

Следуя этому определению, даже прямолинейный отрезок можно представить в виде кривой Безье. У него нет контрольных точек и всего один отрезок, поэтому линейная интерполяция применяется только один раз.

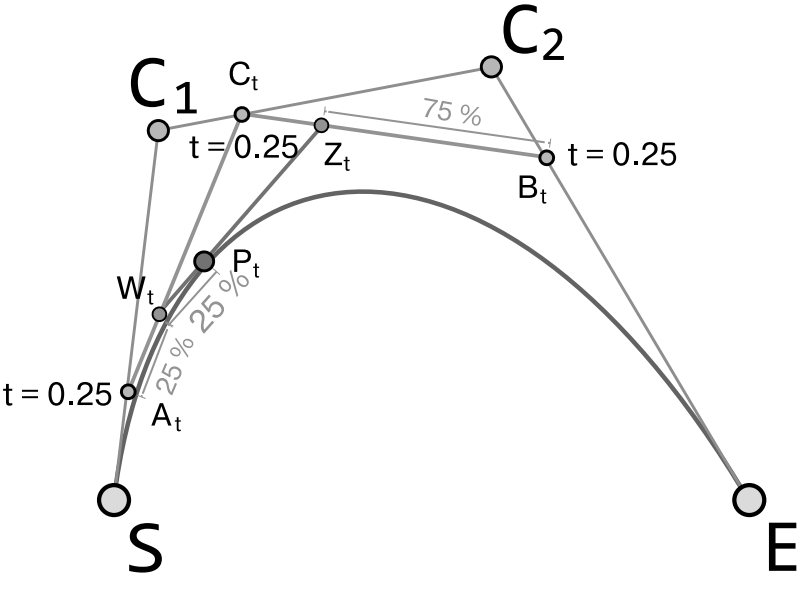


Рис. 15.22. Рисование кубической кривой Безье. Оцените повышенную сложность по сравнению с квадратичными кривыми

В этом пограничном случае легко увидеть, что общая точка P_t задается векторным уравнением:

$$P_L(t) = E \times t + S \times (1 - t), t \in [0, 1].$$

Я не буду выводить уравнение для квадратичной кривой, а просто покажу его:

$$P_Q(t) = E \times t^2 + C_1 \times 2 \times t \times (1 - t) + \mathrm{S} \times (1 - t)^2, t \in [0, 1].$$

В общем случае точки для кривой Безье с $n - 2$ контрольными точками можно выразить уравнением:

$$P(t) = \sum_{i=0}^{n} Q_i \binom{n}{i} t^i (1-t)^{n-1}, \; t \in [0, 1],$$

используя соглашение: $S = Q_0$, $E = Q_n$ и $C_i = Q_i \; \forall \; i = 1 \ldots n - 1$.

15.4.4. Пересечение квадратичных кривых Безье

В наших примерах мы будем использовать только подмножество кривых Безье — симметричные квадратичные кривые Безье, когда контрольная точка находится на равном расстоянии между двумя конечными точками. Благодаря этому можно упростить рассуждения о кривой и способ поиска пересечений.

На рис. 15.23 можно заметить несколько интересных фактов:

- кривая всегда является частью параболы;
- точку *C* можно сохранить, используя единственное действительное значение (расстояние от линии, проходящей через конечные точки);
- касательная к кривой, параллельная отрезку *SE*, проходит точно посередине между этим отрезком и точкой *C*.

Далее вы увидите, почему третий пункт особенно важен для нас.

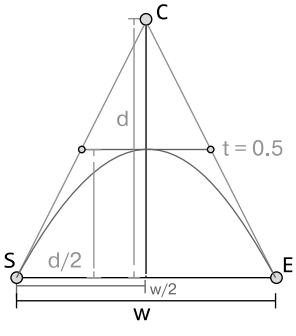


Рис. 15.23. Квадратичная кривая Безье, контрольная точка которой находится на одинаковом расстоянии от конечных точек. Касательная к кривой, параллельная отрезку SE, находится на расстоянии d/2 от этого отрезка, где d — расстояние точки C от отрезка SE

Для начала кратко перечислим методы, которые можно использовать для проверки пересечения кривых Безье.

- *Разбиение Безье (Bézier subdivision)* — это метод, основанный на *выпуклой оболочке*[1]. Выпуклая оболочка для кривой Безье с $n - 2$ контрольными точками представляет собой многоугольник с n сторонами, вершины которого являются конечными и контрольными точками. На рис. 15.24 показано, как вычислить пересечение двух кривых путем сравнения их выпуклых оболочек. Если они не перекрываются, кривые не пересекаются; иначе каждая кривая разбивается пополам, и две половины одной кривой проверяются на перекрытие с двумя половинами другой кривой. Описанный шаг можно повторять до тех пор, пока не обнаружится перекрытие или кривые не разобьются на настолько малые участки, что их можно будет аппроксимировать отрезками (в пределах некоторой допустимой погрешности). После этого для проверки пересечения отрезков можно использовать алгоритм из подраздела 15.4.1.

[1] Выпуклая оболочка фигуры — наименьшее выпуклое множество, содержащее ее.

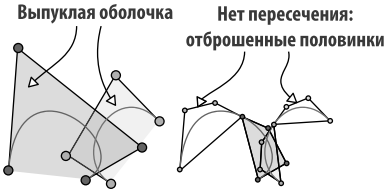


Рис. 15.24. Метод разбиения Безье (первая итерация). Показанные точки фактически являются контрольными точками кубических кривых. Очевидно, что этот метод можно применить и к квадратичным, и к квартичным, и к обобщенным кривым

- *Усечение Безье (Bézier clipping)* — для кривых Безье в целом это самый эффективный, но и самый сложный метод. Мы не будем вдаваться в его подробности.
- *Интервальное разбиение (interval subdivision)* — этот метод похож на разбиение Безье, но лучше адаптируется к нашим более строгим требованиям. Разница в том, что в этом случае сначала отыскиваются вертикальная и горизонтальная касательные к кривой. Разбивая кривую так, чтобы точки, в которых касательная параллельна одной из осей, находились на крайних точках каждого отрезка, мы гарантируем, что на каждом отрезке кривая монотонна (поскольку функция может менять свой тренд только в таких точках) и что для каждого значения на оси X есть только одна точка, принадлежащая кривой на каждом из этих участков. На рис. 15.25 показано, как работает метод интервального разбиения для обобщенной кривой Безье. Перечисленные свойства можно использовать для дальнейшего изучения кривой, разделив пополам каждый отрезок по оси X и вычислив точную точку на кривой. Эта точка будет одним из углов ограничивающего прямоугольника каждой секции, и можно будет сравнить ограничивающие прямоугольники двух кривых. Их стороны будут параллельны декартовым осям, поэтому останется лишь проверить несколько неравенств, чтобы узнать, пересекаются ли эти кривые.

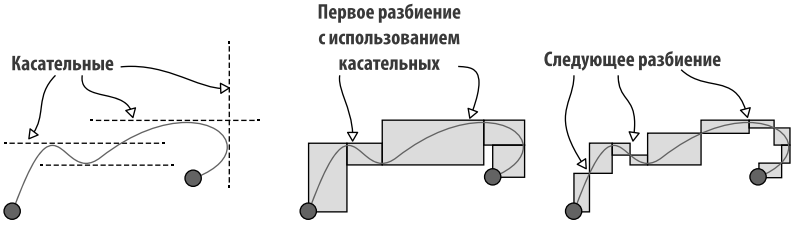


Рис. 15.25. Метод интервального разбиения применительно к обобщенной кривой Безье

Применение метода интервального разбиения к симметричным квадратичным кривым Безье еще проще, потому что эти кривые имеют не более двух точек, в которых их касательная параллельна одной из осей. Более того, для каждой из кривых легко вычислить начальный ограничивающий прямоугольник. Как показано на рис. 15.26, все они лежат внутри прямоугольника, ограниченного линией, проходящей через конечные точки, линией, параллельной бывшей касательной к вершине кривой (на расстоянии $d/2$ от отрезка SE, как упоминалось выше), и линиями, перпендикулярными к SE, проходящими через эти конечные точки.

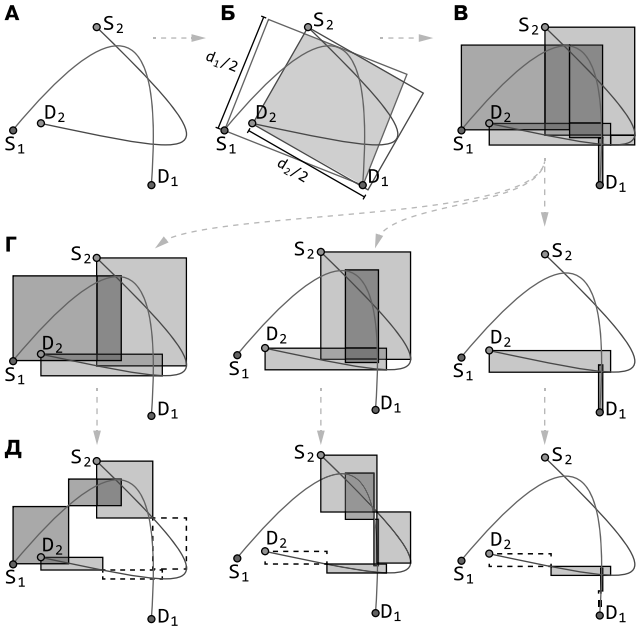


Рис. 15.26. Метод интервального разбиения, используемый для проверки пересечения двух симметричных квадратичных кривых Безье. *А*, *Б*. Вычисляются ограничивающие прямоугольники с использованием знаний о касательных к кривой в ее вершине. Здесь они пересекаются. *В*. Таким образом, нам нужно двигаться вперед, вычислить касательные к кривой, параллельные осям, и использовать их в качестве опорных точек, в которых можно разделить кривые. Во всех сегментах кривая монотонна, и ее можно интерполировать прямым отрезком. Каждая пара сегментов двух кривых будет пересекаться не более чем в одной точке. *Г*. Для всех сегментов кривой C_1 по отдельности вычисляются пересечения с использованием только перекрывающихся участков кривой C_2 (затем результаты суммируются для каждого из трех случаев). *Д*. Любой отрезок можно еще раз разделить пополам. Только некоторые подразделы все еще будут перекрываться. Шаги *Г–Д* можно повторять, пока подразделы не станут достаточно малыми, чтобы ошибка аппроксимации кривой прямым отрезком оказалась в пределах заданного порога

Таким образом, неофициально сформулированная последовательность шагов, выполняемых алгоритмом проверки пересечения двух симметричных квадратичных кривых Безье, может быть такой, как в списке ниже.

1. Вычисляем ограничивающие прямоугольники двух кривых, как показано на рис. 15.26, *А*.
2. Если ограничивающие прямоугольники не пересекаются, возвращаем 0; если пересекаются, то продолжаем дальше.
3. Вычисляем вертикальные и горизонтальные касательные к каждой кривой и используем их для разделения кривой на две или три части (в зависимости от того, имеют ли они обе касательные или только одну).

4. Рекурсивно:
   1) для каждого сегмента кривой C_1 проверяем, какие сегменты кривой C_2 он перекрывает;
   2) если никакие два сегмента не перекрываются, возвращаем 0;
   3) в противном случае возвращаем сумму[1] пересечений перекрывающихся пар:
      а) для каждой пары перекрывающихся секций;
      б) разделяем каждый из перекрывающихся сегментов пополам;
      в) если сегменты достаточно малы, чтобы их можно было аппроксимировать прямыми отрезками, вычисляем пересечение отрезков;
      г) в противном случае рекурсивно проверяем четыре раздела, полученные в результате разбиения.

Если сравнить этот алгоритм с алгоритмом определения пересечения отрезков в подразделе 15.4.1, то можно обнаружить основное отличие, которое бросается в глаза: рассмотренный алгоритм является рекурсивным (или итеративным), тогда как для определения пересечения отрезков всегда выполняется постоянное количество операций.

Это означает, что, если использовать кривые вместо прямых отрезков, вычисление пересечения ребер на каждой итерации алгоритма оптимизации будет обходиться намного дороже.

Более того, если в случае с квадратичными кривыми у нас уже есть четыре возможных пересечения для каждой пары ребер, то в случае с кубическими кривыми ситуация еще хуже: здесь больше параметров для оптимизации и каждое изменение может иметь большее значение.

Вот почему мы фокусируемся на прямолинейных ребрах. Прежде чем принимать решение об использовании кривых, нужно еще и еще раз перепроверить требования и оценить преимущества и затраты.

15.4.5. Пересечения «вершина — вершина» и «ребро — вершина»

До сих пор мы откладывали обсуждение темы, каким образом проверить представление, чтобы убедиться, что никакая пара вершин не находится в одной и той же позиции и что ни одно ребро не пересекает вершину.

Уже было показано, что нарисовать ребра можно множеством способов, но есть также множество способов нарисовать вершины: в действительности они могут изображаться точками, кругами, квадратами, восьмиугольниками и вообще любыми

[1] Напомню, что если два отрезка могут пересекаться только в одной точке, то две параболы могут иметь до четырех пересечений. Разбив обе кривые по точкам касания, приходим к тому, что каждый сегмент может пересечь другой сегмент только один раз.

многоугольниками. В зависимости от выбора вариантов изображения понадобятся разные алгоритмы.

Здесь будем считать, что вершины изображаются кругами (вершины-точки — крайний случай круга с радиусом, близким к нулю), а ребра — прямолинейными отрезками. Это даст нам инструменты для обработки простейших и основу для создания более сложных решений, если понадобится.

В частности, если вы решите изображать вершины квадратами или правильными многоугольниками, то всегда сможете взять за основу *описанную окружность*[1] или *минимальную ограничивающую окружность*[2] многоугольника и изменить ограничения с учетом этих окружностей вместо фактической формы вершин.

Использование кругов существенно упрощает нам жизнь, позволяя просто проверить расстояние между двумя вершинами (между их центрами) или между вершиной и ребром.

Алгоритм проверки пересечения двух вершин тривиально прост: нужно лишь проверить расстояние между их центрами, которое должно быть больше суммы их радиусов, как показано на рис. 15.27.

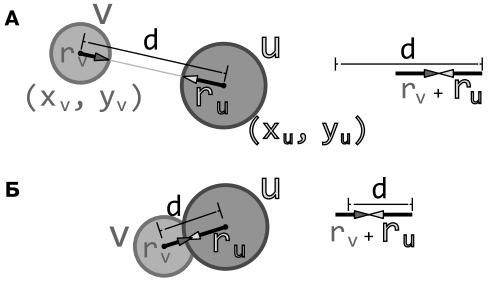


Рис. 15.27. Пересечение двух вершин. Чтобы две окружности перекрывались, расстояние d между их центрами должно быть меньше суммы радиусов окружностей. Если $d > r_v + r_u$, как в случае А, то здесь все ясно; когда $d \le r_v + r_u$, как в случае Б, то два круга перекрываются

Чтобы убедиться, что ребро не пересекает вершину (которая не является конечной точкой ребра), нужно проверить расстояние между ребром и вершиной, оно должно быть больше радиуса вершины.

В свою очередь, когда ребра рисуются прямолинейными отрезками, для этого необходимо проверить три расстояния. Если расстояния между конечными точками отрезка (*S*, *E* на рис. 15.28) и центром вершины (назовем ее *C*) меньше радиуса вершины, то, безусловно, пересечение имеет место. Но даже если эти расстояния больше, ребро может пересекать вершину своей серединой, поэтому нужно проверить расстояние между прямой, проходящей через отрезок, и центром вершины, и убедиться, что точка пересечения прямой с перпендикуляром, опущенным на эту прямую из вершины, попадает в границы отрезка.

---

[1] Описанная окружность — это окружность, проходящая через все вершины многоугольника.

[2] Минимальная ограничивающая окружность — это наименьшая окружность, охватывающая все точки множества.

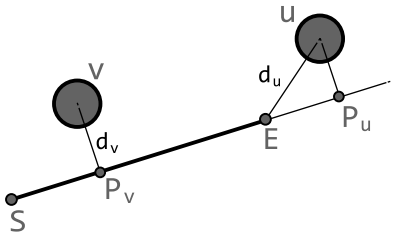


Рис. 15.28. Пересечение вершины с ребром. При оценке расстояния между отрезком и вершиной возможны два случая: один (*v* на рисунке), когда нужно вычислить расстояние от прямой до точки, и другой, когда вычисляются расстояния до концов отрезка

Чтобы найти расстояние от точки до отрезка, можно использовать следующую формулу (выводом которой здесь заниматься не будем):

$$\mathrm{dist}\left(\overline{SE}, C\right) = \frac{\left|\left(E_y - S_y\right)C_x - \left(E_x - S_x\right)C_y + E_x S_y - E_y S_x\right|}{\sqrt{\left(E_x - S_x\right)^2 + \left(E_y - S_y\right)^2}}.$$

И наконец, чтобы проверить, попадает ли точка пересечения прямой, проходящей через отрезок, и перпендикуляром, опущенным на прямую из вершины (минимальное расстояние от вершины до прямой), внутрь отрезка, можно использовать тот же алгоритм, что был разработан для проверки пересечения отрезков с использованием точек S и E с одной стороны и v и P_v (или u и P_u) с другой стороны. Нужно просто убедиться, что коэффициент мультипликатора находится между 0 и 1.

РЕЗЮМЕ

- Графы — это абстрактные алгебраические структуры; для их визуализации можно создать графическое представление в геометрическом пространстве.
- Представление графа — это отображение вершин графа в точки евклидова пространства, а также ребер в (жордановы) кривые в том же пространстве.
- Если представление отображает граф на плоскость и ни одно из ребер не пересекает другое ребро или вершину (кроме его концов), то такое представление называется планарным.
- Все графы можно представить в трехмерном евклидовом пространстве без пересечений ребер, но не все графы являются планарными.
- Если ограничить представление ребер прямолинейными отрезками, а не обобщенными жордановыми кривыми, то для некоторых графов будет невозможно создать прямолинейное изображение с минимально возможным числом пересечений.
- Определение пересечений ребер — непростая задача. В прямолинейных представлениях она менее затратна, потому что проверка пересечения двух прямолинейных отрезков требует постоянного количества операций с векторами. При переходе к кривым необходимо использовать рекурсивные алгоритмы (или их итерационные аналоги).

16 Градиентный спуск: оптимизация задач (не только) на графах

В этой главе

- ✓ Рандомизированная эвристика поиска минимального числа пересечений.
- ✓ Введение в функции стоимости, показывающие работу эвристики.
- ✓ Градиентный спуск и реализация универсальной версии.
- ✓ Сильные и слабые стороны градиентного спуска.
- ✓ Применение градиентного спуска к задаче представления графа.

Если я упомяну прием, называемый *градиентным спуском*, не покажется ли вам это название знакомым? Возможно, вы вовсе не слышали об этом приеме или слышали название, но не помните, как он работает. И это нормально. Однако если я спрошу вас о машинном обучении, задачах классификации или нейронных сетях, то, скорее всего, вы точно поймете, о чем я говорю; бьюсь об заклад, эти термины вызвали у вас гораздо больший интерес.

Итак, градиентный спуск (или вариация на тему) — это метод оптимизации, который используется многими алгоритмами машинного обучения. Но знаете ли вы, что задолго до того, как стать основой обучения с учителем, указанный метод был разработан для решения задач оптимизации, подобных тем, с которыми мы с вами сталкивались, когда обсуждали графы?

Если вы не в курсе или просто хотите углубиться в эту тему и узнать больше о том, как работает градиентный спуск, то материал этой главы станет для вас идеальным источником знаний.

В главе 15 было показано, что для планарных графов существуют эффективные алгоритмы поиска планарного представления за линейное время, но не все графы планарны и не всегда возможно нарисовать ребра графа на плоскости без пересечений.

Что еще хуже, каждый непланарный граф имеет минимальное количество пересечений при начертании на плоскости, но нет (и не может быть[1]) никакого эффективного алгоритма, способного найти это число (или представление с как можно меньшим числом пересечений) для любого обобщенного графа.

Задача поиска минимального пересечения ребер является NP-трудной, и в настоящее время проверка числа пересечений невозможна для многих категорий непланарных графов более чем с ~20 вершинами.

Похоже, мы мало что можем сделать... или все-таки можем?

В этой главе вы познакомитесь с эвристикой для получения представлений графа с минимальным количеством пересечений, приближающимся к его числу пересечений (ЧП). Обратимся к рис. 16.1. Цель подобных эвристик при применении, например, к графу K_4 состоит в том, чтобы найти такое представление, как *справа* или эквивалентное представление без пересечения ребер. Начнем с простых алгоритмов перебора, а затем усовершенствуем их, чтобы получить лучшие результаты (меньше пересечений)[2] за меньшее время, и поговорим об оптимизации.

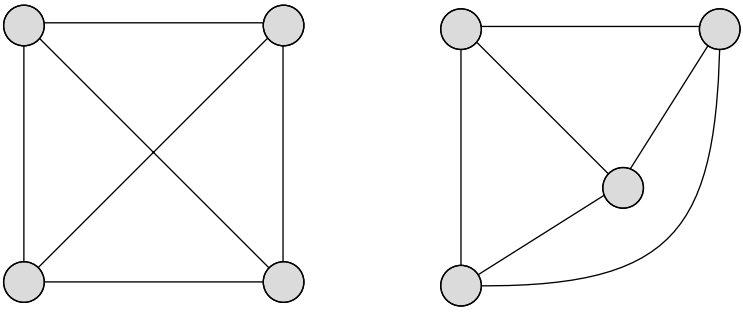


Рис. 16.1. Два представления полного графа K_4. Представление слева неоптимальное, потому что имеет пару пересекающихся ребер

В следующих разделах будет определена категория задач оптимизации, играющих важную роль в нашей повседневной жизни: сетевая маршрутизация, графики поставок, создание печатных плат и проектирование компонент. Все эти задачи можно

---

1 Если только кто-нибудь не докажет, что P = NP, а это пока не доказано, но считается маловероятным.

2 Этот процесс уточнения продолжится в следующих нескольких главах, где вы познакомитесь с новыми категориями алгоритмов оптимизации.

выразить в виде графов, функций стоимости и алгоритмов оптимизации, целью которых является поиск конфигурации с минимальной стоимостью. Нам предстоит обсудить три разных метода оптимизации для поиска приближенных решений: *случайную выборку*, *восхождение на холм* и *градиентный спуск*.

16.1. ЭВРИСТИКА ДЛЯ ОПРЕДЕЛЕНИЯ ЧИСЛА ПЕРЕСЕЧЕНИЙ

Во введении к главе прозвучал вопрос: можно ли что-то предпринять, чтобы выбраться из описанного затруднительного положения в анализе числа пересечений. Фактически, насколько нам известно, мы мало что можем сделать, по крайней мере если не ослабить требования к детерминизму алгоритма, гарантирующему правильный ответ, и не довольствоваться представлением графа с минимальным числом пересечений (или минимальным числом прямолинейных[1] пересечений — ЧПП).

Обычная стратегия решения *NP-трудных* задач — использование эвристик. Эти алгоритмы, часто работающие недетерминированно, способны дать субоптимальный ответ за разумное время.

ПРИМЕЧАНИЕ

Мы уже встречали рандомизированные алгоритмы в этой книге, например в главах 3 и 4, но если вам нужно освежить память, не постесняйтесь заглянуть в приложение E, где дается краткий обзор рандомизированных алгоритмов.

16.1.1. Мне послышалось, вы только что сказали «эвристики»?

Я понимаю, что понятие «эвристика» может сбивать с толку: как можно удовольствоваться алгоритмом, который не возвращает правильный ответ? Иногда нельзя. Есть задачи, для которых нам совершенно необходимо получить правильный ответ, даже если придется ждать дольше. Например, управляя атомной электростанцией или разрабатывая новое лекарство, вы не захотите согласиться с неоптимальным решением, не испробовав все возможные (и разумные) конфигурации.

Но иногда получить правильный ответ просто невозможно. Экспоненциальный алгоритм становится неприменимым, когда объем входных данных превышает несколько десятков элементов, а при еще большем объеме данных мы можем просто не дожить до вывода программой результатов.

---

[1] Как было показано в главе 15, если изображать ребра только прямолинейными отрезками, то для некоторых графов невозможно получить представление с минимальным количеством пересечений. Вот почему было введено понятие числа прямолинейных пересечений (ЧПП): минимального количества пересечений ребер на всех возможных изображениях графа с прямолинейными ребрами.

РАЗВЕ НЕЛЬЗЯ СОЗДАТЬ БОЛЕЕ БЫСТРЫЕ КОМПЬЮТЕРЫ?

Даже суперкомпьютеру, выполняющему ~10^{16} операций в секунду для вычислений по экспоненциальному алгоритму, выполняющему $O(2^n)$ шагов, при объеме входных данных, равном 100 элементам, потребуется $O(10^{14})$ с, что равно плюс-минус 3 миллионам лет!

И если бы мы с вами могли сделать наши компьютеры в миллион раз быстрее (что, даже при условии соблюдении закона Мура, займет 30 лет), то лишь немного увеличили бы собственную способность решать подобные задачи: для вычислений при объеме входных данных, равном 100 элементам, потребовалось бы три года, но уже три тысячи лет — при объеме 110 и снова 3 миллиона лет — при объеме 120 входных элементов!

Время решения более простых задач, даже если оно не превысит продолжительности истории человечества, все равно может оказаться слишком долгим. Подумайте о быстро меняющихся событиях или о прогнозах — мало толку от прогноза погоды на завтра, если он будет получен лишь три дня спустя!

Итак, во всех этих случаях можно было бы согласиться на неоптимальный ответ при условии, что он будет получен в разумные сроки. Но и это еще не все.

Еще одно важное соображение в пользу эвристик: несмотря на то что они не гарантируют оптимального решения для всех входных данных, некоторые эвристики способны вернуть оптимальный результат за разумное время для подмножества всего пространства задачи, а иногда достаточно хорошо справляются и с реальными данными.

Как такое возможно? Это связано с тем, что теория NP-трудности касается *производительности в худшем случае*. Но для некоторых задач существует очень малое количество пограничных случаев, которые трудно решить. Для практических приложений нас часто интересует *трудность в среднем случае*, то есть насколько сложно в среднем решить задачу на экземплярах, наблюдаемых в реальных сценариях.

Существует множество эвристик, разработанных для алгоритмов обработки графов, например для решения задачи о коммивояжере[1] или поиска клики в графе[2]. Конечно, не все эвристики одинаковы. Обычно они представляют собой компромисс между производительностью и точностью, и для некоторых из этих алгоритмов можно

[1] Имея список городов и расстояния между каждой парой городов, нужно найти кратчайший маршрут, пролегающий через все города, пересекающий каждый город ровно один раз и возвращающийся в исходный город. Разумеется, это NP-полная задача.

[2] Поиск наибольшего подмножества вершин графа, каждая вершина которого смежна со всеми другими вершинами в том же подмножестве. Другими словами, поиск наибольшего полного подграфа в данном графе.

установить пределы их точности, например доказать, что возвращаемое ими решение будет не хуже оптимального в определенных пределах.

Итак, какую же эвристику можно использовать для поиска хорошего (или по крайней мере приличного) представления графа?

Теперь я хочу, чтобы вы вспомнили, о чем говорилось несколькими строками выше: для решения каждой задачи может иметься множество разных приближенных алгоритмов, но не все они одинаково хороши.

Начнем с простого, с простой эвристики, далекой от идеала. Она обрисует нам цель и позволит понять сценарий решения задачи, а затем мы сможем (и будем) постепенно улучшать ее, чтобы увеличить производительность.

Прежде чем приступить к первой попытке, я предлагаю вам приостановиться, поразмышлять над задачей и попытаться выработать свою идею. Кто знает, может быть, вы сделаете прорыв и придумаете новое решение! Когда будете готовы, продолжим.

Будем повторно применять алгоритм, который видели в предыдущих главах, в частности вспомогательный метод, который использовался при кластеризации.

Помните метод k-средних?[1] Перед нами стояла следующая задача, связанная с инициализацией центроидов: придумать начальный выбор для k точек в n-мерном пространстве. Она выглядела проще, чем оказалось на самом деле, верно? Тогда вы увидели, как важно выбрать хорошие начальные точки, чтобы получить быструю сходимость и хороший результат.

Загляните в раздел 12.2 и, в частности, рассмотрите листинги 12.1 и 12.2, где реализована случайная инициализация точек для центроидов.

Разбираясь с представлениями графа, имеем аналогичную задачу. Нужно выбрать определенное количество двумерных точек, и то, как будет сделан этот выбор, напрямую определит результат (на этот раз после выбора не будет запускаться алгоритм оптимизации).

Одно важное отличие состоит в том, что в методе k-средних у нас имелось базовое распределение и выбор точек был сложнее и требовал особой осторожности: его надо было провести так, чтобы точки были равномерно распределены относительно этого распределения.

Для представления графа можно рисовать точки в конечной области плоскости, обычно в пределах прямоугольника[2], и эти точки будут определять количество пересечений ребер.

---

[1] Загляните в главы 12 и 13.

[2] Полезно иметь некоторую гибкость в отношении области, в которой изображаются точки, потому что это позволяет получить лучшие результаты в зависимости от размера графа и избежать слишком плотных и слишком разреженных представлений.

Теперь определим задачу более формально.

Дан простой граф $G = (V, E)$ с $n = |V|$ вершинами и $m = |E|$ ребрами. Нужно нарисовать n случайных точек в границах конечного прямоугольника, углы[1] которого имеют координаты $(0, 0)$ и (W, H), так что:

- для каждой точки (x, y) выполняется условие $0 \le x < W, 0 \le y < H$;
- ребра изображаются прямолинейными отрезками;
- вершины изображаются в виде точек (или окружностей) с центрами в этих n точках;
- никакие две вершины не могут иметь одинаковые координаты, поэтому для любых двух точек (x_1, y_1) и (x_2, y_2) выполняется условие $x_1 \ne x_2$ или $y_1 \ne y_2$;
- ни одна вершина не может лежать на ребре;
- предполагается, что в G нет петель[2], иначе можно было бы бесконечно долго рисовать петли, не пересекающиеся ни с какими другими ребрами.

На рис. 16.2 показано несколько примеров допустимого и недопустимого выбора центров вершин с учетом наложенных ограничений; в частности, после выбора точек нужно будет проверить условия в пунктах 4 и 5, но обсуждение этих проверок пока отложим. На данный момент просто предположим, что у нас есть вспомогательные методы, выполняющие такие проверки, и если обнаружится, что какое-то из этих ограничений нарушено, можно будет применить одну из двух стратегий, чтобы исправить ситуацию:

- исправить положение вершин, например немного переместив в случайном направлении точки, которые совпадают или попадают на ребро;
- отменить решение, нарушающее ограничение, и начать заново.

Обратите внимание, что если изображать вершины кружками (как на рис. 16.2), а не просто точками, то придется усилить требование к вершинам и добавить проверку непересечения двух окружностей.

Вы видели при обсуждении метода k-средних, что в случае, когда используются рандомизированные методы, в этом моменте иногда нам может повезти, но чаще случается обратное. На рис. 16.3 показано особенно неудачное распределение для графа K_6. Из материалов главы 15 становится понятно, что этот полный граф можно нарисовать на плоскости с тремя пересечениями между ребрами.

Для увеличения шансов на успех в методе k-средних использовался прием повторного запуска: выполнялись несколько прогонов алгоритма (каждый раз со случайной инициализацией), а затем сохранялось лучшее решение из найденных.

---

[1] Верхний левый и нижний правый, в соответствии с тем, как индексируются строки и столбцы экрана в большинстве языков программирования.

[2] Как отмечалось в главе 14, петля — это ребро, которое начинается и заканчивается в одной и той же вершине.

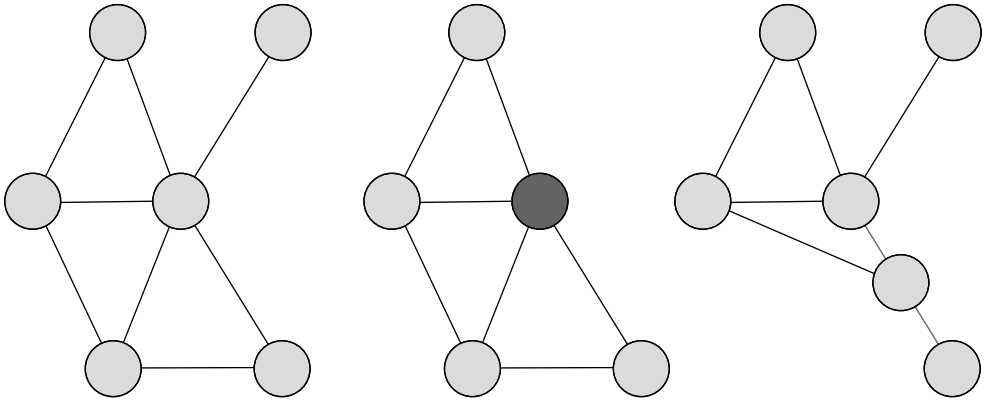


Рис. 16.2. Три представления одного и того же графа. Слева правильное представление. Представление в середине недопустимо, потому что две вершины оказались в одной точке, а представление справа имеет вершину, лежащую на ребре (которая не является конечной точкой этого ребра)

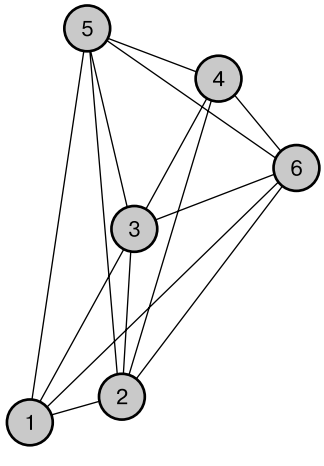


Рис. 16.3. Случайное представление для полного графа K_6. Обратите внимание, насколько неудачным оказался выбор позиций

Эта стратегия выглядит интересной и для нашей задачи. Сформулируем последовательность шагов ее реализации.

1. Случайным образом сгенерировать координаты вершин.
2. Убедиться, что они соответствуют ограничениям для ребер и вершин (и отказаться от текущей попытки, если это не так).
3. Подсчитать количество пересечений ребер.
4. Если это лучший результат на данный момент, то сохранить его; иначе — отбросить.
5. Повторить, начиная с п. 1.

Блок-схема приведенного алгоритма показана на рис. 16.4, реализация — в листинге 16.1. Эта эвристика называется *случайной выборкой*.

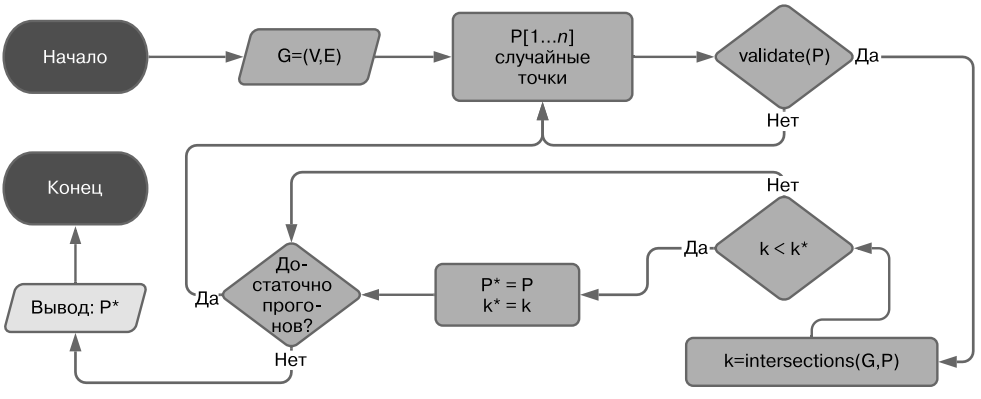


Рис. 16.4. Блок-схема алгоритма, генерирующего случайные представления и выбирающего представление с наименьшим количеством пересечений ребер

Листинг 16.1. Случайная выборка для поиска представления с наименьшим числом пересечений

Инициализируем лучший результат — количество пересечений — максимально возможным значением

Метод randomEmbedding принимает граф, ограничивающий прямоугольник для представления (ширину и высоту) и количество прогонов алгоритма, и возвращает представление в виде массива двумерных точек, соответствующих вершинам

Инициализируем массив точек. Он будет содержать по одной точке на вершину в том же порядке, в каком они хранятся в графе. Эту роль также может играть словарь, отображающий вершины в точки

Для каждой вершины в графе...

Сгенерируются двумерные координаты точки внутри прямоугольной области размером W × H. Метод random может возвращать число с плавающей точкой или просто целое число; это зависит от конкретной реализации и требований

```
function randomEmbedding(graph, W, H, runs)
  kBest ← inf
  for i in {1..runs} do
    embedding ← []
    for j in {0..|graph.vertices|-1} do
      x ← random(0, W)
      y ← random(0, H)
      embedding[j] ← (x,y)
    if not validateEmbedding(embedding, graph) then
      i--
    else
      k ← edgeIntersections(graph, embedding)
      if k < kBest then
        kBest ← k
        embeddingBest ← embedding
  return embeddingBest
```

Выполняем алгоритм runs раз

Добавляем новую точку в массив embedding

Если полученные к данному моменту точки не удовлетворяют требованиям и ограничениям, то попытка аннулируется (i уменьшается, соответственно, эта итерация не будет учтена как полноценная попытка)

Вычисляем количество пересечений ребер: метод является обобщенным, поэтому он может поддерживать как прямолинейные, так и криволинейные ребра, хотя здесь предполагается именно первое

Возвращаем найденное представление с наименьшим числом пересечений ребер

Если текущий результат лучше предыдущего, то сохраняем его

Имейте в виду, что мы не выдвигаем никаких предположений о связности графа, но разбивка его на связанные компоненты перед применением эвристики может улучшить конечный результат.

В листинге 16.1 абстрагированы два важных вспомогательных метода: `validateEmbedding` и `edgeIntersections`. Оба могут быть реализованы отдельно и адаптированы к фактическому контексту. Однако для обоих можно дать общие определения, в свою очередь абстрагирующие более конкретные реализации: эти определения показаны в листингах 16.2 и 16.3.

Листинг 16.2. Метод validateEmbedding

Проверяем пересечение двух вершин. Сама проверка зависит от контекста; это может быть простая проверка несовпадения двух точек или проверка непересечения кругов, используемых для изображения вершин. Если проверка не пройдена, представление должно быть отклонено

Метод validateEmbedding принимает граф и проверяемое представление и возвращает true, если представление прошло проверку, и false в противном случае

```
function validateEmbedding(graph, embedding)
  n ← |graph.vertices|
  for i in {0..n-2} do                          ← Цикл по всем парам вершин
    for j in {i+1..n-1} do
      if vertexIntersectsVertex(embedding[i], embedding[j]) then
        return false
  for edge in graph.edges do                    ← Цикл по всем ребрам графа
    for vertex in graph.vertices do
      if vertex <> edge.source and vertex <> edge.destination and
          edgeIntersectsVertex(
            embedding[indexOf(edge.source)],
            embedding[indexOf(edge.destination)],
            embedding[indexOf(vertex)]) then
        return false
  return true
```

Для каждого ребра выполняем цикл по всем вершинам, чтобы убедиться, что ни одна вершина не накладывается на ребро

Если вершина vertex не является одной из конечных точек ребра, то нужно убедиться, что ребро не будет изображено над вершиной (иначе оно может выглядеть как два ребра, примыкающие к этой вершине). В случае пересечения ребром вершины следует отклонить рассматриваемое представление

Реализация вспомогательных методов, использованных в листингах 16.2 и 16.3, обсуждалась в предыдущей главе, в разделе 15.4.

Обратите внимание, что вспомогательные методы, проверяющие пересечение двух вершин или вершины и ребра, зависят от контекста и от того, как изображаются вершины. Если они изображаются кругами, то нужно убедиться, что круги никаких двух вершин не пересекаются и никакие ребра не пересекают круги вершин (иначе такое ребро может выглядеть как два ребра, примыкающие к пересекаемой им вершине).

Листинг 16.3. Метод edgeIntersections

```
Метод edgeIntersections принимает граф и проверяемое
представление и возвращает количество пересечений между
ребрами графа для текущего представления

                                                    Некоторая
                                                    инициализация
function edgeIntersections(graph, embedding)
  m ← |graph.edges|
  k ← 0
  for i in {0..m-2} do        ← Цикл по всем парам ребер
    edge1 ← graph.edges[i]
    for j in {i+1..m-1} do
      edge2 ← graph.edges[j]
      if edgeIntersection(edge1, edge2) then
        k ← k + 1
  return k
```

Если пара ребер пересекается, то увеличиваем счетчик пересечений на 1

Как и в предыдущем методе, мы абстрагировались от фактического алгоритма проверки ребер. Благодаря этому, в зависимости от контекста, приведенный метод можно использовать и с прямолинейными ребрами, и с ребрами, изображаемыми с помощью кривых Безье, и т. д.

16.1.2. Расширение до поддержки криволинейных ребер

В предыдущем разделе было добавлено ограничение, требующее изображения ребер в виде прямолинейных отрезков. Следовательно, представления были оптимизированы с учетом этого аспекта, причем общее количество пересечений не могло быть меньше числа прямолинейных пересечений графа. Как было показано в главе 15, существует множество графов, для которых число прямолинейных пересечений больше числа пересечений (которое, в свою очередь, является абсолютным минимумом числа пересечений в любом планарном представлении графа).

Требование прямолинейности ребер нигде не выражено в коде, который остается максимально абстрактным. Однако, если понадобится добавить поддержку криволинейных ребер, нужно внести по крайней мере одно изменение в листинг 16.1, решив, как моделировать каждое ребро. Это можно сделать несколькими способами, от самого простого — случайного выбора некоторых параметров, определяющих форму каждого ребра, до более сложных, применяющих некоторые оптимизации к этим параметрам для минимизации пересечений.

Но сначала нужно решить, о каких параметрах говорим. Для простоты ограничимся кривыми Безье. Квадратичные кривые Безье можно описать тремя параметрами (двумя конечными и одной контрольной точками), тогда как кубические версии, которые (как показано на рис. 16.5) более гибкие, требуют двух контрольных точек, а всего — четырех двумерных точек (определяемых восемью скалярными параметрами).

Детали этих представлений обсуждались в подразделе 15.4.3. Выбрав подкатегорию кривых (квадратичные, симметричные кривые Безье, как показано на рис. 16.6), уже можно описать, как расширить алгоритм в листинге 16.1 для поддержки дополнительных параметров.

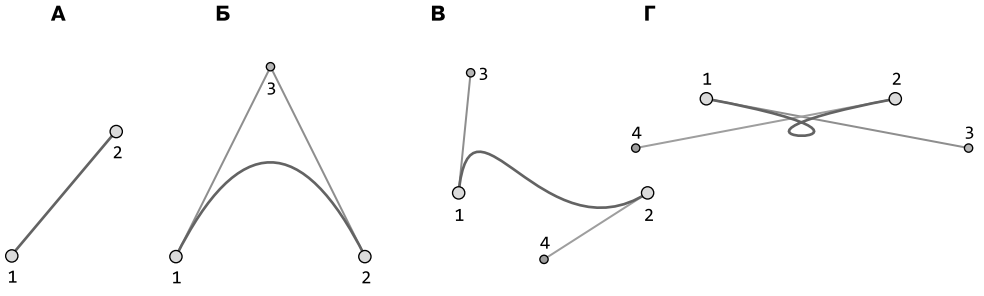


Рис. 16.5. Примеры кривых Безье. Сегмент *А* можно считать крайним случаем с 0 контрольных точек, тогда как квадратичные кривые *Б* имеют одну контрольную точку, а кубические (*В*, *Г*), наиболее гибкий вариант, — две контрольных точки (плюс, без сомнения, две конечные точки)

В частности, чтобы остаться верными выбору полностью рандомизированного алгоритма, можно просто случайным образом выбрать контрольную точку (точки) для каждого ребра; однако такая большая свобода выбора может породить ребра странной формы и тем самым замедлить сходимость алгоритма к хорошему решению.

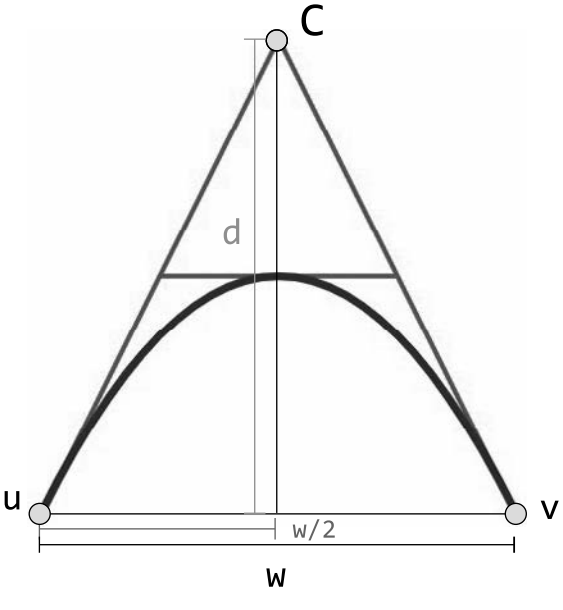


Рис. 16.6. Квадратичная кривая Безье, используемая для представления ребра между вершинами u и v. Точка C является контрольной точкой кривой. Для этого примера ограничимся подмножеством всех возможных квадратичных кривых Безье между двумя точками и, в частности, только кривыми, контрольная точка которых лежит на прямой, перпендикулярной отрезку uv и проходящей через середину отрезка (другими словами, на линии, образованной точками плоскости, находящимися на одинаковом расстоянии от u и v)

Лучшей альтернативой могло бы быть еще большее ограничение возможных значений параметров кривых. Например, можно ограничиться квадратичными кривыми, контрольные точки которых находятся на одинаковом расстоянии от обоих концов ребер. Этот выбор, представленный на рис. 16.6, позволяет сбалансировать гибкость и сложность, и потребуется добавить только одно действительное число в модель каждого ребра — расстояние контрольной точки от отрезка, соединяющего конечные точки (d на рис. 16.6). Желательно (но не обязательно) также ограничить возможные значения d, чтобы они не превышали величины w — расстояния между конечными точками. Отрицательные значения d вызовут изменение выпуклости ребра (на рис. 16.6 при отрицательном значении d кривая пройдет ниже отрезка uv, соединяющего вершины).

16.2. КАК РАБОТАЕТ ОПТИМИЗАЦИЯ

Итак, случайный выбор работает. Если попробовать версию, реализованную в библиотеке JsGraphs[1], то можно заметить, что число пересечений (в среднем), возвращаемое рандомизированным алгоритмом с повторными попытками, лучше числа, возвращаемого однопроходной рандомизированной версией. Причина интуитивно понятна: если подбросить монету 100 раз, то решка выпадет хотя бы один раз с большей вероятностью, чем если подбросить ту же монету только один раз.

Чтобы воочию увидеть, как работает этот подход, можно использовать визуализацию происходящей оптимизации. Но это непросто, потому что оптимизации подвергается функция с $2n$ параметрами, а граф имеет n вершин. Для каждой вершины можно изменить ее координаты x и y.

Мы хорошо научились визуализировать функции одного параметра на двумерном графике: ось X обычно отражает значения параметра, а ось Y — значения функции. Функции с двумя параметрами тоже можно достаточно осмысленно визуализировать. Но, пока AR/VR (дополненная/виртуальная реальность) не войдет плотно в нашу жизнь, нам придется довольствоваться двумерной проекцией трехмерного сюжета, что неидеально, но все же осуществимо (как показано на рис. 16.8).

Задача усложняется, когда имеется три параметра. Одно из возможных решений — введение «времени» в качестве четвертого измерения, благодаря чему график можно изобразить в виде трехмерной волны, изменяющейся в зависимости от третьего параметра.

Однако и это решение трудно осмыслить, к тому же оно еще и не решает проблему, когда есть четыре или большее число параметров. Часто в таких случаях блокируются $k - 2$ параметров из k, чтобы можно было посмотреть, как функция ведет себя при изменении двух переменных.

Мы сделаем что-то подобное для нашей задачи создания представлений графа, чтобы посмотреть, как она работает, и лучше понять проблему. Но о какой же функции идет речь в данном случае?

---

[1] http://mng.bz/w9na.

16.2.1. Функции стоимости

Объектом оптимизации является *функция стоимости*: она хорошо выражает идею о том, что решение (каждое решение, которое мы попробуем) имеет стоимость, которую нужно минимизировать.

Существует обширная категория задач, называемых *задачами оптимизации*, для которых поиск решения эквивалентен исследованию пространства задачи и в конечном счете выбору решения с наименьшими затратами. Для таких задач обычно существует множество возможных определений функций стоимости. Некоторые лучше других (мы увидим почему), и важно тратить время на их анализ и делать выбор с умом, потому что его результат может сильно повлиять на скорость поиска оптимального решения, а иногда даже на саму возможность найти решение. В целом, однако, доказано, что большинство задач оптимизации являются NP-трудными, независимо от конкретного выбора функции стоимости.

Функция стоимости, которая фигурирует в нашей задаче поиска представления графа, —количество пересечений ребер в представлении. Сразу оговорюсь, что подобный выбор проблематичен для некоторых алгоритмов оптимизации, это будет проиллюстрировано позже.

Итак, на рис. 16.7 изображен полный граф K_4, представление которого имеет восемь степеней свободы: давайте заблокируем семь из них и позволим меняться только горизонтальному положению вершины v_4.

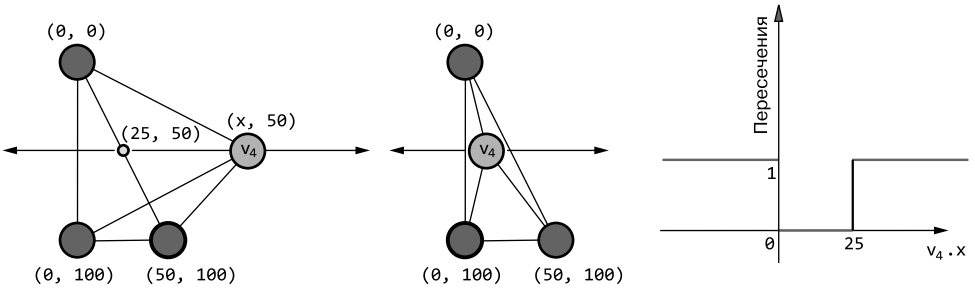


Рис. 16.7. Функция стоимости, отражающая «количество пересечений» в конкретном представлении K_4. Положение трех из четырех вершин зафиксировано, а для последней вершины, v_4 на диаграмме, можно менять только ее горизонтальное положение. При этих ограничениях стоимость как функция позиции x вершины v_4 является прерывистой функцией, равной 0, когда $0 < x < 25$, и равной 1, когда $x < 0$ или $x > 25$. Обратите внимание, что при $x == 0$ и $x == 25$ наблюдаются две точки разрыва и, в частности, предполагается, что стоимость в этих точках бесконечна, потому что вершина точно лежит на одном из ребер между другими вершинами

График функции стоимости имеет *ступенчатую форму*, ее значение резко меняется с 1 на 0, когда вершина входит внутрь подграфа, образованного тремя другими вершинами (треугольник, отмеченный безымянными вершинами на рис. 16.7).

Если бы мы позволили вершине v_4 перемещаться и по вертикали, и по горизонтали, то получили бы трехмерную диаграмму поверхности. Это показано на рис. 16.8 на примере полного графа K_5, точнее, на примере одного из возможных представлений этого графа.

Обратите внимание, что здесь имеются *поверхности* разрыва, где функция стоимости меняет свое значение: так происходит, когда |x| == |y| (тогда v лежит на линии, проходящей через диагональ квадрата) и когда x или y равны 0 или 100 (тогда v лежит на линии, проходящей по стороне квадрата). Для любой точки на этих поверхностях стоимость будет бесконечной, потому что, как показано в третьем примере на рис. 16.8, вершина v будет лежать на одном из ребер между другими вершинами (и введенные нами ограничения присвоят этим недопустимым конфигурациям бесконечную стоимость).

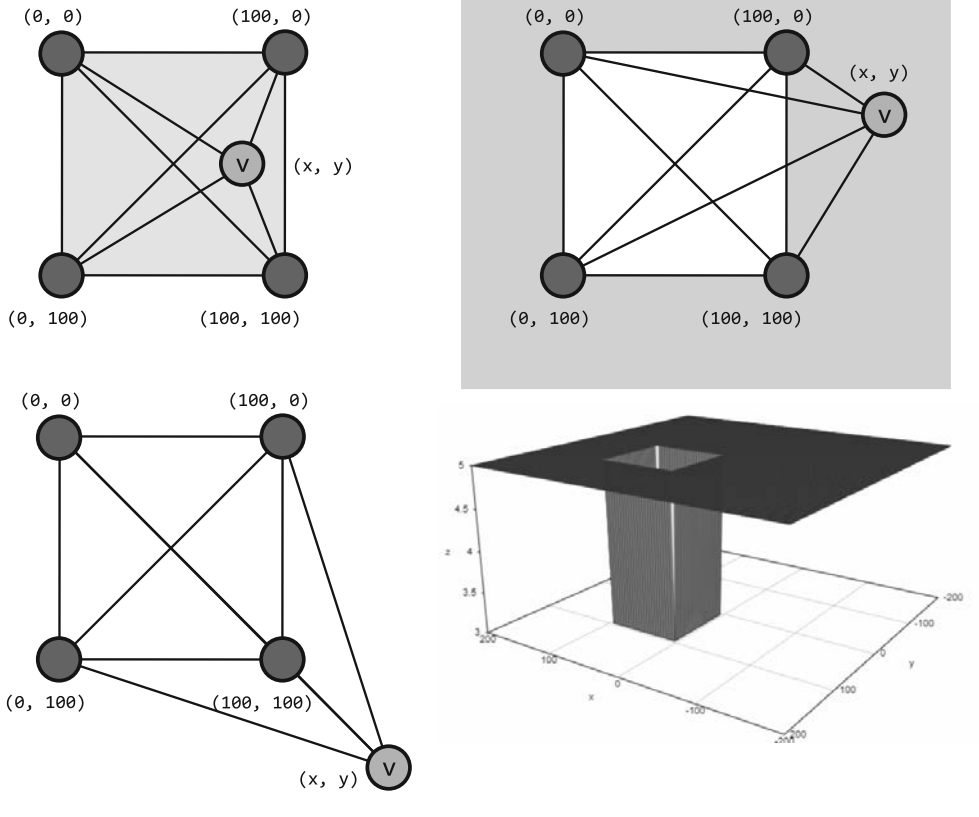


Рис. 16.8. Функция стоимости, отражающая «количество пересечений» для конкретного представления K_{45}. Положение четырех вершин зафиксировано, и только вершина v может свободно перемещаться. Если переместить v внутрь периметра квадрата, образованного другими вершинами, то количество пересечений неизменно будет равно 3; вне квадрата оно становится равным 5. Обратите внимание, что здесь имеются поверхности разрыва!

Более того, как бы ни старались, мы не сможем найти положение вершины v, гарантирующее представление с минимально возможным числом пересечений, потому что положение остальных четырех вершин не является оптимальным для прямолинейного представления K_5.

Основная причина заключается в том, что, если рассмотреть более крупную задачу поиска наилучшего представления (без фиксации положений вершин), будет достигнут локальный минимум функции стоимости: точка или область в n-мерном пространстве задачи, где функция стоимости имеет более низкое значение, чем в окрестностях, но не самое низкое из возможных.

Если бы можно было взять двумерную проекцию функции стоимости на общую конфигурацию графа (а не на конкретное представление на рис. 16.8), то гипотетически она могла бы выглядеть примерно так, как показано на рис. 16.9.

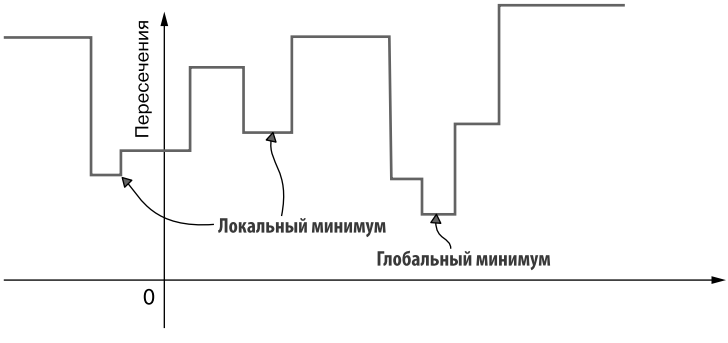


Рис. 16.9. Локальные и глобальные минимумы одномерной функции

16.2.2. Ступенчатые функции и локальные минимумы

Наличие локальных минимумов не очень хорошая новость. В идеале хотелось бы иметь один глобальный минимум и функцию стоимости, плавно сходящуюся к нему.

Например, чашеобразная функция, такая как участок конической кривой, показанный на рис. 16.9, подошла бы как нельзя лучше, и с ней прекрасно справилось бы большинство алгоритмов обучения.

Почему? Представьте, что алгоритм оптимизации — это шарик, который катится по поверхности функции стоимости. Шарики бывают разного веса и с разным коэффициентом трения, некоторые шарики застревают на определенных поверхностях, и их нужно подталкивать. Аналогично существуют разные алгоритмы обучения.

Отпустим шарик на поверхность, подобную функции стоимости на рис. 16.9. Высока вероятность, что он просто останется в точке, где его разместили. Если немного подтолкнуть шарик, он может скатиться на дно ямы, соответствующей локальному минимуму, и застрять там, до тех пор пока его не перенесут в другое место.

И наоборот, если отпустить шарик на гладкую поверхность, как на рис. 16.10, например бросив в чашу, то он скатится на дно чаши, может быть, немного поколеблется и в конце концов остановится в самой нижней точке, где сила тяжести не может утянуть его дальше вниз.

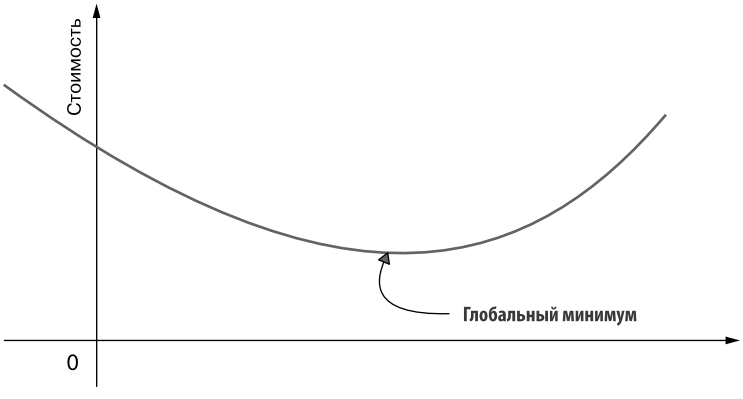


Рис. 16.10. Выпуклая функция с единственным глобальным минимумом

16.2.3. Оптимизация случайной выборки

Используя аналогию с чашей, проблему оптимизации можно рассматривать с двух сторон.

- Функция стоимости подобна чаше: чем ровнее ее стенки на пути к оптимальной стоимости, тем лучше (теоретически) могут работать алгоритмы. Обратите внимание, что можно проложить несколько «путей» между начальной и конечной точками — поиск наилучшего возможного пути является частью решения.
- Алгоритм оптимизации подобен шарику, катящемуся по стенке чаши. Однако, чтобы аналогия была более точной, нужно сказать, что характеристики шарика, особенности его подбрасывания и взаимодействия со стенками чаши являются частью модели алгоритма оптимизации.

А как насчет алгоритма *случайной выборки*? Как можно выразить это в нашей модели? Независимо от функции стоимости, алгоритм делает одно и то же. Представьте, что стенки сделаны из песка или покрыты липкой грязью, поэтому, упав в песок или в грязь, шарик проделывает небольшое углубление и останавливается там, где приземляется. Этот механизм не зависит от формы функции стоимости и работает одинаково даже с гладкой чашеобразной функцией, подобной той, что изображена на рис. 16.10.

На рис. 16.11 показана попытка показать, как работает случайная выборка (которая не выполняет никакой оптимизации после инициализации).

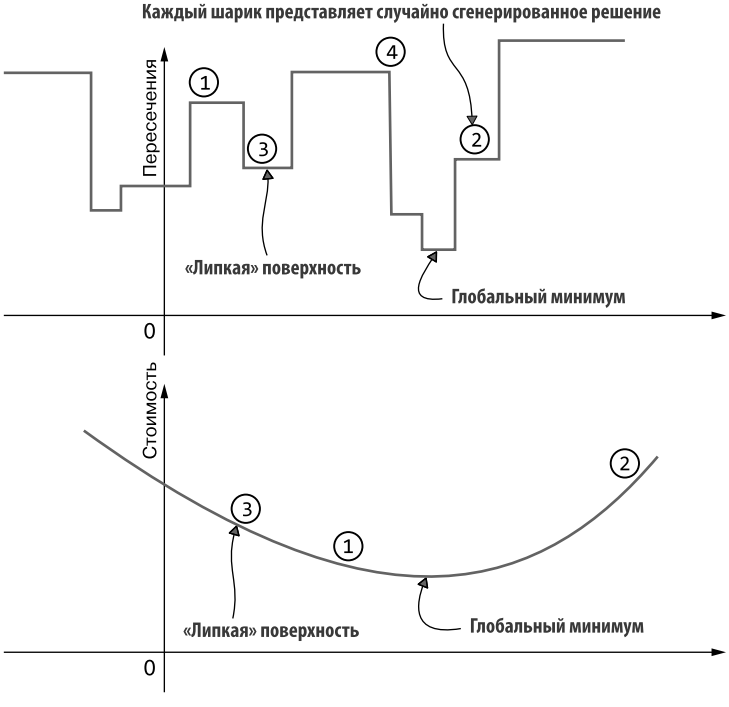


Рис. 16.11. Объяснение работы типичной рандомизированной эвристики с помощью аналогии с чашей

Рандомизированная эвристика подобна бросанию десятков, сотен или тысяч шариков на нашу грязную поверхность (и у нас нет возможности увидеть, где находится финишная черта). Они остаются именно там, где приземлились, и в конце концов приходится просто выбирать ту точку приземления, которая оказывается ближе всего к финишу. Суть состоит в том, чтобы сделать много попыток, надеясь, что хотя бы одна окажется достаточно близкой к оптимальному решению. Конечно, поверхность может быть настолько обширной, что, сколько бы мы ни пытались, не сможем приблизиться к хорошему решению: это случай экспоненциальных задач, когда число возможных решений огромно и объем входных данных достаточно велик.

Более того, нужно быть осторожными при использовании полностью рандомизированного алгоритма. Можно сбросить несколько шариков в одну и ту же позицию и сгенерировать одно и то же решение несколько раз.

Если в этом подходе и есть что-то хорошее (помимо того, что оно довольно дешевое), так это то, что форма функции стоимости не имеет значения. Не нужно прокладывать оптимальную «траекторию», потому что после инициализации шарики не будут «скатываться по стенкам чаши».

В то же время это также и плохо — нет возможности воспользоваться преимуществом наличия хорошей функции стоимости, которая плавно деградирует к глобальному минимуму.

Поразмышляв над аналогией с шариками, мы, вероятно, сможем найти обходной путь. Если вы когда-нибудь играли с шариками на пляже, то, вероятно, знаете, что делать, когда они застревают в песке: их нужно только немного подтолкнуть, чтобы они покатились дальше.

Аналогией с рандомизированными алгоритмами является *локальная оптимизация*. Используем дополнительные эвристики, выполняющие локальную оптимизацию, например пытающиеся перемещать вершины одну за другой на небольшое расстояние от случайно выбранного начального положения, и затем проверим — улучшается ли число пересечений.

Этот алгоритм называется *восхождением на холм*, или, в нашем случае, поскольку мы пытаемся минимизировать, а не максимизировать функцию, его можно назвать *спуском с холма*.

Его модель показана на рис. 16.12. В нашем случае, несмотря на толчок, шарик преодолевает лишь небольшое расстояние (как мы помним, стенки чаши покрыты песком или липкой грязью), но все-таки, возможно, получится некоторое улучшение. Такими действиями мы фактически исследуем функцию стоимости в небольшой области вокруг каждого решения и, найдя позицию, где стоимость ниже, перемещаем наш «шарик» туда.

Если внимательно посмотреть на рис. 16.12, то можно увидеть, что форма функции стоимости действительно имеет значение. В случае с дифференцируемой чашеобразной функцией всегда получается улучшение, со ступенчатой функцией (как в нашем примере) шарики иногда будут застревать в локальных минимумах, а иногда оказываться на широких плато, и поэтому их перемещение вокруг не поможет получить лучший результат.

Следует также отметить, что алгоритм должен попробовать двигаться в обоих направлениях. Случайным образом исследуя область вокруг решения, мы вслепую ищем похожие решения, но без всякого обоснования. Например, глядя на шарики и форму функции стоимости внизу на рис. 16.12, можно заключить, что для улучшения решений 1 и 3 нужно увеличить значение единственного параметра, а для решения 2 — уменьшить. А используя спуск с холма, мы пробуем увеличивать и уменьшать параметр всех решений и просто смотрим, что получится. Для функций с несколькими параметрами предстоит либо опробовать область вокруг текущего решения, либо выбирать случайное направление и двигаться туда, если выбранное движение будет приводить к улучшению.

Применительно к задаче с графом это означает, что мы будем перемещать вершину во всех направлениях и каждый раз вычислять пересечения ребер нового представления.

И если это выглядит плохо уже с двумерной функцией стоимости, то напомню, что с увеличением размерности пространства поиска в полный рост встает проклятие

размерности (см. главы 9–11), и для настройки графа с n вершинами придется подобрать $2n$ параметров.

Однако можно попробовать и более эффективный подход.

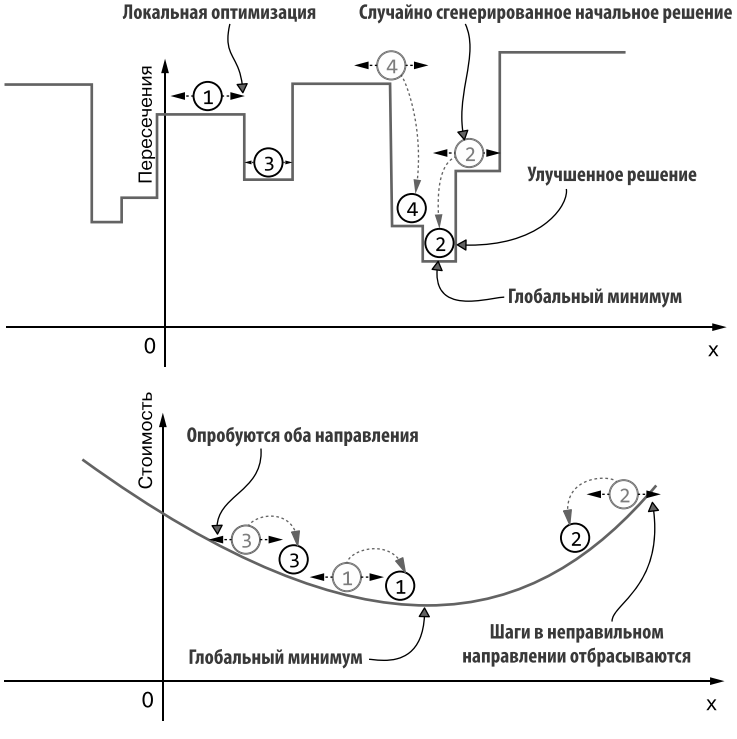


Рис. 16.12. Работу обычной рандомизированной эвристики с локальной оптимизацией можно объяснить с помощью аналогии с шариками. Обратите внимание, что нужно попытаться изменить x в обоих направлениях и проверить, приведет ли это к улучшению

16.3. ГРАДИЕНТНЫЙ СПУСК

Зачем действовать наугад, пытаясь выполнить локальную оптимизацию? Напомню, что для графа с n вершинами это означает оптимизацию в $2n$-мерном пространстве и, следовательно, попытки двигаться во всех направлениях[1] случайным образом... это огромный объем работ!

[1] Здесь мы говорим не о перемещении воображаемого шарика по поверхности функции стоимости: это обобщенное описание, подходящее для всей категории подобных задач оптимизации. Для конкретной задачи поиска представления графа с минимальным числом пересечений ребер способ исследования окружающей области функции стоимости в действительности заключается в перемещении всех вершин.

При взгляде на двумерный пример, где функция стоимости зависит от одной переменной, направление движения может показаться довольно очевидным! Но как узнать в многомерном пространстве, где нельзя визуализировать форму поверхности, какие параметры следует настраивать и в каком направлении?

Математическое решение такой задачи называется *градиентным спуском*. Градиентный спуск — это метод оптимизации, который при определенных условиях может применяться к различным категориям задач оптимизации.

Идея проста: можно оценить наклон функции в заданной точке и двигаться в направлении наибольшего убывания функции. Название метода основано на том, что направление наибольшего наклона дифференцируемой функции определяется ее градиентом.

На рис. 16.13 показан пример дифференцируемой функции с одной переменной.

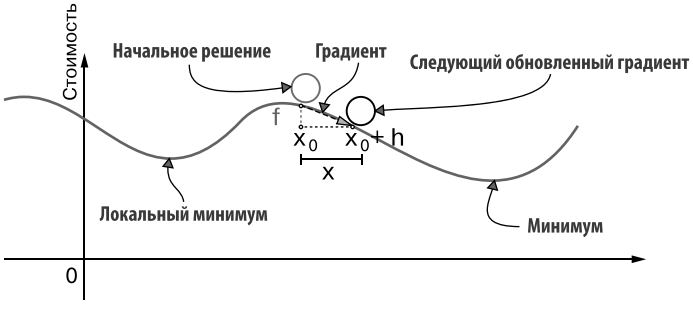


Рис. 16.13. Градиентный спуск: Δf — изменение функции стоимости, вызванное изменением Δx ее параметра

Прежде чем переходить к более формальному обсуждению, посмотрим, как можно представить эту идею в нашем примере с шариками: допустим, разрешено подталкивать шарики и они могут продвинуться на небольшое расстояние, прежде чем снова остановятся в песке. Как при игре с шариками на реальном песке, нужно несколько раз толкнуть шарики, чтобы достичь цели, и каждый раз они перемещаются только на небольшое расстояние. Если подталкивать их в правильном направлении, они будут постепенно приближаться к финишу (к цели нашей оптимизации). Но, чтобы усложнить игру, допустим, что видеть можно только короткую часть пути рядом с текущим положением шарика.

Градиентный спуск формально описывается с помощью исчисления. Не волнуйтесь, если вы еще не знакомы с исчислением, потому что оно вам не понадобится для применения градиентного спуска: есть много отличных библиотек, которые реализуют все необходимое. На самом деле задумываться о разработке собственной версии — плохая идея, потому что этот алгоритм нуждается в тонкой настройке и высокой оптимизации, чтобы он мог использовать вычислительные мощности графического процессора.

16.3.1. Математика градиентного спуска

Если вам доводилось слушать лекции по математическому анализу, то вы, вероятно, помните понятие производной: для заданной непрерывной функции *f(x)* одной переменной ее производная определяется как отношение между тем, как *f* изменяется в ответ на небольшое изменение ее аргумента *x*. Формально это записывается так:

$$f'(x) = \frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Это значение может быть конечным, бесконечным или даже неопределенным для данной функции и конкретного значения *x*. Если для функции *f* всегда определена ее первая производная, то *f* называется дифференцируемой функцией.

Для дифференцируемых функций существуют формулы, позволяющие найти точное математическое определение их производных, которые, в свою очередь, сами являются функциями. Например, если $f(x) = x$, то $f'(x) = 1$ (постоянная функция). Производная квадратичной функции $f(x) = x^2$ равна $f'(x) = 2x$, а производная экспоненциальной функции $f(x) = e^x$ равна $f'(x) = e^x$ (да-да, той же самой экспоненциальной функции!).

Существует много интересных геометрических интерпретаций[1] производных функций, но мы не будем перечислять их здесь.

Самый важный результат, с моей точки зрения, заключается в том, что если вычислить значение первой производной функции в данной точке, то оно скажет само, растет ли функция в ней и насколько. Другими словами, оно скажет, увеличится, уменьшится или останется прежним значение *f(x)* при небольшом увеличении *x*.

Этот прием можно применить к нашему алгоритму оптимизации. Например, как показано на рис. 16.13, если вычислить первую производную функции стоимости в точке x_0, получим отрицательное значение, которое скажет нам, что *f* растет с уменьшением *x*. Поскольку нам-то нужно получить меньшее значение *f*, необходимо увеличить *x*.

Этот шаг можно повторять снова и снова, следуя по нисходящей траектории вдоль поверхности функции стоимости.

Несмотря на кажущуюся простоту, вычисления в многомерных пространствах становятся намного более сложными: для *n*-мерной функции градиент *g* в данной точке — это вектор, компонентами которого являются частные производные функции, вычисленной в этой точке.

Например, для двумерного пространства (рис. 16.15 показывает, как это можно представить) частная производная $g(x, y)$ определяется вдоль *x* как:

$$g'_x(x, y) = \frac{\partial g}{\partial x} = \lim_{h \to 0} \frac{g(x+h, y) - g(x, y)}{h}.$$

[1] Обратите внимание, что существуют производные более высокого порядка, но здесь мы остановимся только на производной первого порядка.

И градиент g в точке $P_0 = (x_0, y_0)$ определяется как вектор с одним столбцом и двумя строками, компонентами которого являются частные производные функции g по x и y, вычисленные в точке P_0:

$$\nabla g(P_0) = \begin{bmatrix} \frac{\partial g}{\partial x}(P_0) \\ \frac{\partial g}{\partial y}(P_0) \end{bmatrix}.$$

С геометрической точки зрения градиент функции можно интерпретировать как вектор, указывающий в направлении наибольшего роста функции. Вот почему градиентный спуск на самом деле использует *отрицательный градиент* $-\nabla g$, который просто направлен в противоположную сторону.

16.3.2. Геометрическая интерпретация

В листинге 16.4 приводится краткая реализация алгоритма градиентного спуска.

Листинг 16.4. Метод gradientDescent

Метод gradientDescent принимает функцию f, начальную точку P0, скорость обучения alpha и максимальное количество выполняемых шагов maxSteps. Возвращает точку в пространстве — в идеале точку, где f имеет наименьшее значение (локальное или глобальное)

Цикл, выполняющий основные шаги алгоритма не более maxSteps раз

```
function gradientDescent(f, P0, alpha, maxSteps)
  for _ in {1..maxSteps} do
    for i in {1.. |P0|} do
      P[i] ← P0[i] – alpha × derivative(f, P0, i)
    if P == P0 then
      break
    P0 ← P
  return P0
```

Цикл по координатам точки P0

Создаем новую точку P, каждая координата которой получает значение соответствующей координаты P0 с небольшим значением дельты, вычисленным из градиента f: в частности, нужно вычислить частную производную f по ее i-й координате, а затем значение этой производной в точке P0. Значение градиента умножается на alpha — скорость обучения

Если все производные равны 0, то это означает, что P0 находится в области плато или в точке минимума, поэтому градиентный спуск не может найти улучшения. На самом деле здесь следовало бы проверить норму разности, сравнив ее с некоторым порогом точности, потому что компьютерная арифметика имеет конечную точность, а также потому, что для малого градиента улучшение может оказаться незначительным, не стоящим затраченных вычислительных ресурсов

В конце каждого шага текущая точка перемещается в новую позицию

Важно понимать, что у нас нет полного представления о функции стоимости при выполнении градиентного спуска, и мы не «катимся» по поверхности; скорее, на каждом шаге мы лишь вычисляем, насколько нужно изменить входные переменные, в зависимости от наклона поверхности в этой точке. Глядя на рис. 16.14, можно получить представление о выполняемых шагах, а на рис. 16.15 — понять, как это выглядит для двумерной функции.

В аналогии с шариками поверхность, по которой они катятся, как будто окутана густым туманом, и мы можем видеть в разные стороны только на несколько десятков

сантиметров: этого достаточно, чтобы увидеть, куда прицелиться и осторожно выполнить следующий шаг, потому что, если слишком сильно толкнуть шарик, он может проскочить дальше, чем нужно, или покатиться в неправильном направлении.

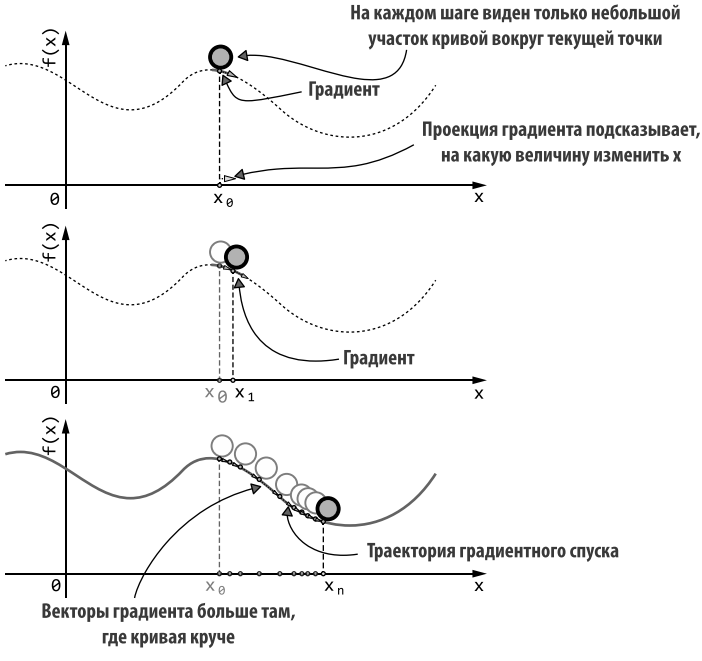


Рис. 16.14. Несколько шагов градиентного спуска: обратите внимание, что шарик преодолевает путь, полученный объединением векторов градиента после каждого обновления. Длинные векторы означают, что градиент был больше (и наклон кривой круче), поэтому, в свою очередь, величина Δx обновления переменной на этом шаге тоже будет больше

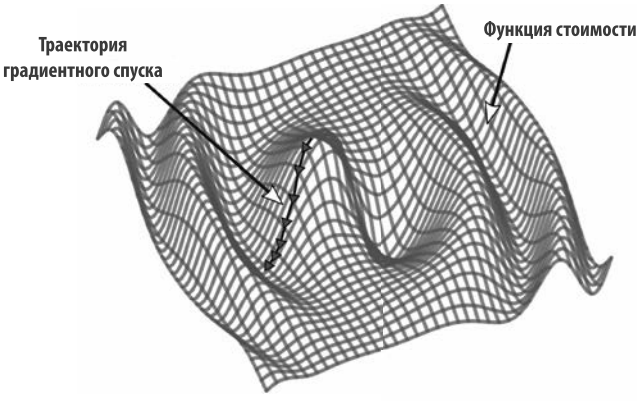


Рис. 16.15. Градиентный спуск применительно к функции двух переменных. Функция стоимости определяет поверхность

Сам метод удивительно прост и понятен, не так ли? Это итерационный алгоритм оптимизации, поэтому его реализация включает главный цикл, выполняющийся не более заданного числа раз. Количество выполнений определяется аргументом, гарантирующим, что алгоритм не зациклится (как мы увидим далее, в некоторых ситуациях такое вполне возможно).

Шаг обновления повторяется снова и снова, пока не будет достигнут минимум функции или заданное максимальное число повторений. Реализация самого шага тоже выглядит просто: на ней вычисляется градиент каждой координаты входной функции посредством определения частной производной первого порядка от f по каждому из направлений в пространстве f (пространстве задачи).

Как уже упоминалось, алгоритм может остановиться по достижении минимума. Одним из наиболее важных постулатов в исчислении является теорема Ферма[1], которая доказывает, что точка в области определения дифференцируемой функции является минимумом или максимумом тогда и только тогда, когда производная этой функции равна нулю в этой точке; следовательно, можно просто сравнить все частные производные с нулем (или, что более практично, с некоторым пороговым значением точности).

Используя градиент функции f, чтобы решить, насколько (и в каком направлении) двигаться, мы, естественно, делаем большие шаги, когда f изменяется быстро, и маленькие, когда f изменяется медленно. В примере шарик будет катиться быстро на крутом прямом участке, но нам нужно быть осторожными вблизи поворотов, чтобы не сойти с верного пути.

В отношении начальной точки P_0 может возникнуть вопрос: как ее выбрать? Есть разные способы, но если у вас нет фактов из предметной области, требующих обратного, то лучше выбрать ее случайным образом и, возможно, применить оптимизацию несколько раз, начиная каждый раз с новой, случайно выбранной точки и запоминая лучший результат.

Видите? Мы снова вернулись к случайной выборке, но на этот раз применили сложную эвристику локальной оптимизации после взятия каждой выборки.

16.3.3. Когда применяется градиентный спуск

Для применения градиентного спуска функция стоимости должна быть дифференцируемой, по крайней мере в окрестностях точек, где вычисляется градиент.

Пригодится также знание точной формулы функции, которую требуется оптимизировать. Это позволит выражать частные производные с помощью математических формул и точно вычислять градиенты. Однако в отсутствие определения оптимизируемой функции всегда можно прибегнуть к формальному определению производных как математических пределов и вычислять значение градиента, явно

[1] Но не самым известным! Хотя можно доказать, что именно эта теорема является наиболее важной.

оценивая отношение между Δf и Δx_i для все более малых приращений координат в пространстве задачи.

Один из вопросов, который у вас может возникнуть: если есть определение f и она дифференцируемая, то зачем проводить итерационную оптимизацию? Разве нельзя просто найти точный минимум с помощью исчисления?

В теории это, конечно, возможно и даже реализуемо, по крайней мере, для низкоразмерных пространств и некоторых видов функций. Однако в многомерных пространствах поиск точных решений трудно автоматизировать и нелегко даже просто вычислять их. Количество уравнений, необходимых для аналитического определения глобального минимума, растет экспоненциально с размером задачи. Причем эти функции могут иметь сотни, тысячи и даже бесконечное число локальных минимумов (вспомните, например, $\sin(x + y)$ или даже $x \times \sin(y)$), а для автоматизации поиска глобального оптимума нужно проверить все эти точки.

В общем случае градиентный спуск работает хорошо, когда есть возможность разработать функцию стоимости, имеющую глобальный минимум или хотя бы несколько локальных минимумов (лучше, если в них функция стоимости имеет примерно одинаковую величину). Как вы увидите в разделе 16.4, именно поэтому он отлично работает с функциями стоимости, которые разрабатываются для обучения с учителем.

16.3.4. Задачи с градиентным спуском

В листинге 16.4 следует отметить одну важную деталь: в алгоритм передается параметр `alpha`, определяющий скорость обучения. Это гиперпараметр алгоритма, регулирующий размер выполняемых шагов. Как и в аналогии с шариком: когда нет четкого видения всего пути, который нужно преодолеть, большие шаги могут ускорить движение, но также могут направить шарик в неверном направлении. То же происходит и при оптимизации методом градиентного спуска: делая большие шаги, можно перешагнуть минимум (рис. 16.16, *А*) или, что еще хуже, в некоторых ситуациях вызвать зацикливание или даже уход от лучшего решения, как показано на рис. 16.16, *Б*.

Рис. 16.16. Градиентный спуск: Δf — изменение функции стоимости, вызванное изменением Δx ее параметра

И наоборот, когда параметр `alpha` слишком мал (см. рис. 16.16, В), сходимость может быть слишком медленной, и алгоритм оптимизации не сможет достичь минимума за разумное время (или за максимально допустимое количество итераций).

Какие значения `alpha` слишком велики или слишком малы, зависит от контекста и, в частности, от конкретной оптимизируемой функции.

Если бы не было возможности передать параметр, определяющий скорость обучения, то для функций стоимости, подобных изображенной на рис. 16.16, *Б*, где кривая имеет крутой наклон, оптимизация не сходилась бы к минимуму (а скорее расходилась бы). В то же время для таких случаев, как на рис. 16.14, *В*, сходимость была бы слишком медленной, или алгоритм мог бы застрять в локальных минимумах.

Имея возможность определять скорость обучения, можно настроить[1] гиперпараметр `alpha` и адаптировать алгоритм оптимизации к оптимизируемой функции.

Однако лучшим решением является корректировка значения переменной `alpha`, например уменьшение ее значения с каждым следующим шагом. Первоначально оно может иметь достаточно большое значение, чтобы позволить оптимизации быстро исследовать большую область и, возможно, выскочить из локальных минимумов, а затем становится все меньше и меньше и, в конце концов, за счет выполнения точной настройки `alpha`, уменьшается вероятность колебаний вокруг стационарных точек (минимумов).

Еще один отличный вариант — ввод понятия импульса. В этом случае следующий шаг можно основывать не только на последнем градиенте: градиентный спуск с импульсом может сглаживать величину обновления, вычисляя значение дельты как линейную комбинацию последних нескольких градиентов (при этом более старые градиенты получают меньший вес, чем более новые).

Можно предположить, импульс (как это происходит в кинематике) означает, что если скорость алгоритма была высокой и он обновлял координату большими приращениями, то при изменении наклона кривой скорость будет сглаживаться, но не резко.

Самая простая формула добавления импульса в правило обновления координат точек может выглядеть так:

$$P_{t+1} \leftarrow \beta \times P_t - \alpha \times (1 - \beta) \times \nabla g(P_t),$$

где P_t — точка в пространстве задачи, точнее, точка, достигнутая алгоритмом в момент времени t. Чем выше коэффициент бета, тем плавнее (и медленнее) будет происходить обновление:

$$P_2 \leftarrow \beta \times P_1 - \alpha \times (1 - \beta) \times \nabla g(P_1) = \beta^2 \times P_0 - \alpha \times \beta \times (1 - \beta) \times \nabla g(P_0) - \\ - \alpha \times (1 - \beta) \times \nabla g(\beta \times P_1 - \alpha \times (1 - \beta) \times \nabla g(P_1)).$$

---

[1] Настройка гиперпараметров алгоритма, как будет показано далее, является одной из задач использования эвристики. У этих параметров не существует единого значения, подходящего для всех случаев, и их настройку обычно трудно автоматизировать. Вот та область, где опыт алгоритмиста действительно имеет значение.

Так после двух шагов, если считать, что $\beta = 0{,}99$, то на 98 % значение P_2 определяется точкой P_0; и наоборот, если $\beta = 0{,}1$, то P_0 влияет на P_2 только на 1 %.

16.4. ПРИМЕНЕНИЕ ГРАДИЕНТНОГО СПУСКА

Как упоминалось выше, градиентный спуск — это метод оптимизации, который, опираясь на функцию стоимости, помогает найти решение (точку в пространстве задачи), имеющее оптимальную (или почти оптимальную) стоимость.

Он лишь часть процесса решения задачи, и в то же время его можно применять к нескольким различным задачам и методам.

Общий алгоритм в первую очередь зависит, как было показано, от используемой функции стоимости, а также от цели оптимизации.

Итак, мы уже обсудили оптимизацию функции стоимости для поиска самого дешевого решения четко определенной задачи и выяснили, что алгоритмы из этой категории значительно выигрывают от применения градиентного спуска, когда есть возможность описать дифференцируемую функцию стоимости.

Однако в последнее время популярность приобрела другая категория алгоритмов, использующих градиентный спуск: алгоритмы обучения.

ПРИМЕЧАНИЕ

Я всегда считал термин «машинное обучение» немного обманчивым, потому что он предполагает вовлеченность машин, которые могут учиться так же, как люди, что, к сожалению, неверно, хотя некоторое сходство определенно есть.

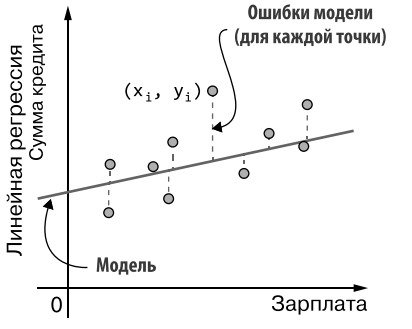
При применении градиентного спуска к решению таких задач, как задачи о коммивояжере или поиск представления графа, у нас есть статическая (обычно огромная) область и цель — найти точку в этой области. В машинном обучении вместо этого у нас есть набор данных, и наша цель — понять его, «обучая» модель, описывающую набор данных и (что более важно) обобщающую входные данные, которых не было в исходном наборе данных.

Возьмем, к примеру, обучение с учителем (наиболее яркие примеры которого показаны на рис. 16.17); это область, где широко применяется градиентный спуск!

Целью обучения с учителем является получение математической модели[1], которая может кратко описать набор данных и способна предсказать результат для новых, ранее не наблюдавшихся входных данных, будь то реальное значение (линейная регрессия), категория (логистическая регрессия) или метка (кластеризация).

---

[1] На самом деле всего лишь функции от области для диапазона. Независимо от того, насколько сложной может быть эта функция, она все еще представляет собой детерминированное отображение входных данных (возможно, многомерных) в выходные.



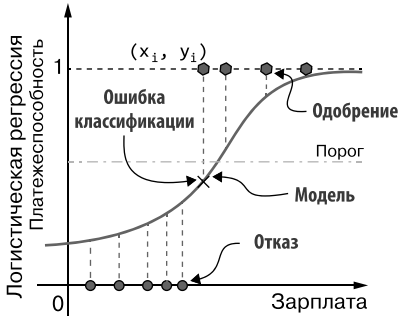
Модель: $f(x) = m \times x + b$

Функция потерь:

$$J(m,b) = \frac{1}{n} \times \sum_{i=1}^{n} (y_i - m \times x_i - b)^2$$

Gradient:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial b} \\ \frac{\partial J}{\partial m} \end{bmatrix} = \begin{bmatrix} -\frac{2}{n} \times \sum_{i=1}^{n} (y_i - m \times x_i - b) \\ -\frac{2}{n} \times \sum_{i=1}^{n} x_i \times (y_i - m \times x_i - b) \end{bmatrix}$$

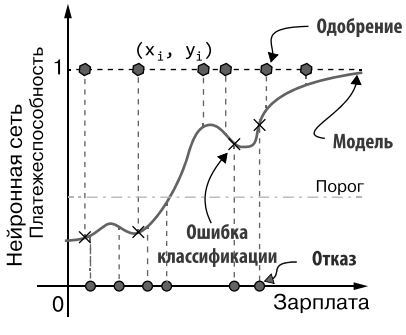


Модель: $h_{m,b}(x) = \frac{1}{1 + e^{m \times x + b}}$

Функция потерь:

$$J(m,b) = \frac{1}{n} \sum_{i=1}^{n} [y_i log(h(x_i)) + (1 - y_i) log(1 - h(x_i))]$$

Gradient: $\nabla J = \begin{bmatrix} \frac{\partial J}{\partial b} \\ \frac{\partial J}{\partial m} \end{bmatrix} = \begin{bmatrix} \frac{1}{n} \times \sum_{i=1}^{n} (h_{m,b} - y_i) \\ \frac{1}{n} \times \sum_{i=1}^{n} x_i \times (h_{m,b} - y_i) \end{bmatrix}$



Модель:

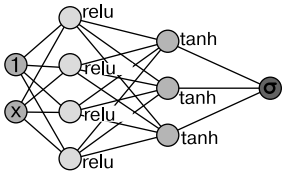


Рис. 16.17. Несколько подходов, используемых в машинном обучении с учителем. Во всех этих методах для обучения используется градиентный спуск. Обратите внимание, что модели выбираются произвольно и возможны другие варианты функций потерь. Градиенты и функции потерь для нейронных сетей для краткости опущены. Здесь показано только обучение с учителем, но вообще существуют также алгоритмы кластеризации, использующие градиентный спуск

Во всех этих типах обучения есть один дополнительный шаг, которого не было в задачах оптимизации, представленных выше в этой главе. Там мы просто исследовали пространство задачи. Теперь же нужно выбрать модель для использования, и на самом деле это нужно сделать в первую очередь.

Чтобы лучше объяснить ситуацию, углубимся в некоторые детали линейной регрессии.

16.4.1. Пример: линейная регрессия

Упомянув обманчивость названий, нельзя не обратиться к линейной регрессии. История происхождения названия этой методики обучения весьма увлекательна, и ее стоит погуглить. Надеюсь, я пробудил ваше любопытство.

Но что действительно важно, так это то, что суть линейной регрессии заключается в поиске модели, которая описывает отношение между одним или несколькими входными данными (также известными как *независимые переменные*) и реальным числом, возвращаемым моделью *(зависимая переменная)*.

Эта «модель» уже несколько раз упоминалась, ее можно увидеть на рис. 16.17, поэтому вы вполне резонно можете спросить, что это такое.

Модель — это категория математических функций, которые выбираются для аппроксимации реального отношения между зависимыми и независимыми переменными в наборе данных.

Например, у нас может быть набор данных, отражающий некоторые характеристики автомобилей (год выпуска, мощность двигателя, пробег и т. д.) с их рыночной ценой, и мы хотели бы узнать, как связаны характеристики с ценой, чтобы можно было ввести описание автомобиля (в терминах тех же независимых переменных, что и в наборе данных), который мы заметили у дилера, и проверить, насколько запрашиваемая цена является справедливой (в надежде избежать обмана и не заплатить слишком много).

Для этого нужно выбрать модель, которая, по нашему мнению, лучше всего соответствует данным[1]. Для простоты ограничимся функциями с одним параметром (это может быть мощность двигателя). Как показано на рис. 16.18, можно выбрать постоянную функцию ($y = m$, линия, параллельная оси X), общую линию вида $y = mx + b$, квадратичную кривую ($m_1 \times x^2 + m_2 \times x + b$) или еще более сложную модель.

Как правило, чем проще модель, тем меньше точек данных требуется для ее «обучения», потому что после выбора сложности модели получаем категорию функций и возникает необходимость подбора параметров, которые подскажут, какая именно функция в этой категории лучше подходит для исследуемого набора данных.

Например, если выбрать общую линию, нужно будет определить значения параметров m и b: это может быть $y = x + 1$ или $y = -0{,}5 \times x + 42$.

Выбор производится с помощью *обучения*, которое представляет собой не что иное, как применение градиентного спуска.

Фактически в линейной регрессии определяется функция стоимости (обычно называемая в машинном обучении *функцией потерь*), которая измеряет расстояние

[1] Обычно опробуется автоматизированным способом несколько разных моделей и выбирается та из них, которая хорошо предсказывает цену, но эта тема выходит далеко за рамки нашего обсуждения.

между значением, которое предсказано моделью для зависимой переменной, связанной с каждой точкой в наборе данных[1], и фактическим ее значением в данных.

Обычно в роли такой функции используется сумма, полученная методом наименьших квадратов ошибок, или ее вариант. Как показано на рис. 16.17 и 16.19, минимизируется квадрат расстояния по оси Y между каждой точкой и линией модели, и это дает выпуклую чашеобразную функцию с глобальным минимумом — как мы уже знаем, это идеальный вариант для градиентного спуска!

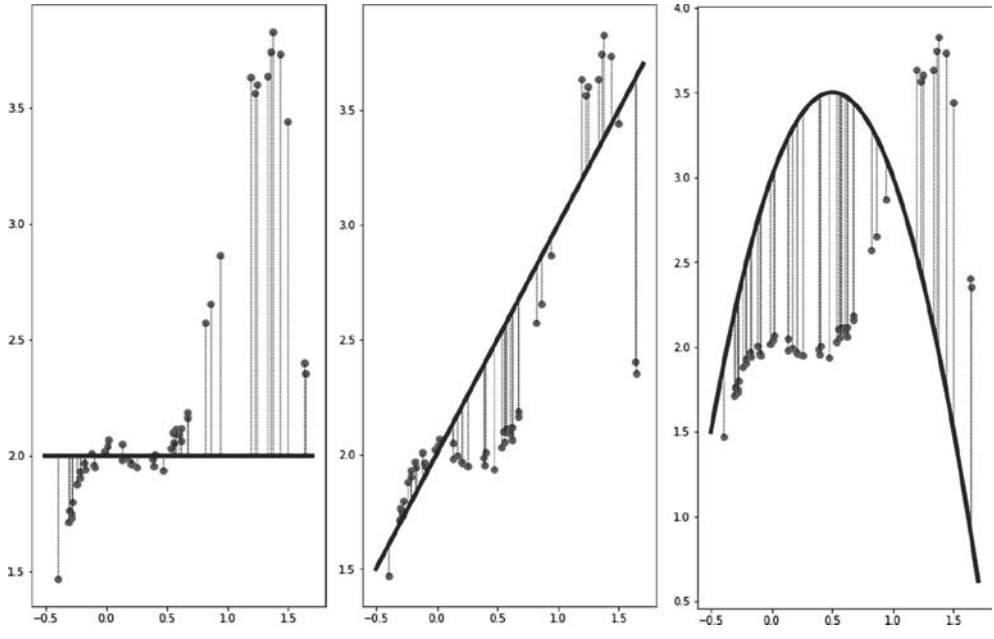


Рис. 16.18. Линейная регрессия на наборе данных (точки данных обозначены точками графическими) с использованием все более сложных моделей: константа, прямая линия и квадратичная кривая. Как видите, модель более высокого порядка не обязательно лучше соответствует данным

Исходя из рассмотрения рис. 16.17 и 16.19 следует подчеркнуть один важный факт: функция потерь зависит от параметров m и b[2], а не от точек в наборе данных, поэтому частные производные вычисляются относительно параметров модели, которые затем обновляются градиентным спуском.

---

1 На самом деле подмножество, представляющее набор данных во время обучения, называется обучающим набором. Не будем вдаваться в подробности того, как и почему выбираются данные для него, просто помните, что очень важно оставить некоторые точки из исходного набора для последующего тестирования, чтобы оценить качество модели.

2 При использовании линейной модели: обычно вектор параметров модели обозначается как W или Θ; напомню, что на самом деле эти параметры являются целью алгоритма обучения.

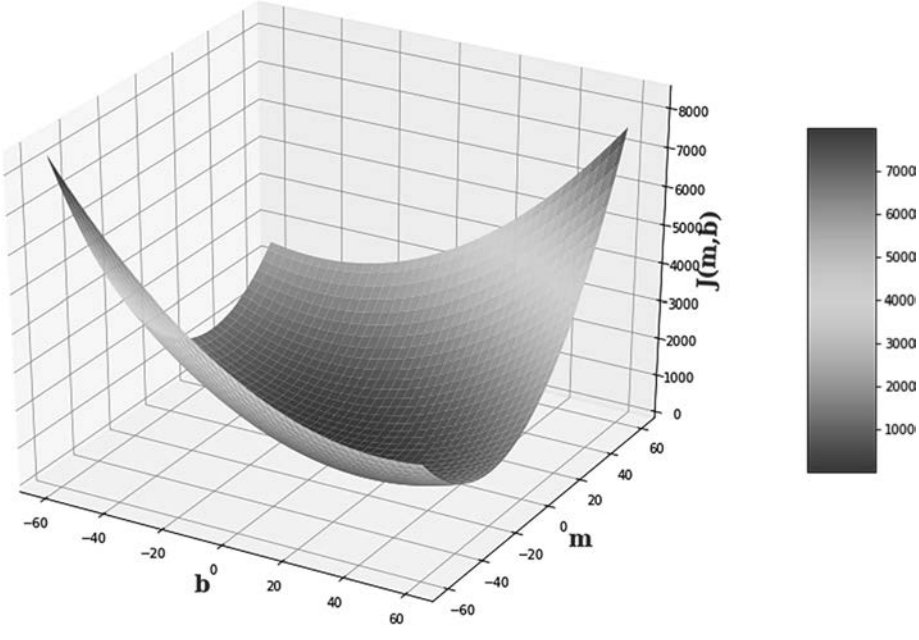


Рис. 16.19. Функция стоимости для примера на рис. 16.18, *справа*. При использовании суммы квадратов ошибок стоимость является функцией от m и b

О линейной регрессии и обучении с учителем можно рассказать так много, что для этого потребовался бы отдельный опус!

И действительно, есть очень много книг, которые вы можете прочитать, если захотите углубиться в машинное обучение. Вот те из них, которые я лично считаю чрезвычайно полезными.

- *Grokking Machine Learning*[1], хорошая отправная точка для начинающих, написанная Луисом Серрано (Luis Serrano; Manning Publications, 2019). Лучшего гида по машинному обучению и желать не надо.
- *Grokking Deep Learning*, автор Эндрю В. Траск (Andrew W. Trask) из DeepMind (Manning Publications, 2019), отличное введение, идеально подходящее для знакомства с миром глубокого обучения[2].
- *Deep Learning with Python*, написана Франсуа Шолле (François Chollet), автором библиотеки Keras (Manning Publications, 2017); из книги вы узнаете, как использовать эту библиотеку для создания моделей и генераторов классификации изображений и текста[3].

---

1 *Серрано Л.* Грокаем машинное обучение. — СПб.: Питер, 2024.

2 *Траск Э.* Грокаем глубокое обучение. — СПб.: Питер, 2022.

3 *Шолле Ф.* Глубокое обучение на Python. — СПб.: Питер, 2023.

- *Deep Learning with JavaScript*, авторы Шэнкуинг Цэй (Shanquing Cai) и др. (Manning Publications, 2020), если вы хотите создавать модели для Интернета, которые работают в браузере, используйте библиотеку Tensorflow.js, написанную основными авторами книги[1].

Есть и другие замечательные книги. Невозможно перечислить их все здесь, но упомянутые выше послужат вам хорошей отправной точкой.

16.5. ГРАДИЕНТНЫЙ СПУСК ДЛЯ ПОИСКА ПРЕДСТАВЛЕНИЯ ГРАФА

Итак, после подробного обсуждения работы градиентного спуска, а также оценки его сильных и слабых сторон пришло время разобраться, как применить его к нашему конкретному случаю, чтобы получить эвристику для поиска прямолинейного представления графов с минимальным количеством пересечений между ребрами.

И ответ таков... градиентный спуск не способен нам помочь. Если посмотреть на рис. 16.7–16.12, то станет ясно, что функция стоимости для «минимального количества пересечений» имеет ступенчатую форму с большими плато (где градиент равен нулю) и резкими перепадами.

Возникает вопрос: а зачем мы тогда затронули тему градиентного спуска? Некоторые читатели могут догадаться о том, что будет на следующем шаге, но если у вас нет времени, чтобы перебрать в уме все, что узнали в последних двух главах, давайте углубимся в следующую задачу.

Прежде чем описать ее, позвольте подчеркнуть, что обсуждение в разделах 16.3 и 16.4 позволило вам лучше разобраться в функциях стоимости и дало их полуформальную характеристику. Даже если бы дело ограничилось только этим, и то время нельзя было бы считать потраченным впустую, потому что обретенные знания помогут описать и понять алгоритмы, представленные в этой и в следующих двух главах.

Но и это еще не все. Если вы прочитали главу 15, то должны помнить, как в разделе 15.3 шло рассуждение о том, что означает хорошее представление и чем одно представление может быть лучше другого. Пересечения ребер являются, пожалуй, самой важной частью этого понятия, но есть и другие соображения, например, что смежные вершины должны располагаться близко друг к другу, а когда между парой вершин нет ребра, их можно и нужно рисовать далеко друг от друга.

Умение создавать эстетически привлекательное изображение графа в чем-то не менее или даже более важно, чем простое уменьшение количества пересечений ребер.

---

[1] *Шолле Ф., Нильсон Э., Байлесчи С., Шэнкуинг Ц.* JavaScript для глубокого обучения: TensorFlow.js. — СПб.: Питер, 2021.

Это умение может сделать диаграмму более ясной и понятной, а также помочь оптимально использовать доступное пространство (особенно на динамических веб-сайтах) или создавать более содержательные диаграммы.

И последнее, но не менее важное: эстетически приятный внешний вид можно выразить с помощью лучшей функции стоимости, более гладкой, к которой можно применить алгоритмы оптимизации, такие как градиентный спуск.

16.5.1. Другие критерии

В прямолинейных диаграммах вершины можно представить как ионы, электрически заряженные частицы. Когда между двумя вершинами есть ребро, частицы притягиваются друг к другу (как если бы они имели противоположные заряды), а вершины, не соединенные ребрами, отталкиваются.

Используя эту аналогию, можно попытаться найти точку равновесия системы, такое расположение частиц, при котором все силы уравновешивают друг друга и система может сохранять стабильное состояние. Иногда вместо явного вычисления[1] точки равновесия можно попробовать аппроксимировать ее, моделируя эвристику эволюции системы.

Оказывается, для построения представлений графов существует целый класс алгоритмов, использующих этот принцип: так называемые *силовые алгоритмы визуализации графов*.

Цель этих алгоритмов — расположить вершины графа в двумерном пространстве так, чтобы соседние вершины находились более или менее на одном и том же расстоянии, соответственно, все ребра имели одинаковую длину в плоскости, при этом, конечно же, пересечений ребер должно быть как можно меньше. Достигается такое расположение путем вычисления сил притяжения между соседними вершинами и сил отталкивания между всеми несмежными вершинами. Основываются расчеты на относительных положениях вершин с последующим обновлением этих положений исходя из вычисленных сил и некоторых дополнительных параметров; все действия направлены на минимизацию энергии всей системы.

Чтобы было понятнее, в роли физического аналога ребер можно использовать пружины (или гравитацию) и слабое электрическое отталкивание[2] между всеми парами вершин. Обратите внимание, что все эти силы зависят от расстояний между вершинами — их можно заменить другими формулами, главное — чтобы эти

---

[1] Математическое решение включает поиск нулей дифференциальных уравнений, описывающих систему.

[2] В действительности электрические силы между частицами, конечно, на порядки сильнее гравитации. Но цель модели не в том, чтобы быть точной, а в том, чтобы быть реалистичной и интуитивно понятной, а также позволять повторно использовать приобретенные ранее знания.

характеристики сохранялись. Изображение на рис. 16.20 помогает понять, как могут работать такие системы.

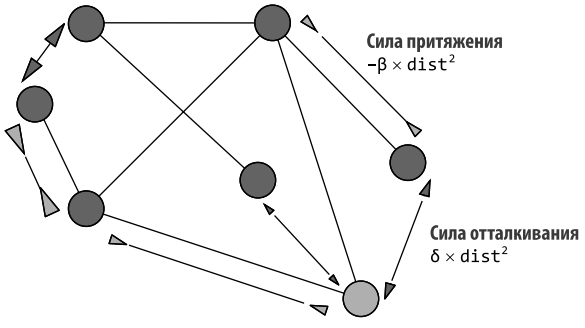


Рис. 16.20. Силовой алгоритм визуализации с использованием физического моделирования на основе сил притяжения и отталкивания, действующих на вершины. Алгоритм обеспечивает эстетически привлекательное представление. Обратите внимание, что действие сил больше (более толстые линии), когда вершины ближе. Для ясности здесь показаны не все силы, действующие на все пары вершин, а лишь несколько примеров

Следующее, что необходимо, — это формализовать перечисленные критерии и свести их в формулу функции стоимости. Она будет описывать ландшафт задачи, который затем можно попытаться исследовать, используя градиентный спуск или алгоритмы из других категорий, которые будут рассмотрены далее в книге:

$$J(V) = \delta \sum_{v \in V} \sum_{u \in V/\{v\}} \|u - v\|_2^2 - \beta \sum_{v \in V} \sum_{u \in \text{adj}/\{v\}} \|u - v\|_2^2,$$

где член внутри суммы — это квадрат 2-нормы, который при вычислении разности (векторов) двух точек дает квадрат расстояния между двумя точками.

$$\|u - v\|_2^2 = (u_x - v_x)^2 + (u_y - v_y)^2.$$

Квадрат расстояния используется просто потому, что его вычисление обходится дешевле[1], чем вычисление производной квадратного корня (последнее довольно дорогостоящее). И конечно же, форма поверхности функции тоже будет другой.

Имеем большое улучшение по сравнению со ступенчатой функцией. По крайней мере, эта функция дифференцируема, что очень хорошо. Частные производные по координатам x и y вершины w можно вычислить точно:

$$\frac{\partial}{\partial w_x} J(V) = -4\delta \sum_{u \in V/\{w\}} (u_x - w_x) + 4\beta \sum_{u \in \text{adj}/\{w\}} (u_x - w_x);$$

$$\frac{\partial}{\partial w_y} J(V) = -4\delta \sum_{u \in V/\{w\}} (u_y - w_y) + 4\beta \sum_{u \in \text{adj}/\{w\}} (u_y - w_y).$$

[1] Извлечение квадратного корня, как известно, дорогостоящая операция.

Скаляры β и δ — это так называемые гиперпараметры алгоритма. Они уравновешивают важность сил притяжения и отталкивания. Их значения нужно скорректировать, чтобы получить желаемый результат; это можно сделать вручную или автоматически.

Конечно, произвести такую коррекцию не всегда легко. Например, большое значение параметра силы притяжения хорошо подходит для разреженных графов, так как не позволяет вершинам расходиться слишком далеко, но для плотного графа при $\beta > \delta$ все вершины будут стянуты к его центру.

Возможная альтернатива — определение идеальной длины ребра (или идеального диапазона длины) на основе вершин/ребер в графе и размера холста, на котором строится визуальное представление графа. При таком подходе оптимизация уйдет от решений, в которых все вершины группируются слишком близко:

$$\bar{J}(V) = \delta \sum_{v \in V} \sum_{u \in V / \{v\}} \|u - v\|_2^2 - \delta \sum_{v \in V} \sum_{u \in \text{adj}(v)} \left[\|u - v\|_2 - (\text{идеальная длина ребра}) \right]^2.$$

Ни одна из этих функций стоимости не направлена на прямое уменьшение количества пересечений, но, как нетрудно представить, наличие более коротких ребер и размещение смежных вершин вблизи друг от друга косвенно могут помочь уменьшить это число. И ни одна из функций не идеальна, потому что они не чашеобразные и имеют несколько локальных минимумов. Исправить этот недостаток нелегко, но у нас есть обходное решение — использовать алгоритм случайного перезапуска, произвольно выбирающий начальные позиции для вершин и перемещающийся по функции стоимости вниз с помощью градиентного спуска.

16.5.2. Реализация

Настойчивость — ключ к победе, поэтому если повторить шаг градиентного спуска несколько раз (или, может быть, много раз, что в действительности зависит от контекста!), начиная с разных позиций и даже с разными скоростями обучения, то окончательный результат может оказаться очень даже неплохим.

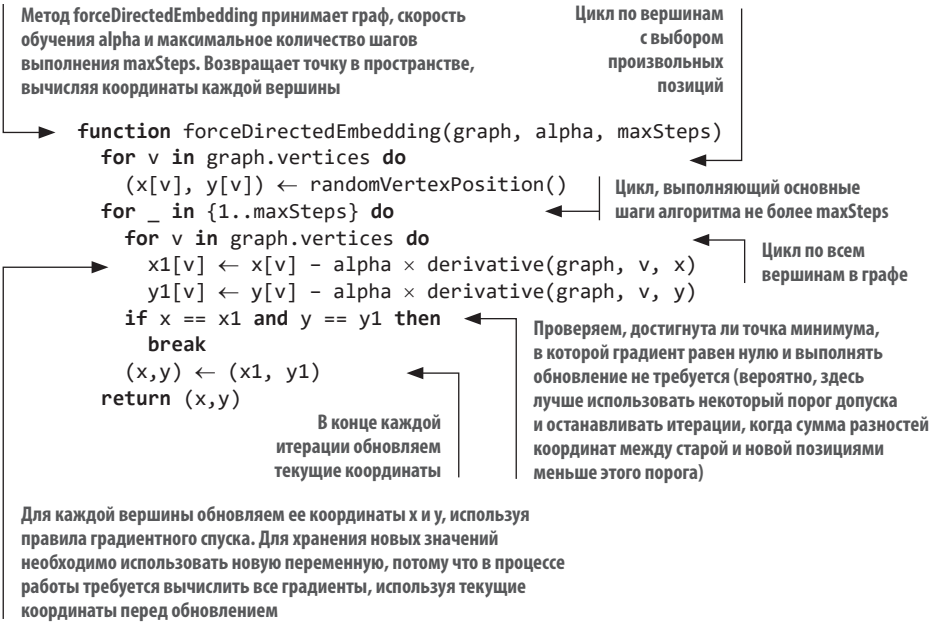
В следующей главе вы познакомитесь с более сложной методикой, которая сделает описанную оптимизацию более гибкой и повысит шансы на получение хорошего результата.

А пока реализуем решение градиентного спуска с одной итерацией, начав с листинга 16.5.

Код в листинге 16.5 повторяет тело обобщенной реализации градиентного спуска, представленной в листинге 16.4. Обобщенную реализацию тоже можно использовать, но в данном случае мы имеем более конкретную ситуацию, позволяющую несколько оптимизировать код и в целом, я полагаю, более четко выразить работу алгоритма.

Например, здесь не используется функция стоимости, вычисляются только ее частные производные. (Теперь должно стать понятнее, почему, как уже упоминалось, на роль функции стоимости выбран квадрат расстояния, чтобы избежать квадратных корней.)

Листинг 16.5. Метод forceDirectedEmbedding



Теперь о градиенте. Частные производные легко вычислить, применяя представленную формулу (или аналогичную, если используется другая функция стоимости). Требуется только два цикла `for` для обхода вершин, поэтому явный код здесь не показан.

Этот метод легко использовать в алгоритме случайного перезапуска: достаточно определить количество попыток и вызывать метод `forceDirectedEmbedding` в цикле. Но будьте внимательны: в этом случае нужно явно определить функцию стоимости, потому что (как показано в листинге 16.6) после каждого вызова `forceDirectedEmbedding` необходимо проверять стоимость полученного решения и сравнивать ее с лучшим результатом, найденным к этому моменту.

Листинг 16.6. Метод forceDirectedEmbeddingWithRestart

```
function forceDirectedEmbeddingWithRestart(graph, alpha, runs, maxSteps)
  bestCost ← inf
  for _ in {1..runs} do
    (x,y) ← forceDirectedEmbedding(graph, alpha, maxSteps)
    if cost(graph, x, y) < bestCost then
      (bestX, bestY, bestCost) ← (x, y, cost(graph, x, y))
  return (bestX,bestY)
```

На этом завершим обсуждение градиентного спуска. В следующих главах рассмотрим альтернативные алгоритмы оптимизации решений, основанных на стоимости.

РЕЗЮМЕ

- Многие задачи, включая задачи машинного обучения, основаны на определении правильной функции стоимости, которая оценивает качество решения, и на выполнении алгоритма оптимизации, пытающегося найти решение с минимальной стоимостью.
- Градиентный спуск основан на геометрической интерпретации функций стоимости. Для дифференцируемых функций предполагается, что каждое решение задачи можно интерпретировать как точку в n-мерном пространстве, а один шаг градиентного спуска выполняется путем вычисления градиента функции стоимости в текущей точке.
- Точки, в которых функция стоимости принимает локально оптимальное значение, являются врагами алгоритмов оптимизации в целом и градиентного спуска в частности. Алгоритмы могут застрять в локальных минимумах, что не позволит найти решение, оптимальное в глобальном смысле.
- Число пересечений плохо подходит на роль функции стоимости, потому что дает ступенчатую функцию с большим количеством плато локальных минимумов.
- В качестве альтернативы можно преобразовать задачу в так называемую силовую имитацию представления графа, главная цель которой — получить красивое изображение графа, при этом оптимизация числа пересечений происходит лишь попутно.

17 *Имитация отжига: оптимизация за пределами локальных минимумов*

В этой главе

- ✓ Введение в имитацию отжига.
- ✓ Использование имитации отжига для улучшения планирования сроков доставки.
- ✓ Пример решения задачи о коммивояжере.
- ✓ Использование имитации отжига для поиска представления графа с минимальным числом пересечений.
- ✓ Использование алгоритма на основе имитации отжига для построения красивых изображений графов.

Если вы прочитали главы 15 и 16, то уже должны быть знакомы с представлениями графов и задачами оптимизации. В частности, в предыдущей главе демонстрировалось, как переформулировать задачу оптимизации представления графов, и был представлен градиентный спуск — метод оптимизации, который можно использовать для поиска (почти) оптимальных решений подобных задач. В частности, мы с вами обсудили два решения, рассматривая их как оптимизацию числа пересечений и как представления графа в виде системы сил; градиентный спуск хорошо подходит для последнего случая, но плохо для первого.

Одна из проблем, с которыми мы столкнулись в процессе знакомства с градиентным спуском, заключается в его тенденции застревать в локальных минимумах, а это

крайне нежелательно, учитывая, что часто приходится иметь дело с функциями стоимости, имеющими множество локальных пиков.

Было предпринято также обсуждение одного обходного решения этой проблемы: многократного запуска градиентного спуска с выбором каждый раз другой начальной точки.

И тем не менее даже при использовании этого приема каждая итерация градиентного спуска практически обречена (в лучшем случае) заканчиваться в ближайшем локальном минимуме, находящемся в направлении самого крутого уклона от начальной точки. Вот было бы здорово, если бы даже одиночный прогон мог иметь некоторую ненулевую[1] вероятность выбраться из локального минимума и найти лучшее решение!

В этой главе обсудим алгоритм, реализующий именно это и многое другое. Он также снимает некоторые ограничения, сужающие применимость градиентного спуска. Прочитав эту главу, вы узнаете об *имитации отжига*, мощном методе оптимизации, поймете, как его применять для решения некоторых сложных задач с графами и как взвесить все за и против, выбирая именно его из множества других методов оптимизации.

Имитация отжига — довольно мощный алгоритм: среди описанных в этой книге он единственный, который способен сбалансировать широту поиска, исследовать большую часть пространства задачи и добиться локального прогресса.

Более того, он известен с 1970-х годов, и это объясняет, почему имитация отжига считается одной из лучших эвристик, используемых в оптимизации. В течение последних 20–30 лет были разработаны новые методы оптимизации, такие как *генетические алгоритмы* (которые обсудим в следующей главе) и *искусственные иммунные системы* (ИИС), использующие различные биологические подходы для ускорения сходимости. Но имитация отжига продолжает широко применяться на практике, и относительно недавно этот подход был воплощен в форме *квантового отжига*.

И последнее, но не менее важное: читая эту главу, вы улучшите свои способности планировать доставку по нескольким направлениям, познакомившись с задачей о коммивояжере и с тем, как находить субоптимальные решения в разумные сроки. Продолжая разговор об оптимизации логистики в воображаемой интернет-компании, мы перейдем от планирования доставки одного заказа к оптимизации маршрута проезда грузовика, доставляющего заказы в несколько городов, чтобы транспортное средство можно было загрузить на складе компании один раз и доставить сразу несколько заказов без возврата за товаром.

---

[1] Технически даже градиентный спуск может проскочить локальный минимум, в зависимости от формы функции вокруг него и скорости обучения, как это показано на рис. 16.16.

17.1. ИМИТАЦИЯ ОТЖИГА

В главе 16 были представлены методы локальной оптимизации как способ улучшить алгоритмы простой случайной выборки. Мы подробно обсудили градиентный спуск и упомянули метод рандомизированной оптимизации. Если вспомнить аналогию с шариками в чаше, то градиентный спуск всегда будет перемещать шарик (наше решение) по траектории с самым крутым уклоном в ландшафте, созданном функцией стоимости, но останавливаться внизу, в долинах. На рис. 17.1 приводится сравнение метода градиентного спуска с локальной оптимизацией со случайным поиском, называемой *спуском с холма*[1]. В последнем случае алгоритм выбирает направление случайным образом и проверяет, будет ли достигнуто улучшение, если сделать короткий (возможно, случайный) шаг в этом направлении. Если да, то он переходит к новому решению; в противном случае никаких изменений не производится и на следующей итерации выполняется еще одна попытка. В главе 16 также было показано, что градиентный спуск теоретически может преодолеть локальные минимумы, если скорость обучения достаточно велика; однако это маловероятно.

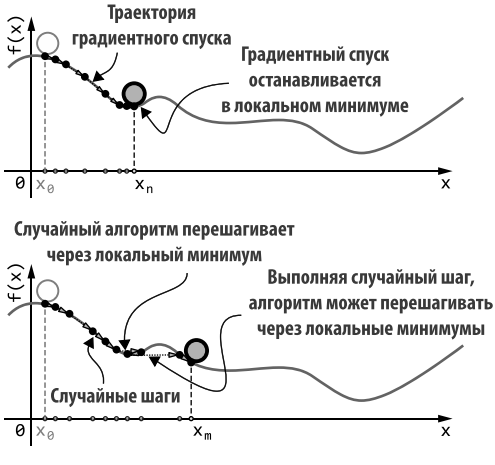


Рис. 17.1. Сравнение градиентного спуска и локальной случайной оптимизации. Градиентный спуск выбирает шаги, пропорциональные крутизне кривой, а случайная локальная оптимизация делает этот выбор... случайно (в разумном радиусе). Вот почему случайному алгоритму проще перешагнуть через локальный минимум или даже выйти за пределы его воронки (технически оба варианта возможны и для градиентного спуска при определенных скорости обучения и формы функции стоимости, как и было показано в главе 16)

Но обратите внимание, что на рис. 17.1 показаны не все неудачные попытки, когда случайная локальная оптимизация пыталась пойти в неправильном направлении. В целом, чтобы добраться до одной и той же точки, ей потребуется больше шагов, чем градиентному спуску, потому что вместо движения в направлении максимального изменения она блуждает беспорядочно. Это становится особенно очевидно, если взглянуть на рис. 17.2 — трехмерную поверхность двумерной задачи.

Как бы то ни было, указанному подходу свойственны некоторые из тех же проблем градиентного спуска, что обсуждались в главе 16, в их числе вероятность

[1] Первоначальное название этой методики — восхождение на холм, потому что целью оптимизации является максимизация; однако, поскольку мы стремимся не увеличить, а уменьшить стоимость, такое название может сбивать с толку.

застревания в локальных минимумах и на плато. И, что еще хуже, движение туда идет медленнее.

С другой стороны, случайная локальная оптимизация имеет некоторые преимущества. Первое — отсутствие ограничения на дифференцируемость функции стоимости и фактическая возможность вообще игнорировать определение функции, если, конечно, есть способ ее вычисления[1]. Это особенно полезно для таких функций, как пересечение ребер в представлении, — ступенчатых, не дифференцируемых в точках скачкообразного изменения значения и имеющих производную, тождественно равную нулю в других местах.

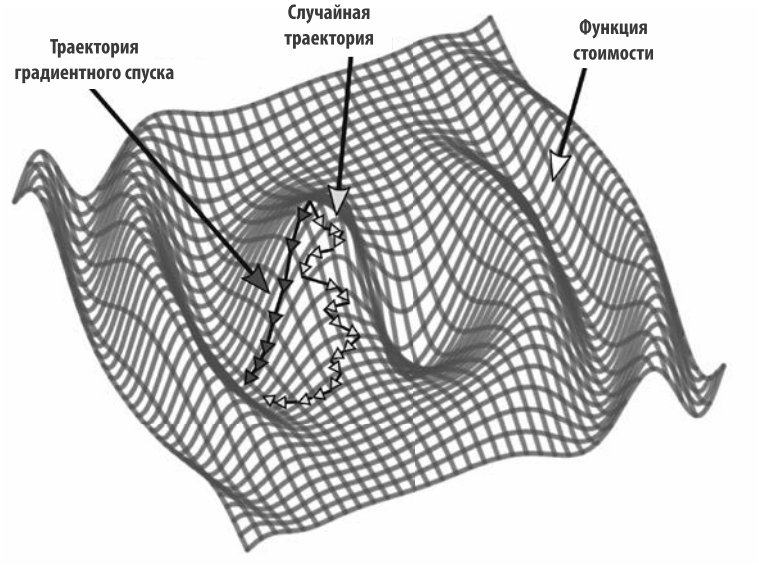


Рис. 17.2. Градиентный спуск в сравнении со случайной локальной оптимизацией для функции стоимости двух параметров. Нетрудно догадаться, какой из этих подходов сделает больше шагов

Очевидным недостатком случайной локальной оптимизации, как показано на рис. 17.2, является гораздо большее количество выполняемых шагов, потому что она часто выбирает неоптимальное направление. Хуже того, поскольку она чаще движется не в направлении наибольшего убывания функции, то, скорее всего, окажется в другом месте, чем градиентный спуск. Невозможно предсказать, где найдет решение случайная локальная оптимизация и будет ли оно лучше или хуже. Единственная стратегия для преодоления этого недостатка — настойчивость: увеличение количества прогонов и сохранение лучшего результата (чем больше прогонов, тем лучше ожидаемое решение).

[1] Отсутствие статической формулы позволяет применять динамические функции стоимости, например зависящие от внешних вычисляемых метрик или моделей (скажем, при обучении с подкреплением стоимость определяется путем запуска моделирования).

Наконец, градиентному спуску свойственны проблемы, от которых не может избавиться даже рандомизированный метод.

Например, алгоритм случайной оптимизации может выходить за пределы локального минимума, но такое его поведение не гарантируется. Оба алгоритма являются жадными[1], поэтому они перемещаются из текущей позиции, только когда находят точку в пространстве задачи, которой соответствует более низкая стоимость. Диапазон случайных шагов может оказаться недостаточно широким, чтобы выбраться из ямы. Это видно на рис. 17.1, где расстояние, которое необходимо преодолеть, чтобы выбраться из локального минимума, слишком велико по сравнению с размером шага. Но даже если такие большие шаги будут разрешены (в зависимости от особенностей настройки алгоритма через его гиперпараметры), то, скорее всего, потребуется много случайных попыток, прежде чем метод сгенерирует шаг одновременно в правильном направлении и нужного размера.

17.1.1. Иногда нужно подняться, чтобы спуститься

На рис. 17.3 показан пример еще более сложной конфигурации для алгоритма локальной оптимизации. Расстояние между текущим минимумом, где застрял рандомизированный алгоритм, и следующей точкой на ландшафте функции стоимости с более низким значением (ближайшее улучшение) слишком велико, чтобы добраться туда за один шаг. Нужно, чтобы алгоритм мог сказать: «Хорошо, неважно, что я нахожусь в минимуме; я перелезу через этот холм и посмотрю, не глубже ли следующая долина (или следующая за ней)».

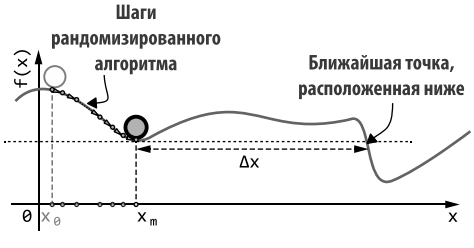


Рис. 17.3. Ограничение, которое присуще и алгоритму случайной локальной оптимизации, и градиентному спуску, — он движется только вниз. Если следующая позиция с еще лучшим значением функции стоимости находится слишком далеко, то алгоритм не сможет достичь ее за один шаг и застрянет

Сделать это можно множеством способов. Например, можно сохранить лучшее найденное решение и продолжать исследовать его окрестности за пределами локального минимума. Другой вариант — допустить, что иногда шаг в гору — это нормально. Его можно планировать систематически, через каждые несколько шагов, или вероятностно, допуская шаг в гору с определенной вероятностью.

В имитации отжига используется последний подход; в своей первоначальной формулировке он не предполагает сохранение лучшего найденного решения, но это легко реализовать.

---

[1] Жадные алгоритмы делают локально оптимальный выбор (например, перемещаются только к позициям с более низкой стоимостью). К сожалению, это не всегда приводит к достижению оптимального результата.

Эта эвристика[1] получила свое название от метода отжига, используемого в металлургии. Там он заключается в повторении циклов, когда металл нагревается, а затем контролируемо охлаждается для повышения его прочности: что-то вроде кузнеца, выковывающего железный меч путем закалки в холодной воде, но более контролируемым образом (и без воды!).

Имитация отжига действует очень похоже. В своей простейшей форме она состоит из одной фазы охлаждения, но существуют варианты чередования фаз нагрева и охлаждения.

Температура системы, в свою очередь, напрямую связана с энергией моделируемой системы и вероятностью перехода в более высокоэнергетическое состояние (иными словами, перехода к решению, когда функция стоимости принимает более высокие значения).

На рис. 17.4 показана возможная траектория на трехмерной поверхности, напоминающая процесс оптимизации методом имитации отжига. На этот раз алгоритм способен выйти из локального минимума, даже совершив противоречивый шаг в гору: в этом его отличие от жадных алгоритмов (в том числе описанных в разделе 17.1) и истинное преимущество этой методики.

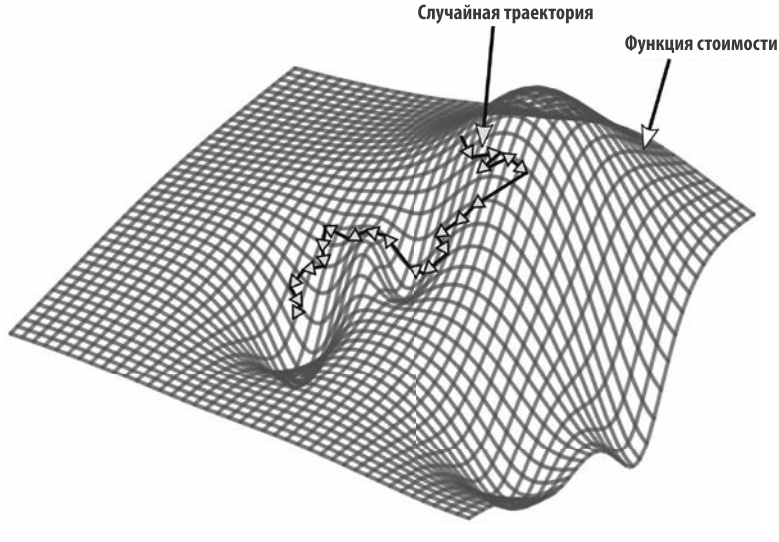


Рис. 17.4. Подъемы и спуски возможной траектории алгоритма оптимизации методом имитации отжига. Будьте внимательны, это искусственный пример. Весьма вероятно, что в реальных прогонах на начальных этапах алгоритм будет пробовать разные направления, а потом, при понижении температуры, сходиться к одному из минимумов

[1] Имитацию отжига вполне можно считать категорией эвристик, также известной как метаэвристика, когда каждый алгоритм, использующий имитацию отжига для решения конкретной задачи, считается эвристикой.

Траектория, показанная на рис. 17.4, хоть и возможна[1], но не типична для имитации отжига. Скорее, типична гораздо более хаотичная траектория, но ее использование для иллюстрации сделало бы рисунок слишком запутанным.

В реальных приложениях алгоритм, скорее всего, сначала будет прыгать туда-сюда, исследуя несколько областей ландшафта и часто двигаясь в гору. Но по мере остывания вероятность перехода в более высокие позиции будет снижаться, и одновременно с этим можно прямо или косвенно уменьшать длину случайных шагов. В общем случае после первоначального этапа исследования алгоритм должен перейти к этапу тонкой настройки.

Пример всего процесса показан на рис. 17.5. Как видите, я не преувеличивал, говоря о хаотичности!

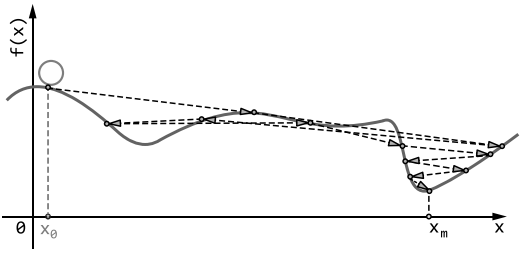


Рис. 17.5. Более реалистичный пример возможной траектории имитации отжига на пути к минимуму функции одного параметра. Изначально алгоритм делает большие шаги и вероятность получить худший результат выше. Но по мере охлаждения начинается точная настройка размера шагов (в основном) для достижения лучшего результата

17.1.2. Реализация

Но хватит говорить о том, насколько крут этот алгоритм, посмотрим на его псевдокод. Он представлен в листинге 17.1, а его реализация на JavaScript в библиотеке JsGraphs доступна в репозитории книги на GitHub[2].

Алгоритм выглядит на удивление просто, верно? И достаточно лаконично, даже притом что, как всегда, я постарался представить обобщенные методы в виде шаблонов и по возможности абстрагировать как можно больше подпрограмм в форме вспомогательных методов, которые потом можно будет реализовать с учетом контекста.

В данном случае их три; в следующем разделе мы обсудим функцию, вычисляющую изменение температуры (`temperature`) и функцию, возвращающую вероятность выбора в пользу перехода (`acceptance`). А здесь предстоит проанализировать функцию `randomStep`, позволяющую построить следующее предварительное решение, к которому может перейти алгоритм.

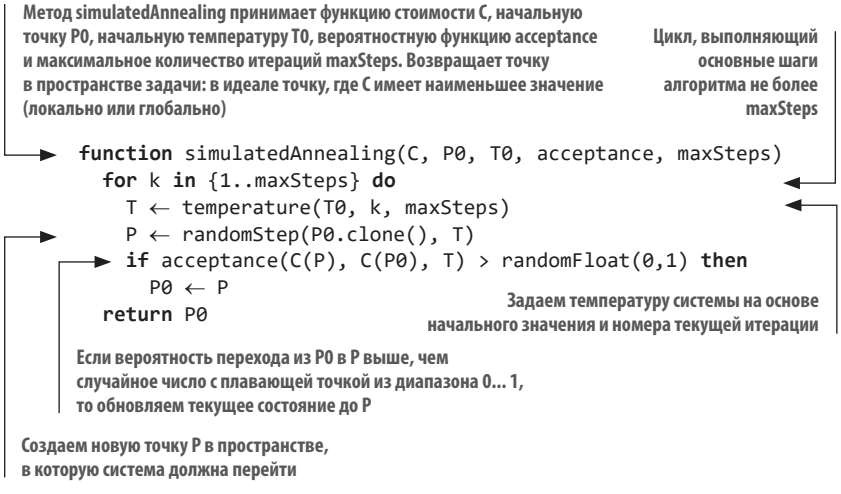
Очевидно, что эта функция должна зависеть от особенностей предметной области: размер пространства задач и тип решений будут определять, как можно изменить

[1] При правильных настройках и, как мы увидим далее, с функцией перехода малого радиуса действия.

[2] https://github.com/mlarocca/AlgorithmsAndDataStructuresInAction#simulated-annealing.

текущее решение (точку в пространстве задачи). Например, в задаче поиска представления графа можно случайным образом перемещать каждую вершину по обеим осям в пределах максимальной площади, в которую должен уместиться граф.

Листинг 17.1. Обобщенная реализация алгоритма имитации отжига



Но, как можно видеть в строке 4, в эту функцию добавлена зависимость от температуры!

В предыдущем разделе упоминалось, что можно прямо или косвенно регулировать длину случайного шага. Проще говоря, желательно проявлять некоторую осторожность, изменяя ее напрямую, но это лишь один из вариантов. Чтобы понять причины, необходимо сначала подробно объяснить две другие функции и понять, почему имитация отжига работает.

17.1.3. Почему имитация отжига работает

Может, у вас складывается впечатление, что работа имитации отжига несколько противоречит здравому смыслу? Знайте, что вы не одиноки. В конце концов, если алгоритм может делать большие шаги и допускает возможность перемещения в худшие позиции, не используя никаких знаний о функции стоимости (как предварительных, так и полученных в процессе работы), то на чем основывается уверенность, что он окажется в области глобального минимума и не застрянет в воронке какого-нибудь локального минимума, когда мы начнем снижать температуру? Да, такой уверенности нет. Но в большинстве ландшафтов, если найти правильную конфигурацию и если позволить алгоритму работать достаточно долго, он сможет приблизиться к глобальному оптимуму и преодолеет градиентный спуск.

В последнем предложении много «если». Однако, как и большинство эвристических алгоритмов, этот хорошо работает на практике, когда находится в руках специалиста,

который знает, как правильно его настроить, и имеет достаточно вычислительного времени, чтобы заставить его работать в течение многих итераций.

В таких случаях имитация отжига — отличная альтернатива для сценариев, где градиентный спуск оказывается бессильным. Дело не в том, что одно лучше другого; как всегда, есть задачи, с которыми градиентный спуск справляется лучше, но есть задачи, где его трудно или невозможно применить.

Разберемся подробно, почему работает имитация отжига.

Ключевой особенностью алгоритма является вероятностный механизм, позволяющий двигаться в гору (строки 2–5), к худшим значениям функции стоимости и, в частности, функции, которая возвращает вероятность выбора в пользу положительного значения дельты в зависимости от температуры и величины дельты.

Если предположить, что в данной итерации, когда температура имеет значение T, алгоритм пытается перейти из текущей точки P_0 в точку P, то эту функцию вероятности $A(P_0, P, T)$ можно выразить как:

$$A(P, P_0, T) = \begin{cases} -\dfrac{C(P_0) - C(P)}{kT}; \\ e, \text{ если } C(P) \geq C(P_0); \\ 1 \text{ в ином случае,} \end{cases}$$

где $C(P)$ — стоимость решения P (аналогично для P_0), а k — константа, которую нужно откалибровать по начальной температуре и максимальному значению дельты функции стоимости, чтобы на начальном этапе алгоритма (когда температура системы близка к начальной температуре) вероятность принятия дельты положительного значения была близка к 1 для любых двух состояний.

Обычно вероятность перехода в состояние с меньшей энергией устанавливается равной 1, так что такой переход всегда разрешен на любом этапе моделирования.

Конечно, это не единственный способ определения функции вероятности выбора в пользу положительного значения дельты, потому что он вытекает непосредственно из металлургической аналогии: эта формула основана на распределении Больцмана, которое измеряет вероятность нахождения системы в состоянии с определенной энергией и температурой[1]. Но вместо абсолютного значения энергии в имитации отжига рассматривается переход из более низкого в более высокое энергетическое состояние.

Как это распределение вероятностей влияет на один шаг алгоритма? Рассмотрим случай, когда ширина шага обновления в предметном пространстве не ограничена (поэтому алгоритм может даже переместиться из одного угла области в противоположный). Взглянем на рис. 17.6.

---

[1] Соответственно, константа k называется постоянной Больцмана.

Изначально, когда температура системы высока (рис. 17.6, *А*) и распределение вероятностей должно допускать практически любое обновление (каким бы плохим оно ни казалось), алгоритм может перемещаться по всему ландшафту и выбираться из локальных минимумов, даже если это означает, что он может отказаться от оптимального положения ради худшего. Фактически на этом этапе алгоритм легко может уйти от хороших решений.

Это поведение алгоритма имитирует высокоэнергетические системы, в которых частицы (или молекулы) хаотично движутся во всех направлениях.

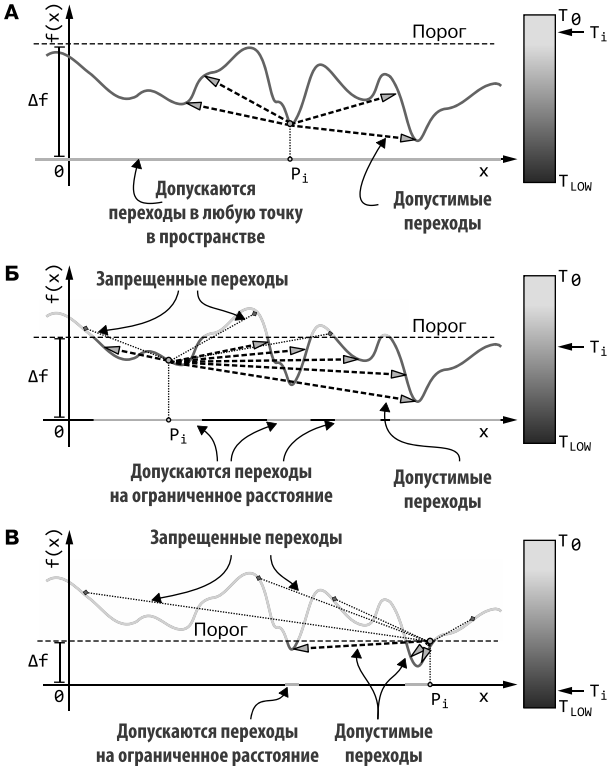


Рис. 17.6. Влияние температуры на вероятность перехода в более высокоэнергетическое состояние и, в свою очередь, на алгоритм имитации отжига. *А*. Первоначально температура достаточно высока, чтобы был возможен любой переход. На этом этапе алгоритм ведет себя как алгоритм случайной выборки (без сохранения лучшего результата) и исследует большой участок предметной области. *Б*. Когда система остывает, переходы в более высокие состояния становятся менее вероятными, а состояния выше некоторого порога становятся настолько маловероятными, что их можно считать запрещенными. Переходы в более низкие состояния разрешены всегда. *В*. Ближе к концу моделирования вероятность перехода в более высокоэнергетические состояния настолько мала, что ею можно пренебречь. Система может двигаться только к лучшим решениям

По мере остывания системы распределение меняется (см. рис. 17.7, *Б*). Подъем в гору становится менее вероятным, а некоторые позиции выше определенного значения дельты (по их стоимости) вообще становятся недостижимыми.

Наконец, когда температура приближается к T_{LOW}, температуре остановки (или, что то же самое, при приближении к максимальному количеству итераций), движение вверх становится настолько маловероятным, что оно фактически запрещено, и алгоритм может переходить только во все более низкоэнергетические состояния. Он ведет себя как *спуск с холма*[1]. Однако на рис. 17.6, *В* можно заметить, что разница между этими алгоритмами есть и переходы к точкам, расположенным далеко в пространстве задачи, все еще разрешены, если они имеют более низкую стоимость. Это означает, что алгоритм все еще может выйти из локальных минимумов (и есть вероятность, что так и будет!) и оказаться в локальном оптимуме.

Обратите внимание, что допустимость или недопустимость перехода в новое состояние не связаны с расстоянием до нового состояния в пространстве задачи и зависят только от уровней энергии в двух состояниях, которые, в свою очередь, задаются (или по крайней мере пропорциональны) значению функции стоимости в этих состояниях.

Что еще более важно, после самых начальных стадий, когда разрешены практически любые переходы, чем выше оказывается новое состояние, тем меньше вероятность, что переход будет разрешен. Вот почему алгоритм работает хорошо — он последовательно поощряет переходы в области с более низкой стоимостью (см. рис. 17.6, *Б*, *В*), не ограничивая поиск в окрестностях текущей позиции.

Почему работают переходы в гору? Потому что, поднимаясь в гору, алгоритм может преодолеть неглубокий овраг и достичь более глубокой долины. На двумерной диаграмме это может выглядеть нелогичным, но становится более актуальным в многомерном пространстве.

Однако, как и в металлургическом процессе, важную роль в качестве конечного результата играет правильное охлаждение системы.

И конечно же, поскольку это чисто стохастический процесс, ему тоже нужна «удача»; особенно ближе к концу, когда многие случайные попытки пойти в гору будут отклоняться. Но если выделить достаточно времени, то есть вероятность, что за множество итераций положительное изменение будет найдено случайным образом.

В этом заключается сила алгоритма, и в этом же его слабость. Он может найти лучшую позицию, но, поскольку постепенно отвергает все больше и больше попыток обновления, ему потребуется несколько итераций (технически: много!) и, конечно же, необходима хорошая функция, генерирующая случайные шаги для исследования пространства задачи.

Это делает алгоритм особенно медленным и ресурсоемким, особенно если итерации (генерация случайной точки и оценка функции стоимости) требуют больших затрат.

---

[1] Спуск с холма описан в главе 16.

17.1.4. Переходы на короткие и большие расстояния

Теперь снова возникает вопрос: насколько постепенно ограничивать дальность переходов и как далеко алгоритм сможет двигаться на каждом шаге в пространстве предметной области?

Чтобы длина максимального шага обновления сокращалась с течением времени, ее можно поставить в зависимость от температуры *T*.

Но самое интересное заключается в том, что даже если сохранить неизменной максимальную длину шага на протяжении всего процесса, то при снижении температуры все равно будет происходить некое косвенное ее уменьшение. Как показано на рис. 17.6, фильтрация на самом деле происходит в области функции стоимости, но косвенным эффектом является ограничение достижимой области подобластью исходного пространства задачи. И это тот же эффект, что действует на поток воды, стекающей по воронке, — она склонна двигаться в определенном направлении, и скорость потока замедляется (по мере роста давления).

Если дополнительно ограничить допустимое расстояние переходов вдоль пространства задачи, то, с одной стороны, получим большее количество попыток обновления в области, окружающей текущую точку, что может быть хорошо, если алгоритм близок к глобальному минимуму, потому что это ускорит сходимость. С другой стороны, потеряется способность выходить из локальных минимумов, которая, в свою очередь, является главной причиной использования имитации отжига.

Поэтому одним из «безопасных» решений является реализация метода `randomStep` как чистой случайной выборки. Однако его использование означает, что потребуется много итераций, чтобы найти переход вниз, и алгоритм будет продолжать прыгать между долинами, не сосредоточиваясь на тонкой настройке. Интересным компромиссом могло бы стать увеличение вероятности небольших шагов с сохранением возможности выполнять большие шаги и даже принудительно предпринимать такие попытки каждые несколько итераций. Это можно сочетать с сокращением диапазона локального поиска (маленькими шагами) по мере снижения температуры.

Последнее, что нужно обсудить, — порядок изменения температуры. Как и в случае с вероятностью разрешения выбора в пользу положительного значения дельты, для функции существует несколько возможных вариантов — для нее нет никаких ограничений.

Однако обычно используется геометрический (он же экспоненциальный) спад, когда значение температуры обновляется не каждую итерацию, а через некоторый интервал (например, каждые 1000 итераций или около того).

Вот как выглядит математическая формула для этой функции:

$$T_i = \alpha T_{i-1}, 0 < \alpha < 1.$$

Температура в итерации *i* составляет часть температуры в итерации *i* – 1; коэффициент α должен иметь значение в диапазоне между 0 и 1, он определяет, насколько снижается температура между двумя итерациями.

Может понадобиться поэкспериментировать со значением α и интервалом обновления температуры, чтобы настроить их и получить наилучшие результаты. Как правило, экспоненциальный спад быстро происходит в начале и медленно — начиная с середины и до конца. На рис. 17.7 показано несколько примеров, как величина замедления зависит от выбора α.

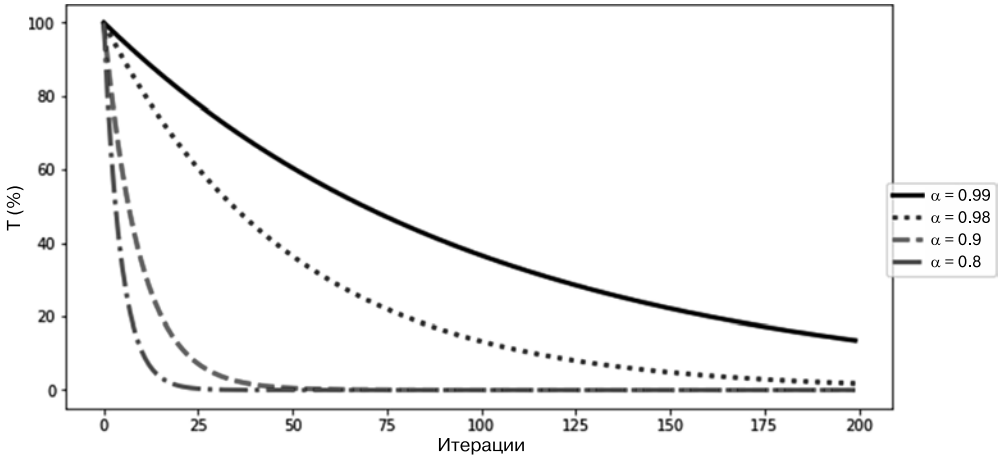


Рис. 17.7. Чтобы понять, как параметр α управляет охлаждением в алгоритме имитации отжига, посмотрим на экспоненциальную функцию затухания: $f(i) = T_i = \alpha \times f(i-1), \forall\, i > 0$. На этой диаграмме $f(0) = T_0 = 100$, и показано, как форма функции изменяется со скоростью α, которая определяет скорость стремления функции к нулю. Вы спросите: почему это называется экспоненциальным спадом? Так как $T_1 = \alpha T_0$, $T_2 = \alpha T_1 = \alpha_2 T_0$ и в общем случае $T_n = \alpha_n T_0$. Если α находится в диапазоне от 0 до 1, то α_n и, в свою очередь, T_n постепенно (экспоненциально) уменьшаются

На практике обычно $\alpha \approx 0{,}98$ считается безопасным выбором для начала (потом его можно постепенно настраивать).

17.1.5. Варианты

Как было показано, метод имитации отжига, несмотря на то что он бесспорно является мощным инструментом, также имеет некоторые недостатки, и наиболее важный из них — низкая скорость сходимости из-за стохастической природы алгоритма.

Итак, как и в случае других алгоритмов, со времени его открытия было изучено множество вариантов имитации отжига с целью исправить недостатки метода и сделать его еще лучше.

Выше уже упоминалась тривиальная модификация: сохранение лучшего решения, найденного во всех итерациях. На самом деле, особенно в больших, многомерных пространствах задач, может случиться так, что в начальной высокоэнергетической фазе алгоритм случайно окажется рядом с глобальным минимумом, а затем поднимется в гору и больше не сможет вернуться к такому хорошему результату. Вероятность такого события зависит от многих факторов, например от формы ландшафта (функция стоимости) и выбора правильной конфигурации параметров. В любом случае запоминание лучшего результата может быть легкой победой.

В развитие этого соображения алгоритм *имитации отжига с перезапусками* мог бы сохранить один или несколько лучших результатов, полученных в ходе итераций, и, застряв на каком-то этапе, переместиться в одну из этих предыдущих (и выгодных) позиций и продолжить спуск, начиная с нее, или просто переместиться в другую область.

Выше отмечалось, что вероятность застрять в локальных минимумах у этого алгоритма ниже, чем у жадных алгоритмов, но все же она не равна нулю.

В частности, застревание может быть вызвано сразу несколькими факторами, которые приведены ниже.

- Неоптимальный выбор параметров алгоритма. Охлаждение происходит, например, слишком быстро, и алгоритм застревает вдали от глобального минимума.
- Невезение. Как уже говорилось, это стохастический алгоритм, и он может достичь оптимума слишком рано и затем не сумеет вернуться назад, когда система остынет.
- Шаг обновления (наиболее вероятный фактор). Мы видели, что случайная выборка вызывает наименьшее количество ограничений в отношении других областей пространства задачи с более низкой энергией, но она замедляет локальную сходимость (иногда слишком сильно, так, что та становится неприемлемой).
- В одних случаях правила обновления могут допускать длинные шаги, а в других (как мы увидим далее) проще реализовать небольшие шаги, вследствие чего переходы в другие области, способные вывести алгоритм за пределы локального минимума, случаются реже.

Когда возникают некоторые или все эти факторы, использование случайного перезапуска может спасти положение.

Еще одна проблема алгоритма имитации отжига — настройка параметров может оказаться сложной и раздражающей задачей. При адаптивной имитации отжига параметры k и α настраиваются автоматически в процессе выполнения алгоритма или в зависимости от тенденции изменения уровня энергии. Последний использует механизмы, заимствованные из термодинамики, допускающие даже повышение температуры (имитация циклов охлаждения и нагревания).

17.1.6. Имитация отжига или градиентный спуск: что использовать?

Вы видели, что имитация отжига работает медленнее градиентного спуска: последний выбирает самый быстрый путь, поэтому его трудно превзойти в простых случаях.

Однако градиентный спуск требует дифференцируемой функции стоимости и легко застревает в локальных минимумах.

Но когда функция стоимости не дифференцируемая или имеет ступенчатый вид, как в задаче поиска представления графа с минимальным числом пересечений, или когда пространство задачи является дискретным (например, как в задаче о коммивояжере, которая описывается ниже в этой главе), тогда имитация отжига предпочтительнее градиентного спуска.

Для задач, имеющих гибкие требования к времени выполнения и качеству решения, имитация отжига тоже может быть предпочтительнее градиентного спуска. Имейте в виду, что имитация отжига — это *алгоритм Монте-Карло*, и поэтому (как обсуждается в приложении Е) он возвращает субоптимальное решение, качество которого улучшается с увеличением времени, которое отводится для его выполнения.

Напротив, когда имеются гарантии относительно дифференцируемости и формы функции (например, если график функции имеет чашеобразную форму, как в случае линейной/логистической регрессии и т. д.), тогда лучше использовать сверхспособности градиентного спуска.

Бывают ли случаи, когда лучше избегать имитации отжига?

Очевидно, как уже обсуждалось, если задача не допускает решений, близких к оптимальным, а требует наилучшего из возможных, то имитация отжига не лучший инструмент для ее решения.

Форма функции стоимости тоже имеет значение. Может случиться, что функция стоимости имеет узкие долины, и алгоритм рискует их не обнаружить, а в таких долинах будет находиться глобальный минимум. Тут становится маловероятно, что имитация отжига сойдется к почти оптимальному решению.

Наконец, как вы увидите в примерах далее, важным требованием является простота вычисления функции стоимости для нового решения-кандидата (возможно, это позволит напрямую вычислить значение дельты, основываясь только на разнице с текущим решением). Поскольку стоимость вычисляется в каждой итерации, то дорогостоящая в вычислительном отношении функция замедлит оптимизацию, заставляя запускать алгоритм с меньшим количеством итераций.

17.2. ИМИТАЦИЯ ОТЖИГА ПЛЮС КОММИВОЯЖЕР

Я надеюсь, что обсуждение имитации отжига показалось вам интересным. Вы познакомились с чрезвычайно полезным инструментом, и теперь пришло время применить его на практике. К счастью, для этого у нас есть идеальное приложение!

Помните нашу компанию, занимающуюся электронной коммерцией? Мы оставили ее в главе 14, которая была посвящена доставке и оптимизации маршрута в одну точку внутри города.

Как уже упоминалось, планировать доставку заказов по отдельности нерентабельно и нереально.

Это не означает, что знакомство с оптимизацией маршрутов с помощью алгоритмов Дейкстры и A* в главе 14 было бесполезным. Наоборот: тогда вы познакомились с дополнительной тонкой оптимизацией, которую всегда можно применить для доставки одного заказа, переходя от общего *i*-го пункта назначения к следующему. Поскольку маршруты придется вычислять на лету, важность использования эффективного алгоритма возрастает еще больше.

И все же, поскольку для амортизации затрат на доставку и захвата рынка нам необходимо загрузить каждый грузовик несколькими заказами (а возможно, и загрузить его максимально) и выезжать в рейсы к нескольким заказчикам, поиска оптимального пути от склада до места назначения уже недостаточно.

В этом разделе сосредоточимся на оптимизации маршрута доставки, предполагая, что загрузка автомобиля (и, в свою очередь, пункты назначения) фиксированы. После оптимизации этого этапа нас ждет еще одна оптимизация более высокого уровня: формирование пакетов заказов таким образом, чтобы минимизировать расстояния (или времени в пути, или, что более вероятно, стоимости пробега) для всех грузовиков и всех заказов. Но не будем торопиться и станем решать поставленные задачи последовательно. В этой главе пока сосредоточимся на такой: имея список городов, связанных дорогами, найти оптимальный маршрут (последовательность посещения городов) такой, что, перемещаясь из города в город за некоторую цену (за цену, например, можно принять расстояние между двумя городами), грузовик в конце концов вернется в первый город, и при этом стоимость его поездки будет минимальной.

Эту ситуацию иллюстрирует рис. 17.8: здесь показаны десять реальных и вымышленных городов, соединенных между собой дорогами, и — выделено — кратчайший маршрут, проходящий через каждый город ровно один раз.

Чтобы не усложнять пример, будем также считать, что в каждый город доставляется только один заказ, при этом все этапы имеют одинаковый масштаб. Однако ничего

не изменится, если принять, что в каждый город доставляется несколько заказов. Поскольку внутригородские расстояния меньше расстояний между городами, доставки в одном и том же городе будут естественным образом сгруппированы вместе. Задачу также можно оптимизировать на разных уровнях в два этапа: сначала найти наилучший порядок посещения городов, а затем для каждого города вычислить наилучший маршрут внутри города, используя тот же алгоритм[1], это просто низкоуровневые детали.

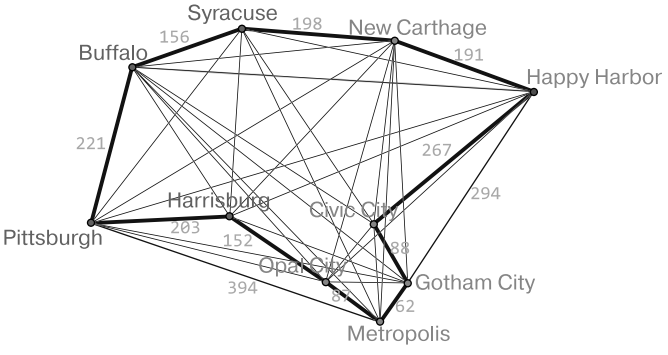


Рис. 17.8. Пример карты с реальными и вымышленными городами и расстояниями: поскольку это полный граф с почти 50 ребрами, мы показываем только несколько чисел, почти только для ребер в лучшем туре (более толстые края)

Абстрактная формулировка этой задачи совпадает с хорошо известной задачей информатики — *задачей о коммивояжере*. Как легко догадаться, эту задачу сложно решить, в частности, потому, что она является *NP-полной*, а также *NP-трудной* задачей, где *NP* в первом случае означает отсутствие известного детерминированного алгоритма, способного решить ее за полиномиальное время, а во втором — что при наличии решения-кандидата его можно проверить за полиномиальное время.

Неформально можно сказать, что любому детерминированному алгоритму, решающему задачу о коммивояжере, потребуется экспоненциальное время (хотя мы не можем быть уверенными в этом, потому что у нас нет ответа, как справиться с проблемой *P* и *NP*).

Первым следствием того, что задача о коммивояжере является NP-трудной, является невозможность предположить, что мы сможем решить экземпляр этой задачи

[1] Пришлось бы ограничить первую и последнюю доставку в каждом городе так, чтобы они были ближе всего к въезду/выезду из города. Тем не менее может случиться так, что решение, найденное в два шага, окажется не самым лучшим из возможных. Если дороги, соединяющие данный город с другими городами, начинаются в разных районах, это может повлиять на выбор наилучшей последовательности посещения городов; однако такое влияние, скорее всего, окажется несущественным и, соответственно, незначимым.

на лету, на мобильном телефоне водителя или даже на портативном компьютере (исключением может быть только случай, когда количество городов для доставки невелико; кстати, для десяти городов на рис. 17.8 имеется 10!, или ~ 3,6 миллиона, возможных последовательностей). Нужны компьютерные мощности и заблаговременное планирование и даже предварительные вычисления, чтобы как можно больше использовать результаты повторно.

На самом деле время работы точного алгоритма даже на суперкомпьютере растет слишком быстро. Для случая с 15 городами существует примерно 1,3 триллиона возможных решений, а для случая с 20 городами их число увеличивается до $2{,}4 \times 10^{18}$ (2 миллиарда миллиардов). Даже если предположить, что найден алгоритм, работающий за экспоненциальное время (асимптотически лучше, чем факториал), окажется невозможно обрабатывать случаи с доставкой заказов в более чем ~40 различных городов (на практике эта цифра будет намного меньше).

17.2.1. Точное и приближенное решения

Последнее допущение, которое будет сделано, заключается в том, что можно позволить себе пойти на компромисс в отношении оптимальности решения. Нас вполне устроит решение, близкое к оптимальному, пусть оно и не будет самым лучшим. Например, если мы добавим несколько миль к общей сумме 1000 миль за поездку, то такой «довесок» выглядит вполне приемлемым, удвоение общей длины пути было бы слишком дорогим удовольствием, а удесятерение стало бы катастрофой.

Это верно не для всех случаев, и бывают ситуации, когда нужно самое оптимальное решение, потому что даже небольшое отклонение от него приводит к большим затратам, а само оно иногда способно спасти жизни. Например, в хирургии, как нетрудно себе представить, небольшая ошибка может иметь ужасные последствия.

Но поскольку мы согласны на неоптимальные решения, это означает, что сможем использовать эвристику для получения приемлемого решения в разумные сроки.

Специально для задачи о коммивояжере было разработано несколько эвристик. Например, для графов, в которых расстояния подчиняются неравенству треугольника (как у нас), класс алгоритмов, использующих *минимальное остовное дерево* графа (см. подраздел 2.8.2) и работающих за линейное время, $O(n \times \log(n))$, может гарантировать решение, стоимость которого превышает минимальную стоимость не более чем в два раза, а в среднем[1] оно всего на 15–20 % больше.

Однако попробуем другой способ. Все названное явно проблема оптимизации, так почему бы не попытаться решить ее с помощью нашего нового сверкающего инструмента — имитации отжига?

---

[1] Как отмечалось в главе 16, NP-трудность основана на сценариях наихудшего случая; однако часто задачи сложны только для небольшого количества пограничных случаев, тогда как многие реальные задачи можно решить более эффективно.

ЯВЛЯЕТСЯ ЛИ ИСПОЛЬЗОВАНИЕ ИМИТАЦИИ ОТЖИГА ПРАВИЛЬНЫМ ВЫБОРОМ ДЛЯ МОЕГО ПРОЕКТА?

Хочу сказать несколько слов в предостережение. В этой книге мы несколько раз упоминали молоток Маслоу, и похоже, что сейчас как раз тот случай, когда нужно тщательно подумать, прежде чем выбрать, какой инструмент использовать. Как вы уже знаете, риск заключается в искушении использовать молоток (имитация отжига), когда отвертка подошла бы лучше.

Итак, прежде чем решить, стоит ли использовать имитацию отжига, нужно задать себе несколько вопросов, например: есть ли в нашей команде/компании опыт и умения для разработки и настройки этого алгоритма? Может быть, у нас есть более прочные навыки использования других решений? Какие усилия потребуется приложить для воплощения каждого из этих решений? В чем преимущество одного решения перед другим?

Имитация отжига потенциально может дать лучшее решение, чем среднее значение, обеспечиваемое эвристикой минимального остовного дерева (Minimum Spanning Tree, MST), и даже привести к глобальному оптимуму. Кроме того, предположим, что у нас нет опытных экспертов по другой конкретной проблеме внутри компании: в целом, возможно, стоит попробовать имитацию отжига, которая является высокоуровневой и потенциально может повторно использоваться для других задач оптимизации в будущем.

А вот что касается функции стоимости, здесь нам повезло. Она естественным образом вытекает из самого определения задачи: это сумма длин ребер между соседними вершинами в последовательности (в том числе между последней и первой вершинами, конечно).

17.2.2. Визуализация стоимости

Одна приятная особенность решений задачи о коммивояжере — пространство задачи представляет собой набор всех возможных перестановок вершин графа, и, поскольку каждую последовательность можно отобразить в целое число, несложно построить красивую двумерную диаграмму функции стоимости и даже посмотреть, как продвигается алгоритм. Пространство задачи, конечно, огромно, когда количество вершин велико, как это ясно видно на рис. 17.9, и необходимо увеличить масштаб небольшой части области, чтобы различить особенности ландшафта функции стоимости.

Чтобы получить более четкое представление о процессе, уменьшим количество городов. Например, оставим только шесть вымышленных городов из изображенных на рис. 17.8. В результате получится граф, показанный на рис. 17.10, для которого имеется только 720 возможных перестановок вершин. То есть ландшафт становится таким, как показано на рис. 17.11, он выглядит гораздо менее хаотичным, чем показанный на рис. 17.9.

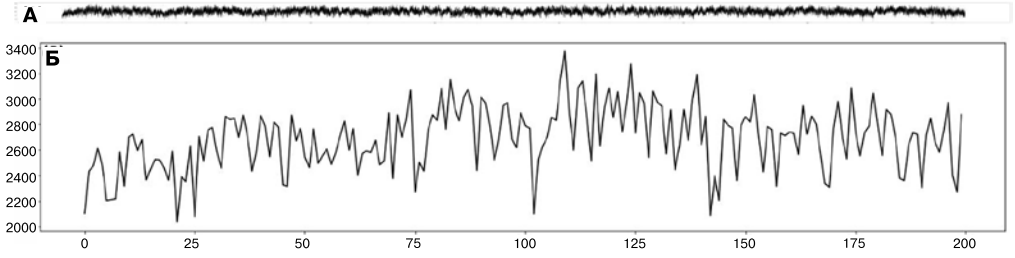


Рис. 17.9. Для задачи о коммивояжере ландшафт функции стоимости, представленной в форме графа на рис. 17.8. *А*. График функции стоимости для всего пространства задачи; в нем трудно разобраться! *Б*. Увеличив отрезок с первыми сотнями перестановок, можно увидеть несколько локальных минимумов

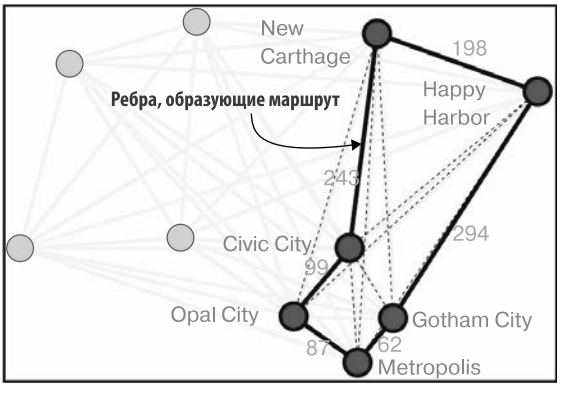


Рис. 17.10. Решение задачи о коммивояжере для подграфа графа K_{10}, показанного на рис. 17.8. Здесь имеем полный граф K_6, образованный вымышленными городами, и найденное лучшее решение. Остальные вершины/ребра исходного графа закрашены серым цветом для ясности

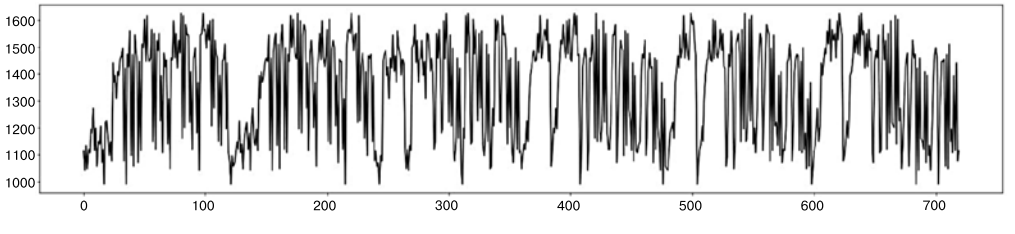


Рис. 17.11. Ландшафт функции стоимости для задачи о коммивояжере на полном графе K_6, изображенном на рис. 17.10. Помимо фактических значений, обратите также внимание на повторяющиеся закономерности

Теперь, когда у нас есть полный ландшафт, вы могли бы сказать, что лучшее решение найдется методом перебора, и были бы правы. Для рисования диаграммы на рис. 17.11 уже подготовлена стоимость каждой перестановки, а значит, можно

просто взять минимальную из них. Как показано на диаграмме, лучшим решением является последовательность:

```
[New Carthage, Happy Harbor, Gotham City, Metropolis, Opal City, Civic City]
```

Дело в том, что нет возможности использовать такое решение для более крупных графов. Например, для полного графа на рис. 17.8 требуется уже несколько минут, чтобы сгенерировать все 3,6 миллиона возможных перестановок!

17.2.3. Сужение пространства задачи

Я хотел показать вам функцию стоимости для полного пространства задачи с шестью вершинами, потому что эта диаграмма весьма поучительна. Например, здесь видно, что ландшафт имеет множество локальных минимумов с одинаковой стоимостью. Догадываетесь, почему так?

Как всегда, потратьте пару минут (если хотите) на обдумывание ответа, прежде чем переходить к решению.

А теперь рассмотрим следующие цепочки городов в графе:

```
[Opal City, Civic City, New Carthage, Happy Harbor, Gotham City, Metropolis]
[New Carthage, Happy Harbor, Gotham City, Metropolis, Opal City, Civic City]
```

В чем разница между этими двумя перестановками с точки зрения решений? Так как рассматриваются замкнутые маршруты (из первого города через все остальные и обратно в начало), они полностью идентичны, за исключением того, что второй маршрут сдвинут влево на два города. На самом деле стоимость двух маршрутов одинакова (потому что они составлены из одних и тех же ребер).

Из последовательности, данной в качестве решения, можно получить шесть эквивалентных последовательностей, по одной для каждого города, в случае использования его в качестве отправной точки.

Поэтому можно заранее указать город, откуда начинается маршрут, зная, что это не повлияет на результат, но уменьшит количество перестановок, которые нужно исследовать. Так с 720 мы опускаемся до 120 перестановок — уже неплохо, а на больших графах экономия будет еще больше.

Кроме того, учтите, что это ограничение отлично подходит для конкретного экземпляра задачи, решаемой в случае нашей компании электронной коммерции. Маршруты всегда должны начинаться (и заканчиваться) на складе, где товары, предназначенные для доставки, загружаются в транспортное средство.

Если, например, выбрать `New Carthage` в качестве отправной точки, то все еще будет возможно получить несколько одинаково хороших решений (когда есть несколько маршрутов с одинаковой общей стоимостью), но для случая неориентированного графа будет не более двух эквивалентных решений:

```
[New Carthage, Happy Harbor, Gotham City, Metropolis, Opal City, Civic City]
[New Carthage, Civic City, Opal City, Metropolis, Gotham City, Happy Harbor]
```

Потому что, если ребра имеют одинаковую стоимость в обоих направлениях, можно просто пройти цикл (круг) в любом направлении (на рис. 17.10, по часовой стрелке или против часовой стрелки).

Но мы с вами не против наличия двух возможных глобальных решений, поэтому здесь нет необходимости предпринимать дальнейшие действия.

17.2.4. Переходы между состояниями

Теперь пришло время воплотить метод со всеми ограничениями в код. К счастью, это совсем не сложно, учитывая нашу предусмотрительность при разработке алгоритма имитации отжига в листинге 17.1. Нужно передать определение функции, вычисляющей переход в следующее состояние, и тут же можно присвоить то же самое значение первой вершине в последовательности. Функция стоимости известна, так что мы готовы реализовать все недостающие части.

Начнем с функции стоимости, представленной в листинге 17.2. Она, как уже было сказано, представляет собой сумму весов ребер между двумя соседними вершинами в перестановке. Однако при вычислении этого значения нужно обратить особое внимание на пару деталей. Прежде всего предположим, что входной граф имеет ребро между каждой парой вершин. Если это не так, то нужно осуществить проверку и вернуть специальное значение (например, бесконечность или любой достаточно большой вес), чтобы гарантировать, что любое решение, включающее отсутствующие ребра, будет естественным образом отброшено. Другая альтернатива — проверка решений при вычислении переходов и ранняя отбраковка решений с отсутствующими ребрами между соседними вершинами.

Листинг 17.2. Функция стоимости для задачи о коммивояжере

Метод tspTourCost принимает граф и решение-кандидат P (точку в пространстве задачи, то есть перестановку списка вершин в графе) и вычисляет стоимость решения как сумму весов всех ребер в проверяемом решении. Предполагается, что каждая пара вершин в графе соединена ребром (или, как вариант, метод может возвращать специальное значение, например inf, если ребра нет)

```
function tspTourCost(graph, P)
  cost ← 0
  for k in {0..|P|} do
    cost ← cost + graph.edgeBetween(P[k], P[(k+1)%|P|]).weight
  return cost
```

Цикл по членам последовательности

Проверяем наличие ребра между k-й вершиной в последовательности и следующей за ней, находящейся в позиции (k + 1) по модулю от длины последовательности (благодаря этому, достигнув последнего элемента списка, функция вернется назад и следующей вершиной станет первая в последовательности)

Другая деталь, которую хотелось бы подчеркнуть, — необходимость перехода из конца массива в начало, потому что необходимо добавить стоимость ребра между последней и первой вершинами в последовательности (ребра, замыкающего маршрут). Это можно организовать разными способами. Проще всего использовать деление по модулю, а эффективнее — рассматривать последнее ребро как отдельный случай вне цикла `for`.

Для перехода к новому решению есть несколько вариантов, перечислим их ниже.

- *Перестановка соседних вершин* — после выбора случайной позиции в последовательности можно поменять местами вершину с заданным индексом и следующую за ней. Например, решение `[1,2,3,4]` может измениться на `[2,1,3,4]` или `[1,3,2,4]` и т. д. Подобный переход соответствует локальному поиску, когда исследуется только непосредственная окрестность текущего решения; это повышает вероятность, что точная настройка приведет к улучшению, но усложнит выход из локальных минимумов.
- *Перестановка любой пары вершин* — случайно выбираются две позиции и просто меняются местами: `[1,2,3,4]` также может превратиться `[3,2,1,4]`, чего не позволяет предыдущий вариант. Переходы такого типа позволяют выполнять поиск на среднем расстоянии, однако эти два решения все еще довольно близки друг к другу.
- *Генерация новой случайной последовательности* — это позволит перепрыгивать в любую точку пространства задачи на любом этапе алгоритма, что снижает вероятность получить лишь небольшое улучшение до локального минимума на заключительном этапе моделирования, потому что попыток вблизи текущих решений будет относительно немного. В то же время описанный вариант оставляет открытой дверь для дальних прыжков в любую область пространства задачи и обнаружения (пусть и случайно) лучшего решения.

И конечно же, возможны многие другие промежуточные варианты, такие как выполнение фиксированного или случайного количества перестановок в одном переходе или перемещение целого сегмента решения в другое место.

Какой из этих вариантов предпочтительнее? Хороший вопрос, на который сложно ответить с теоретической точки зрения. Остается опробовать их на нашем графе K_{10}, изображенном на рис. 17.8, и посмотреть, какой из них дает лучший средний результат. Для этого выполним 1000 симуляций с одинаковым количеством шагов и с одинаковыми значениями k и α и сравним среднюю стоимость решений. Предположим также, что каждая последовательность начинается с `"New Carthage"`, поэтому число повторяющихся решений сокращается в десять раз. Результаты испытаний показаны в табл. 17.1.

На этот раз пространство задачи по-настоящему огромно, ~3,6 миллиона перестановок, но и не настолько велико, чтобы нельзя было применить метод простого перебора (вооружившись некоторым терпением, потому что для полного перебора требуется время). Благодаря этому узнаем, что наилучшее возможное решение имеет стоимость 1625 (как показано на рис. 17.8):

```
["New Carthage", "Syracuse", "Buffalo", "Pittsburgh", "Harrisburg",
➥ "Opal City", "Metropolis", "Gotham City", "Civic City"]
```

Интересно отметить, что лучший результат дает поиск по среднему диапазону, а худший — локальный поиск. Это означает, что при используемой конфигурации локальный поиск застревает в локальных оптимумах, а генерация

новой случайной последовательности слишком хаотична и не может обеспечить локальную сходимость на заключительном этапе алгоритма, когда температура становится низкой.

Таблица 7.1. Средняя стоимость лучшего решения, найденного алгоритмом имитации отжига с разными алгоритмами переходов

Вариант перехода	Средняя стоимость (α = 0,98, T_0 = 200 и k = 1000)
Перестановка соседних вершин	1937,291
Перестановка случайных вершин	1683,563
Генерация новой случайной последовательности	1831,886

Следует также подчеркнуть, что при использовании других значений параметров ситуация может полностью измениться. Возьмем, к примеру, α, коэффициент спада. Уменьшив его, можно замедлить процесс охлаждения, поэтому возникает вопрос: может ли это уменьшение помочь локальному поиску работать лучше? Давайте попробуем; соответствующие результаты показаны в табл. 17.2.

Таблица 7.2. Средняя стоимость с разными значениями параметра, управляющего спадом температуры

Вариант перехода	Средняя стоимость (α = 0,97)	Средняя стоимость (α = 0,98)	Средняя стоимость (α = 0,99)
Перестановка соседних вершин	1972,502	1937,291	1868,701
Перестановка случайных вершин	1692,044	1683,563	1668,248
Генерация новой случайной последовательности	1816,658	1831,886	1913,416

Увеличение коэффициента скорости спада температуры с 0,97 до 0,99 означает, что охлаждение происходит более медленно и равномерно (взгляните на рис. 17.7, где показаны кривые спада). Но, похоже, это помогает, только когда используется вариант с локальным поиском вокруг текущего решения, и ухудшает ситуацию, когда используется вариант с поиском по всему пространству задачи. Поиск на средних дистанциях, выполняемый при использовании варианта с перестановкой случайно выбранных вершин, в среднем становится еще ближе к оптимальной стоимости.

Эти результаты подчеркивают еще два аспекта, на которые стоит обратить внимание.

- Даже в небольшом пространстве задачи (десять вершин) при выполнении тысяч итераций можно сколь угодно близко подходить к лучшему решению, но не всегда получать лучший результат. Это сознательный риск, на который мы идем, применяя эвристики.
- Поиск лучшей конфигурации для алгоритма оптимизации требует времени, опыта и иногда удачи.

Я также попробовал использовать большие значения α, такие как α = 0,995, которые не показаны в табл. 7.2 отчасти потому, что они приводят к плохим результатам. Вероятно, охлаждение становится слишком медленным, и алгоритм приближается к случайной выборке (это подозрение подтверждается тем, что при меньшем значении k, нормирующей постоянной Больцмана, ухудшение результатов сглаживается).

В заключение раздела, посвященного задаче о коммивояжере, в листинге 17.3 я покажу метод, реализующий случайный переход из текущего состояния в последующее путем объединения всех трех обсуждавшихся до сих пор вариантов. На GitHub можно найти реализацию этого метода в библиотеке JsGraphs.

Листинг 17.3. Функция перехода в новое состояние для задачи о коммивояжере

```
function randomStep(P)
  which ← randomFloat(0,1)
  if which < 0.1 then
    i ← randomInt(0, |P|)
    swap(i, (i+1) % |P|)
  elsif which < 0.8 then
    i ← randomInt(0, |P|)
    j ← randomInt(0, |P|)
    swap(i, j)
  else
    P ← randomPermutation(P)
return P
```

Метод randomStep принимает решение-кандидат P (точку в пространстве задачи, то есть перестановку списка вершин в графе) и вычисляет новое состояние после перехода

Выбираем новое случайное значение между 0 и 1 (не включая последнюю)

Опираясь на заданные заранее значения вероятности, выбираем вариант выполнения перехода

Меняем местами две произвольные вершины. Обратите внимание, что теоретически два индекса могут получиться равными. Чтобы не усложнять код, можно рискнуть конфликтом, который, по сути, означает, что перестановка не будет выполнена. Однако вероятность таких конфликтов, особенно для больших списков, достаточно мала

Меняем местами две соседние вершины

Заменяем текущее состояние новым случайно сгенерированным состоянием (случайной перестановкой)

Интересно увидеть результаты, полученные с помощью этого метода? В табл. 17.3 для сравнения приводятся они и результаты трех «чистых» решений. Как можно заметить, рассматриваемому алгоритму удалось взять лучшее из трех возможных стратегий и уменьшить среднюю стоимость при всех опробованных значениях α.

Таблица 7.3. Сравнение средней стоимости решений, найденных комбинацией оригинальных вариантов перехода

Вариант перехода	Средняя стоимость (α = 0,97)	Средняя стоимость (α = 0,98)	Средняя стоимость (α = 0,99)
Перестановка соседних вершин	1972,502	1937,291	1868,701
Перестановка случайных вершин	1692,044	1683,563	1668,248
Генерация новой случайной последовательности	1816,658	1831,886	1913,416
Комбинация трех вариантов	1683,966	1672,494	1660,904

Судя по результатам, комбинированный вариант при правильно выбранном соотношении дальнего и локального поиска может использовать сильные стороны обоих типов эвристики перехода.

Наилучший результат при использовании комбинированного варианта достигается при большем значении α, а значит, при более равномерном процессе охлаждения.

На рис. 17.12 показан алгоритм в действии: график изменения стоимости при охлаждении системы. Здесь видно, что на первом этапе стоимость сильно колеблется, потом колебания постепенно затухают и, как только температура становится достаточно низкой, поиск направляется к глобальному минимуму.

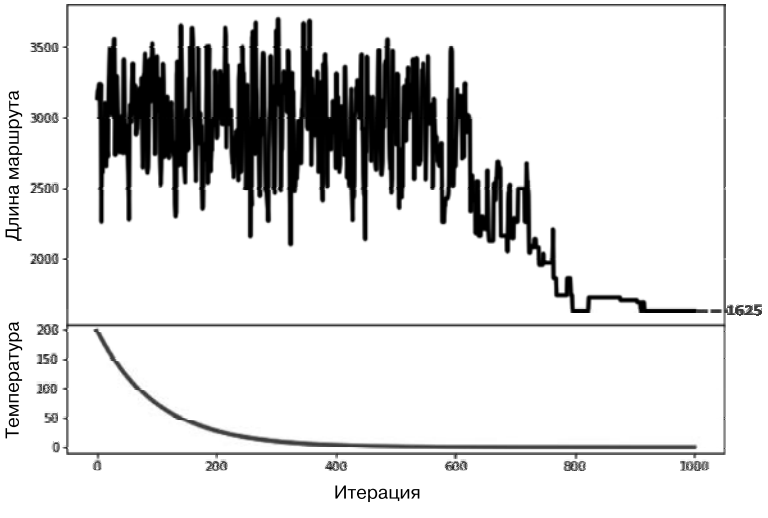


Рис. 17.12. Прогон алгоритма имитации отжига для K_{10} в задаче о коммивояжере. Ему требуется какое-то время для выполнения, но в конце концов находится путь к глобальному минимуму

К этому моменту исследованы не все возможные значения для α, k, T_0, и даже относительные вероятности выбора каждого из трех вариантов перехода в комбинированном методе можно было бы попробовать настроить. Чтобы быть более последовательными, мы должны построить еще несколько примеров с разными графами, а затем написать небольшой фрагмент кода, изменяющий параметры по одному и записывающий найденную среднюю или медианную стоимость. Попробуйте для начала использовать код, включенный в библиотеку JsGraph; это может стать хорошим упражнением.

17.2.5. Перестановка смежных и случайно выбранных вершин

Последний момент, на котором я хотел бы кратко остановиться, — объяснение, почему вариант с перестановкой случайных вершин работает лучше варианта с перестановкой соседних.

Один из факторов, способствующих этому, можно видеть в примере на рис. 17.13: здесь показан случай, когда эвристика перехода локального поиска, меняющая местами только соседние вершины, может дать сбой. Сначала приведен пример перестановки двух случайных вершин (см. рис. 17.13, *А*), дающий значительное уменьшение функции стоимости. Этот переход будет считаться допустимым всегда, независимо от температуры системы.

Для ситуации, когда разрешены только перестановки соседних вершин (см. рис. 17.13, *Б*), в данном примере потребуется несколько перестановок, увеличивающих функцию стоимости (в данном случае две, но их может быть сколько угодно), чтобы перейти в то же состояние, что и на рис. 17.13, *А*. Движение в гору на несколько шагов подряд допустимо на ранних стадиях процесса охлаждения, но по мере приближения к концу вероятность, что такая последовательность плохих ходов будет признана допустимой, становится все меньше. Следовательно, используя только локальный поиск, алгоритм может не суметь улучшить промежуточный результат.

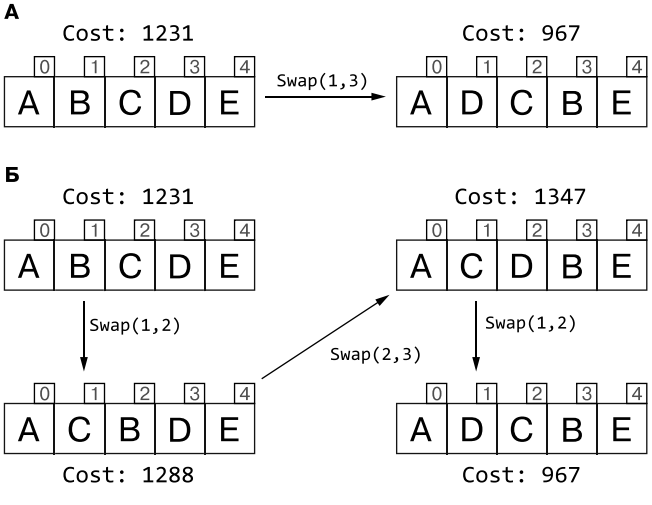


Рис. 17.13. Пример, объясняющий, почему эвристика перехода локального поиска, меняющая местами только соседние вершины, может быть неэффективной. В верхней половине выполняется перестановка двух случайных вершин *А*. Она приводит к существенному уменьшению функции стоимости, поэтому такая перестановка всегда будет считаться допустимой. Когда разрешены только перестановки соседних вершин *Б*, потребуется несколько перестановок, увеличивающих функцию стоимости (в данном случае две, но их может быть сколько угодно), чтобы перейти в состояние *А*. Это может произойти на ранних стадиях процесса охлаждения, но по мере приближения к концу становится маловероятным

Учитывая экспоненциальное уменьшение вероятности посчитать допустимые ходы, ухудшающие стоимость, для метода `randomStep` можно предложить другой подход — повышать вероятность локального поиска на ранних стадиях (когда температура выше) и, наоборот, повышать вероятность дальнего поиска на поздних стадиях.

Я рекомендую попробовать эту идею, изменив код[1] библиотеки JsGraphs, клонированный из репозитория на GitHub. Практические эксперименты — отличный способ глубже изучить имитацию отжига.

17.2.6. Применение решения задачи о коммивояжере

Помимо задачи поиска маршрутов доставки, эффективные алгоритмы решения задачи о коммивояжере могут принести пользу и во многих других реальных сценариях.

Из истории известно, что один из пионеров исследования задачи о коммивояжере в 1940-х годах занялся ею, столкнувшись с необходимостью оптимизировать маршруты школьных автобусов, забирающих детей в школу.

Позже эта задача стала широко применяться к логистике большинства услуг, связанных с планированием маршрутов, от доставки почты до ведения сельского хозяйства.

Затем кабельные компании начали использовать ее для планирования выездов на вызовы, а в последнее время (что более актуально для нашей с вами профессиональной сферы) она стала применяться для оптимизации автоматизированного процесса сверления отверстий и пайки печатных плат и даже в некоторых процессах секвенирования генома.

17.3. ИМИТАЦИЯ ОТЖИГА И ПРЕДСТАВЛЕНИЕ ГРАФА

Чтобы решить задачу о коммивояжере, мы работали с графом, игнорируя его представление[2], и единственное, что имело значение, — это расстояние между парой вершин.

В последних двух главах фокус внимания был нацелен на абстрактные графы и поиск эффективных способов их представления на плоскости.

Вы, наверное, помните, как в начале знакомства с имитацией отжига я говорил, что он хорошо подходит для случаев с дискретными или ступенчатыми функциями стоимости, и вы сами убедились, что в таких случаях целесообразно предпочесть имитацию отжига градиентному спуску.

После всего сказанного могу ли я завершить эту главу, не попытавшись использовать имитацию отжига для решения задачи поиска минимального числа пересечения ребер?

17.3.1. Минимальное число пересечений ребер

Как всегда, чтобы применить шаблонный метод имитации отжига (см. листинг 17.1) к конкретной задаче, нужно выбрать две функции: функцию стоимости и функцию вычисления шага обновления.

---

[1] http://mng.bz/PPWv.

[2] Или, что то же самое, работали ровно с одним представлением графа.

Первая часто подразумевается в самой задаче, так происходит и в данном случае: эта функция должна просто подсчитать количество пересечений ребер.

Вторая функция, вычисляющая шаг обновления, оставляет разработчику алгоритма больше свободы действий. Как перемещать одну вершину? Случайно или как-то иначе? Как далеко? Следует ли попробовать поменять местами две вершины? Следует ли учитывать ребра во время обновления?

Единственный совет, который я могу дать в таких ситуациях, — начать с малого, получить что-то простое, а затем пытаться генерировать новые идеи и проверять их, измеряя, насколько они способствуют сходимости. Именно так мы с вами поступили, решая задачу о коммивояжере: сначала разработали методы, выполняющие одно действие, оценили их эффективность, а затем объединили их в случайный ансамбль.

Для большей конкретики сосредоточимся на поиске представления, близкого к представлению с минимальным числом прямолинейных пересечений (`rcn`) для полного графа K_8. Гипотеза Гая, представленная в главе 15, дает точную формулу числа пересечений этого графа, которое равно 18. Однако там же, в главе 15, было показано, что число прямолинейных пересечений для полных графов может и обычно больше минимального числа пересечений[1]; в этом случае `rcn(K8)` равно `19`.

Итак, начнем с простого шага, слегка перемещающего выбранную вершину в пределах определенного диапазона. Повозившись с диапазонами, принимаем решение обновлять координаты x и y по отдельности, добавляя два случайных значения дельты в пределах 10 % от ширины и высоты области, отведенной для изображения графа, соответственно. Меньшие диапазоны делают алгоритм слишком медленным, а бóльшие приводят к хаотичному поведению алгоритма. Выполнение более 100 прогонов имитации отжига с $k = 0{,}1$, $\alpha = 0{,}97$, $T_0 = 200$ и `maxSteps` = 500 дало среднее значение `rcn`, равное `21.904`. Не плохо, но и не особенно хорошо.

Здесь следует сделать два замечания. Во-первых, использовалось небольшое максимальное количество шагов (если помните, в предыдущем разделе для решения задачи о коммивояжере максимальное число шагов было равно 10 000). Во-вторых, по тем же причинам (которые мы обсудим чуть ниже) нам пришлось сократить количество прогонов с 1000 до 100.

Оба изменения связаны с тем, что вычисление функций стоимости и переходов (путем клонирования представления графа) требуют времени. Конечно, отчасти это связано с использованием универсальной версии имитации отжига. Ее можно было бы оптимизировать, написав специальную версию, которая не требует клонирования всего представления, а просто запоминает, что было изменено и стоимость текущего решения (и, может быть, оптимизации можно было бы достигнуть, вычисляя значение дельты на основе изменений, а не целые решения: например,

[1] Потому что мы ограничены использованием прямолинейных ребер, являющихся подмножеством всех кривых, которые могут применяться для представления ребер.

просто вычисляя количество пересечений для перемещенной вершины до и после перемещения).

Но я не хотел бы сосредоточиваться здесь на этих оптимизациях. Не поймите меня неправильно; оптимизация важна, особенно для кода, передаваемого в промышленную эксплуатацию, но ранняя оптимизация, кроме того, может помешать совершенствованию алгоритма или процесса обучения. Преждевременная оптимизация по-прежнему является источником всех зол (ну пусть не всех, но многих из них!).

В этом случае я предпочел показать простой, чистый, неоптимизированный код, а не более запутанные (хотя и более производительные) подпрограммы.

Следующим моим шагом стало добавление поиска на больших расстояниях: перестановка двух вершин местами в 10 % итераций, чтобы в 90 % случаев применялся локальный поиск. Как вы думаете, каким был результат? Он ухудшился; среднее количество пересечений выросло до 22,89. Однако это не было неожиданностью. Если подумать, то полный граф симметричен, поэтому перестановка двух вершин совершенно бесполезна! Хуже того, она вредна, потому алгоритм тратил 10 % итераций впустую, отсюда и худший результат.

Тем не менее такие перестановки могут быть полезны для других типов графов, несимметричных, поэтому оставим их. (В задачах, рассмотренных выше, используются полные графы, но в реальной жизни алгоритм может и будет применяться к любым графам. В следующем разделе увидим несколько примеров, когда перестановка вершин становится ключом к получению хорошего результата.)

Итак, все еще нужно что-то еще, чтобы улучшить алгоритм. А что, если использовать перемещение случайно выбранной вершины в произвольное место в области рисования?

Я применил этот вариант в 10 % случаев, и среднее число пересечений уменьшилось до 19,17, а это означает, что алгоритм почти всегда находил наилучшее решение. Кстати говоря, на рис. 17.14 сравниваются два решения для представления K_8. Представление слева было найдено алгоритмом случайной выборки, описанным в главе 16, а справа — результат метода, основанного на имитации отжига.

Разумеется, можно было бы продолжить улучшение алгоритма путем точной настройки параметров и, возможно, разработки более эффективных вариантов настройки решений.

И последнее, но не менее важное: алгоритм следует опробовать и оптимизировать на разнообразном наборе графов, чтобы избежать его переобучения на полных графах. (В качестве альтернативы, столкнувшись с конкретной задачей, можно настроить параметры на небольших экземплярах графов, которые ожидается встретить, и потом применить настроенную версию к реальным экземплярам.)

Из того, что можно увидеть, например, при масштабировании до K_{10} (рис. 17.15), полученная конфигурация хорошо работает с большими полными графами.

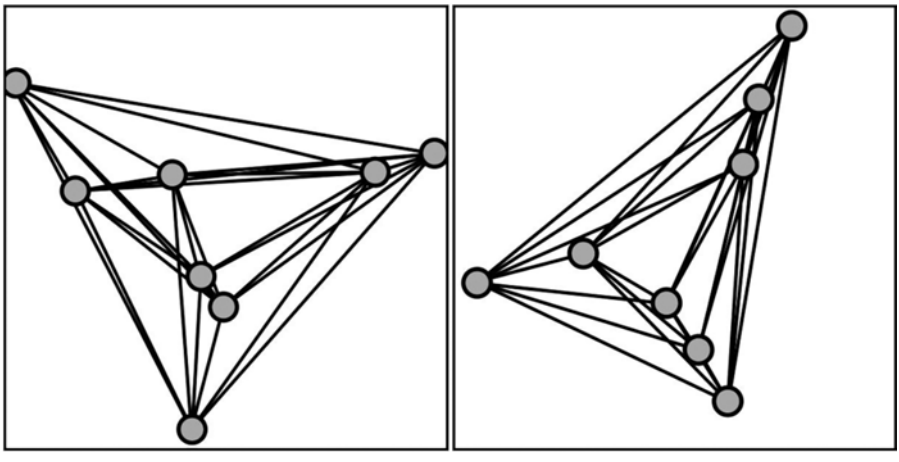


Рис. 17.14. Два представления полного графа K_8. Представление *слева* получено алгоритмом случайной выборки, *справа* — результат имитации отжига. Оба алгоритма выполнили 500 итераций: случайная выборка смогла получить 27 пересечений, а имитация отжига дала оптимальное представление с 19 пересечениями

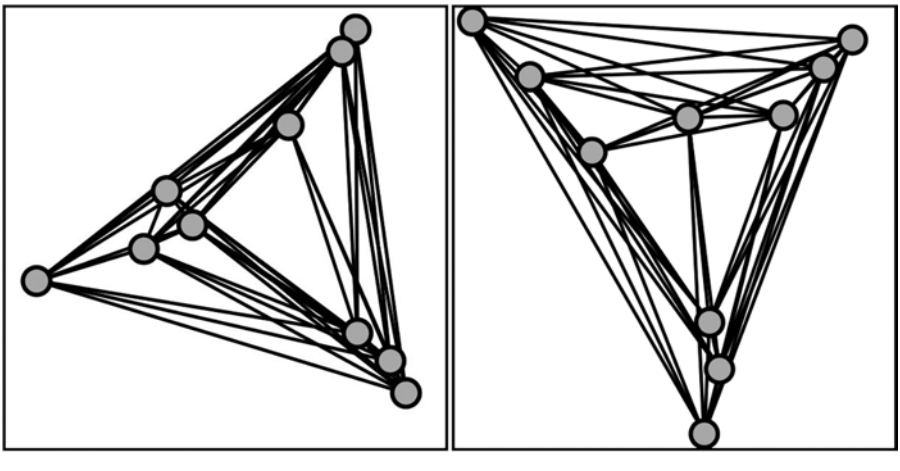


Рис. 17.15. Два представления полного графа K_{10}. Имитация отжига *(справа)* дала оптимальное представление с 62 пересечениями, тогда как лучший результат алгоритма случайной выборки имеет 81 пересечение

17.3.2. Силовые алгоритмы визуализации

В разделе 16.5 был представлен класс алгоритмов рисования графов, называемых силовыми алгоритмами визуализации, которые используют аналогии, основанные на физике, для вычисления эстетически привлекательных представлений графов. Рисунок 17.16 напоминает, почему хорошее представление важно для визуализации графов.

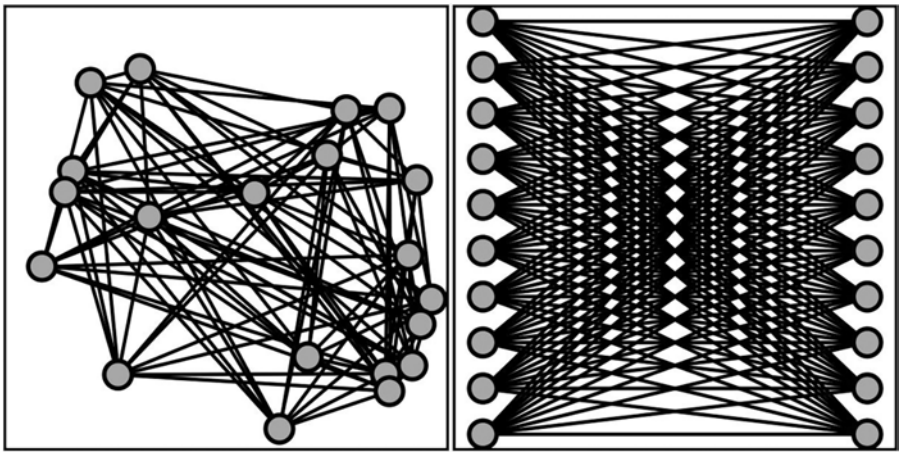


Рис. 17.16. Хорошее представление может изменить восприятие графа. Здесь изображены полные двудольные графы $K_{10,10}$: со случайно выбранным *(слева)* и симметричным представлением *(справа)*. Какое из них лучше отражает структуру этого графа?

В конце 1980-х Питер Идес (Peter Eades) предложил[1] модель изображения неориентированных графов с использованием «пружинных представлений», которая затем была усовершенствована[2] Камадой (Kamada) и Каваи (Kawai), представившими идею оптимальной длины ребра и последовательного обновления вершин (путем перемещения только одной вершины на каждом шаге).

Алгоритм переводится в состояние минимальной энергии с помощью градиентного спуска. Однако часто детерминированные алгоритмы, такие как градиентный спуск, застревают в локальных минимумах, достигая равновесного состояния системы, но не состояния с минимально возможной энергией.

Неудивительно, что имитация отжига может помочь и в этом случае. Дэвидсон (Davidson) и Харел (Harel) первыми использовали эту методику[3] в поиске оптимальных представлений, чтобы избежать ловушек локальных минимумов.

Проблема стандартной имитации отжига в том, что случайный поиск делает сходимость слишком медленной. Чтобы устранить это ограничение, несколько авторов предложили гибридные решения, использующие сильные стороны обоих

---

1 *Eades P.* A Heuristic for Graph Drawing // Congressus Numerantium, 1984. — № 42. — P. 149–160.

2 *Kamada T., Kawai S.* An algorithm for drawing general undirected graphs // Information Processing Letters, 1989. — № 31.

3 *Davidson R., Harel D.* Drawing graphs nicely using simulated annealing // ACM Transactions on Graphics, 1996. — № 15 (4). — P. 301–331.

алгоритмов. Алгоритм GEM[1] выделяется среди них своим инновационным подходом и впечатляющими результатами.

GEM не использует имитацию отжига, но заимствует из нее понятие температуры. В нем нет цикла охлаждения, а температура (которая по-прежнему выражает степень «хаоса» в системе) используется для управления диапазоном движения вершин при обновлении — она вычисляется для каждой вершины после обновления и масштабируется для сглаживания колебаний вершин.

Поскольку алгоритм GEM нельзя применить напрямую как пример имитации отжига, остановимся на алгоритме, разработанном Дэвидсоном и Харелом, который дает сопоставимые результаты.

Как упоминалось в предыдущих главах, первый шаг при создании алгоритмов рисования графов — выбор критериев оценки качества представления, ведь число пересечений не единственный критерий, важный для получения красивых графов. Подход, реализованный Дэвидсоном и Харелом, использует пять критериев:

- равномерное распределение вершин по холсту;
- удержание вершин подальше от границ;
- уравнивание длин ребер;
- минимизация пересечений ребер;
- исключение пересечений ребер и вершин.

Следует также уточнить, что алгоритм предполагает рисование ребер в виде прямолинейных отрезков. А теперь посмотрим, как эти пять критериев преобразуются в формулы, для этого запишем пять компонент функции стоимости.

Первая компонента алгоритма выражается формулой потенциальной электрической энергии; для двух вершин, v_i и v_j, вычисляется отношение:

$$\frac{\lambda_1}{d_{ij}^2},$$

где d_{ij} — расстояние между двумя вершинами, а λ_1 — параметр, который передается алгоритму и управляет весом этой компоненты. То есть это нормирующий коэффициент, определяющий относительную важность самого критерия. Такой член действует подобно силе отталкивания, поэтому более высокие значения заставляют алгоритм предпочитать представления с бóльшими расстояниями между вершинами.

Чтобы вершины не касались границ, добавляется еще одна компонента. Для каждой вершины v_i вычисляется:

$$\lambda_2\left(\frac{1}{r_i^2}+\frac{1}{l_i^2}+\frac{1}{t_i^2}+\frac{1}{b_i^2}\right).$$

[1] *Frick A., Ludwig A., Mehldau H.* A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration) // International Symposium on Graph Drawing. — Springer, Berlin, Heidelberg, 1994.

Здесь значения r_i, l_i, t_i и b_i — это расстояния между v_i и краями прямоугольного холста, на котором изображается граф; λ_2 — еще один нормирующий коэффициент, предназначенный для взвешивания этого члена. С увеличением значения λ_2 представления, имеющие вершины близко к границам, будут получать больший штраф.

Теперь поговорим о ребрах. Для каждого ребра $e_k = u \rightarrow v$ вычисляется:

$$\lambda_3 d_k^2,$$

где d_k = `distance(u,v)` — длина ребра, а λ_3 — обычный нормирующий коэффициент. Этот член действует как сила притяжения, поэтому большие значения λ_3 способствуют уменьшению расстояний между смежными вершинами.

Для критерия пересечений ребер можно просто подсчитать их и умножить количество пересечений на нормирующий коэффициент λ_4.

Наконец, чтобы вершины не пересекались с ребрами (если вы помните, это был один из ключевых критериев, использовавшихся в главе 16 для проверки представлений), можно добавить следующий член для каждой пары «вершина — ребро»:

$$\frac{\lambda_5}{g_{kl}^2},$$

где g_{kl} = `distance(`e_k`,`v_l`)`, а λ_5 — еще один нормирующий коэффициент.

Подобный член (еще одна сила отталкивания) довольно дорогостоящий (для определения расстояния от ребра до вершины требуется значительный объем вычислений, как мы видели в главе 15), и даже в оригинальной статье он не используется по умолчанию. Пока этот член опустим. Вы можете реализовать его применение в качестве упражнения, а затем поэкспериментировать на примерах, которые будут показаны.

В листинге 17.4 представлена полная реализация функции стоимости (со всеми пятью компонентами), но в ней примеры были получены без использования члена, определяющего расстояния «ребро — вершина».

На следующем шаге необходимо реализовать методы вычисления переходов к новым решениям; к счастью, для этого можно повторно использовать те методы, что были определены в предыдущем разделе. В конце концов, пространство задачи одно и то же; единственное, что нужно изменить, — функцию стоимости, потому что критерии хорошего представления изменились.

В алгоритме в статье использовалась только эвристика локального поиска, но не с постоянным диапазоном, а с уменьшением расстояния, на которое можно переместить вершину, по мере продвижения алгоритма.

Работающий код, реализованный для библиотеки JsGraphs, можно найти на GitHub[1].

В этом случае нет особого смысла проверять качество результатов по средним числам, полученным в множестве повторений: нужно, чтобы графы выглядели красиво,

[1] http://mng.bz/JD1a.

а волшебной формулы измерения «красоты» не существует. Единственный способ оценить результаты — представить их человеческому глазу.

Листинг 17.4. Функция стоимости для алгоритма Дэвидсона и Харела

```
Метод cost принимает текущее решение, P (представление графа —
точку в пространстве задачи), ширину (w) и высоту (h) холста, а также
нормирующие коэффициенты для членов формулы

function cost(P, w, h, lambda1, lambda2, lambda3, lambda4, lambda5)
  total ← 0
  for v in P.vertices do                          ← Цикл по вершинам в графе
    total ← total + lambda2 × ((1/x²) + (1/y²) + (1/(w-x)²) + (1/(h-y)²))
    for u in P.vertices-{v} do                    ← Цикл по всем вершинам, кроме v
      total ← total + lambda1 / distance(u, v)²
  for e=(u,v) in P.edges do
    total ← total + lambda3 × distance(u, v)²
    for z in P.vertices-{u,v} do
      total ← total + lambda5 / distance(e, z)²
  total ← total + lambda4 × P.intersections()
  return total

Прибавляем первый член функции стоимости — силу отталкивания между вершинами

Прибавляем третий член — силу притягивания между вершинами, связанными ребром

Цикл по всем вершинам, за исключением конечных точек ребра

Наконец, прибавляем четвертую компоненту, пропорциональную количеству пересечений ребер в представлении

Цикл по ребрам, для каждого ребра e между вершинами u и v...

Прибавляем пятую компоненту — силу отталкивания между ребрами и вершинами (это, конечно, не относится к вершинам на концах ребра)

Прибавляем второй член целевой функции — силу отталкивания между вершинами и границами холста
```

Рисунок 17.17, я думаю, идеально объясняет то, что строилось на протяжении двух последних разделов. Мы попытались получить красивое представление для графа в виде квадратной сетки со стороной 4.

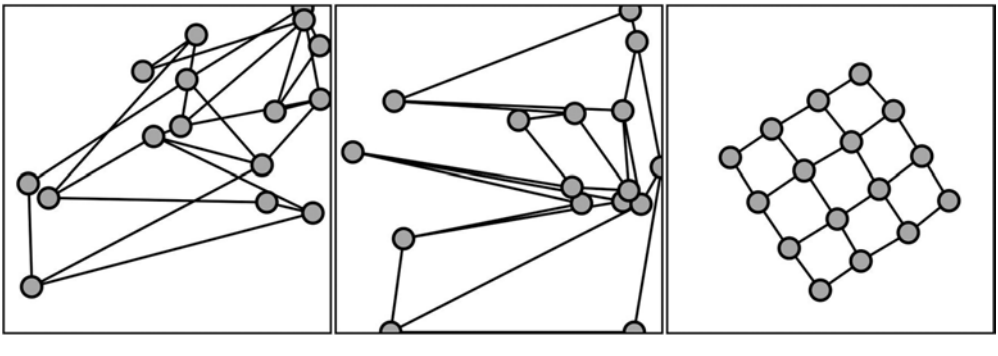


Рис. 17.17. Граф в форме квадратной сетки с 16 вершинами. Представление *слева* получено методом случайной выборки. Представление *в центре* получено имитацией отжига с минимальным числом пересечений в качестве критерия. Представление *справа* получено с помощью алгоритма Дэвидсона и Харела

Случайная выборка изо всех сил пытается найти даже представление без пересечений, используя алгоритм из подраздела 17.3.1, однако не особенно хорошо справляется с задачей прояснения структуры графа.

Напротив, граф справа выглядит почти идеально симметричным. Смогли бы вы понять форму этого графа по двум другим представлениям?

Для справки: это представление было получено с использованием значений, приведенных в табл. 17.4.

Таблица 17.4. Значения параметров алгоритма Дэвидсона и Харела, использованные для получения представления графа в форме квадратной сетки, изображенной на рис. 17.17

Параметр	Назначение	Величина
T_0	Начальная температура	1000
k	Постоянная (псевдо) Больцмана	1e+8
α	Коэффициент падения температуры	0,95
`maxSteps`		10 000
λ_1	Расстояние до границы холста	10
λ_2	Расстояние между вершинами	0.01
λ_3	Длина ребра	2e–8
λ_4	Пересечение ребер	100

На рис. 17.18 показана еще пара примеров с сеткой большего размера и графом другого типа — треугольной сеткой. Для обоих создаются прилично выглядящие представления после некоторой настройки параметров.

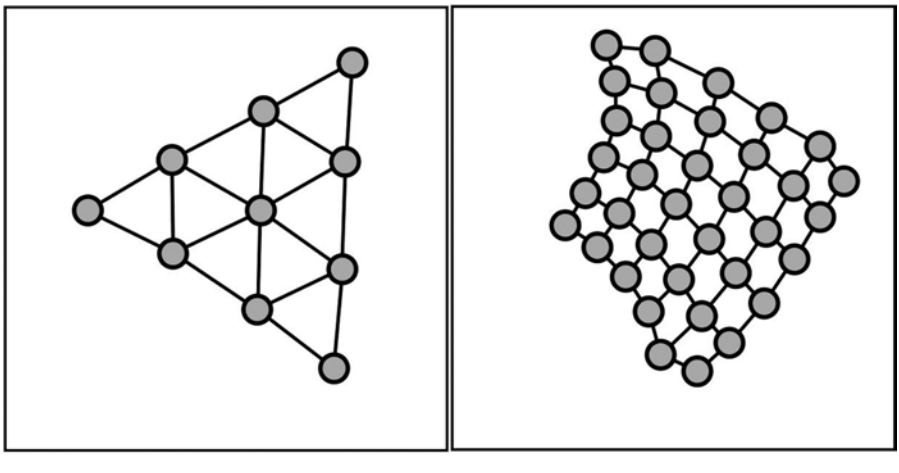


Рис. 17.18. Представления графа в форме треугольной сетки с четырьмя вершинами на каждой стороне и графа в форме квадратной сетки с 36 вершинами. Оба представления получены с помощью алгоритма Дэвидсона и Харела

Но прежде, чем восхищаться и заявлять, что найден идеальный алгоритм для всех графов, следует опробовать его на других типах графов.

На рис. 17.19 показаны результаты для K_5 и K_7. Для каждого графа найдено представление с минимально возможным количеством пересечений, и вершины выглядят хорошо распределенными, но, как видите, эти представления неидеальны, так как некоторые вершины находятся слишком близко к несмежным ребрам и пересекаются с ними.

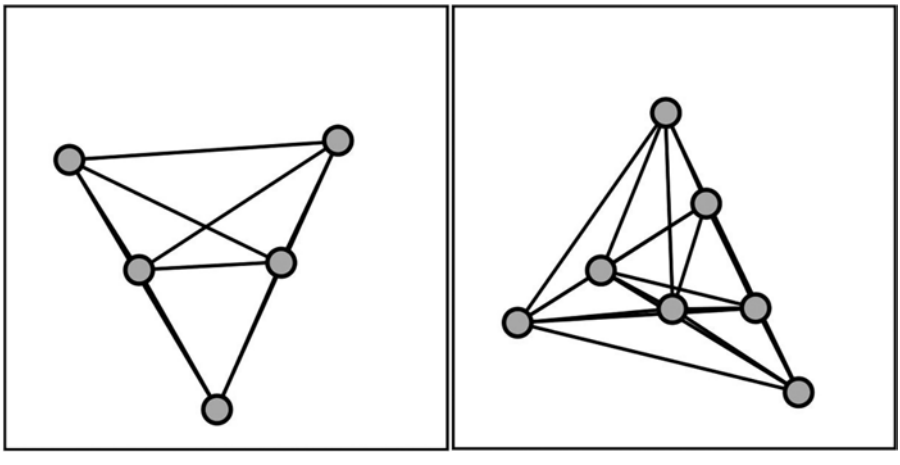


Рис. 17.19. Представления полных графов K_5 и K_7, полученные с помощью алгоритма Дэвидсона и Харела. Здесь можно было бы использовать пятую компоненту функции стоимости, отталкивающую вершины подальше от ребер

Эти ситуации можно исправить, добавив пятую компоненту функции стоимости, препятствующую сближению вершин и ребер.

И в завершение этой главы вот вам небольшое домашнее задание: дополните функцию стоимости и найдите лучшие представления для этих графов.

РЕЗЮМЕ

- Имитация отжига — это стохастическая альтернатива градиентному спуску, в которой используются понятия из физики для предоставления динамического метода, фокусирующегося на поиске в большом диапазоне на начальных этапах и точной настройке в конце.
- Плюсы: имитация отжига предпочтительнее, когда область определения функции стоимости дискретна, функция стоимости не дифференцируема или имеет ступенчатый график с большим количеством локальных минимумов.

- Минусы: имитации отжига следует избегать, когда локальные минимумы находятся в узких «долинах», так как маловероятно, что алгоритму удастся найти их.
- По сравнению с градиентным спуском, выбирающим самую крутую траекторию на пути к минимуму, имитация отжига требует гораздо больше итераций, чтобы добраться до той же точки.
- Имитация отжига не гарантирует оптимального решения. Если субоптимальные решения недопустимы, то следует использовать другой алгоритм.
- Поиск правильной конфигурации требует времени и должен выполняться для каждой задачи.
- Задачу коммивояжера решить сложно, но она повсеместно встречается в логистике, планировании и электронике (для печатных плат).

18 *Генетические алгоритмы: заимствованная из биологии быстросходящаяся оптимизация*

В этой главе

- ✓ Введение в генетические алгоритмы.
- ✓ Сравнение генетических алгоритмов с имитацией отжига.
- ✓ Решение задачи «Подготовка к полету на Марс» с помощью генетических алгоритмов.
- ✓ Решение задачи о коммивояжере и подбор заказов для загрузки транспортных средств с помощью генетических алгоритмов.
- ✓ Создание генетического алгоритма для решения задачи минимального вершинного покрытия.
- ✓ Обсуждение применимости генетических алгоритмов.

Градиентный спуск и *имитация отжига* — отличные методы оптимизации, но оба имеют недостатки: первый работает быстро, но склонен застревать в локальных минимумах и требует использования дифференцируемой функции стоимости, а второй может сходиться довольно медленно.

В этой главе вы познакомитесь с *генетическими алгоритмами*, еще одним методом оптимизации, который использует подход, подсмотренный у природы, для преодоления обеих проблем, не склонен застревать в локальных минимумах благодаря эволюции пула решений и в то же время ускоряет сходимость.

Применяться этот метод будет для решения нескольких сложных задач, которые нельзя эффективно решить с помощью детерминированных алгоритмов. Сначала

я объясню теорию, лежащую в основе генетических алгоритмов, взяв в качестве примера задачу упаковки рюкзака 0-1, представленную в главе 1. Затем кратко обсудим генетический алгоритм для решения задачи о коммивояжере[1] и посмотрим, как (и почему) он сходится быстрее, чем имитация отжига, а затем рассмотрим две новые задачи:

- *вершинное покрытие*, решение которой пригодится во многих областях, от сетевой безопасности до биоинформатики;
- *максимальный поток*, которую часто приходится решать при подключении к сети, оптимизации кода в компиляторах и во многих других областях.

18.1. ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ

Главная задача алгоритмов оптимизации — заменить поиск методом перебора по всему пространству задачи локальным поиском, который отфильтровывает как можно больше точек в этом пространстве, то есть осуществляет поиск того же результата, рассматривая гораздо меньшую область.

Алгоритмы оптимизации исследуют окрестности текущих решений в пространстве поиска и делают это либо детерминированно, либо случайно, пытаясь выявить многообещающие области, в которых можно ожидать появления лучших решений, чем те, что были найдены ранее.

Как выполняется эта фильтрация, как выбирается диапазон поиска и насколько он «локален», как происходят перемещения в пространстве — случайным или детерминированным образом — все это ключевые факторы, характеризующие различные методы, которые были описаны в главах 16 и 17.

В предыдущей главе обсуждалась имитация отжига, и стало понятно, что этот алгоритм сходится к почти оптимальным решениям, допуская переходы вверх, тогда как градиентный спуск движется только вниз, к лучшим решениям, из-за чего легко застревает в локальных минимумах. Как результат, имитация отжига может находить лучшее решение, чем градиентный спуск, когда функция стоимости имеет много локальных (суб)минимумов.

Однако, несмотря на все свои достоинства, имитация отжига неидеальна, и вы видели два основных ее недостатка. Напомню их, приведя список ниже.

- Алгоритм медленно сходится. В отличие от градиентного спуска, который детерминированно выбирает самый быстрый путь вниз[2], имитация отжига (являясь стохастическим алгоритмом) случайно блуждает по ландшафту функции

---

[1] Эта задача подробно разбиралась в разделе 17.2.

[2] Технически он следует пути скорейшего спуска, делая жадный локально оптимальный выбор на каждом шаге. Часто этот выбор не является глобально оптимальным, и, следовательно, градиентный спуск не гарантирует достижения глобального оптимума, если только функция стоимости не имеет определенной выпуклой формы с единственной точкой минимума.

стоимости, и поэтому ей может потребоваться много попыток, чтобы найти правильное направление.

- Если функция стоимости имеет локальные/глобальные минимумы в узких впадинах, то вероятность случайного «попадания» имитации отжига в эти впадины может быть настолько низкой, что, скорее всего, ей вообще не удастся найти решение, близкое к оптимальному.

Раздел 17.1 содержит довольно подробный анализ того, почему имитация отжига работает хорошо и как она осуществляет динамическую фильтрацию, косвенно ограничивая в пространстве задачи точки (в зависимости от их стоимости), которые могут быть достигнуты на данном этапе. Как вы наверняка помните, изначально алгоритм хаотично прыгает от точки к точке, то поднимаясь, то спускаясь, исследуя ландшафт в целом.

Мы с вами уже видели, как оптимизация находит решение, близкое к оптимальному, на ранних стадиях, а затем уходит от него, потому что на начальной фазе часто допускаются переходы в гору. Проблема в том, что имитация отжига не имеет памяти, она не отслеживает прежние решения, и — особенно если допускаются дальние переходы — существует конкретный риск, что алгоритм никогда не вернется к такому хорошему прошлому решению.

Еще более вероятно, что на каком-то раннем этапе алгоритм действительно найдет долину, содержащую глобальный минимум, но не достигнет ее дна (возможно, приземлившись где-то на склоне, ведущем в долину), удалившись и больше никогда не войдя в нее.

Эти ситуации иллюстрирует рис. 18.1. В разделе 17.1 также обсуждался вариант имитации отжига с перезапусками, способный отслеживать прошлые решения и — в случае зависания — случайным образом перезапускаться с одной из этих прошлых позиций, чтобы проверить, найдено ли решение лучше текущего.

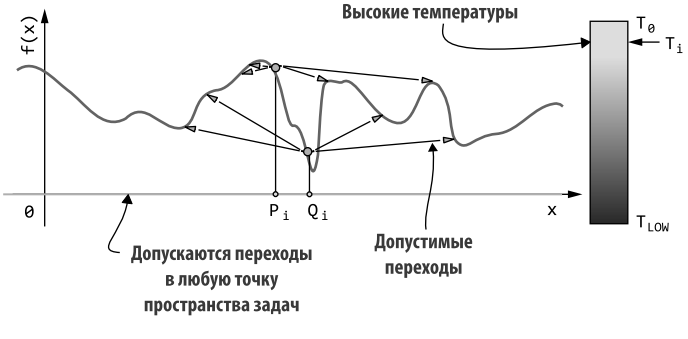


Рис. 18.1. Пример сценария, когда имитация отжига находит многообещающее (P_i) или даже хорошее (Q_i) решение на ранней стадии (на это указывает температура системы T_i, все еще близкая к T_0): поскольку все переходы, даже в гору, допускаются при высокой температуре, алгоритм может отойти от наилучшей точки в ландшафте функции стоимости и никогда не вернуться назад

Это усовершенствование может несколько помочь в первой ситуации, когда на ранней стадии обнаруживается отличное решение, но маловероятно, что он улучшит ситуацию последнюю (обнаружение перспективной позиции на входе в долину), потому что сохраняется ограниченное количество лучших решений, найденных ранее. То есть, даже если оптимизации удалось найти начало пути к глобальному минимуму, она забудет об этом, если только ей не посчастливится приземлиться близко к дну долины.

18.1.1. Заимствование у природы

Метод имитации отжига был заимствован из металлургии, он имитирует процесс охлаждения и перехода системы из хаотического в упорядоченное состояние. Собственно говоря, природа часто служила отличным источником вдохновения для математики и информатики. Просто вспомните о нейронных сетях, одном из наиболее ярких, пожалуй, примеров последнего времени.

В отношении заимствования биологических процессов, таких как нейронные сети, причина очевидна. Удивительные и эффективные решения можно найти повсюду в природе, они адаптировались и совершенствовались миллионы лет, потому что от их эффективности всегда зависело выживание организмов.

Генетические алгоритмы — еще один пример алгоритмов, скопированных из биологии, и, более того, они основаны именно на принципе «эволюции в ответ на стимулирующие воздействия окружающей среды».

Как показано на рис. 18.2, генетический алгоритм — это алгоритм оптимизации, поддерживающий набор решений на каждой итерации. В отличие от имитации отжига он поддерживает большое разнообразие решений, одновременно исследуя разные области ландшафта функции стоимости.

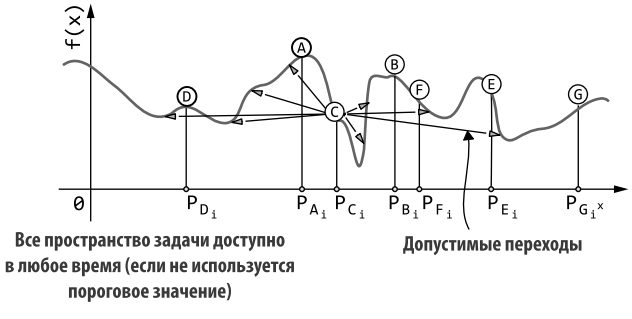


Рис. 18.2. Пример решения задачи с рис. 18.1 с помощью генетического алгоритма. На любой итерации (здесь предполагается, что алгоритм выполняет i-ю итерацию) как в начале, так и в конце моделирования алгоритм поддерживает пул возможных решений. На следующей итерации пул (целиком) получит новый набор решений-кандидатов (на основе текущих — мы увидим это ниже в этом же разделе). Обычно новые решения могут находиться где угодно в пространстве задачи

Один из аспектов, которые нельзя показать на графике «функции стоимости», — генетические алгоритмы также постепенно способствуют эволюции решений, рекомбинируя их. Это позволяет алгоритмам использовать сильные стороны разных решений и объединять их в лучшее. Возьмем, к примеру, задачу о коммивояжере, описанную в разделе 17.2 (мы вновь вернемся к ней в следующем разделе). Представьте, что у нас есть два плохих решения, каждое из которых имеет наилучшую возможную последовательность для разных половин вершин, но каждое содержит ужасно неэффективную последовательность в оставшейся половине. Комбинируя эти два решения правильным[1] образом, можно взять хорошие половины каждого из них и получить отличное решение-кандидат, может быть (если повезет!), даже лучшее из возможных.

Эта идея комбинирования решений — то, чего нет ни в градиентном спуске (очевидно!), ни в имитации отжига, где в каждый момент времени хранится и «изучается» одно решение.

Позже увидим, что эта новая идея воплощена в новом операторе, *операторе скрещивания*, заимствованном из биологии. Но не будем забегать вперед и для начала выясним, что послужило отправной точкой для возникновения генетических алгоритмов и откуда пошло само его название.

Генетический алгоритм — метаэвристика оптимизации, основанная на принципах генетики и естественного отбора. Выше уже говорилось, что этот метод оптимизации поддерживает пул решений, но не было сказано, что сами решения имитируют популяцию, эволюционирующую в течение нескольких поколений — каждый организм в популяции определяется хромосомой (иногда парой хромосом), которая кодирует единственное решение оптимизируемой задачи.

АЛГОРИТМЫ, ЗАИМСТВОВАННЫЕ ИЗ БИОЛОГИИ

Большинство живых организмов на этой планете состоит из клеток[2], каждая из которых несет один и тот же[3] набор генов в множестве хромосом (обычно более одной для сложных организмов, таких как животные или растения). Любая хромосома — это молекула *ДНК (дезоксирибонуклеиновой кислоты)*, которую можно разбить на последовательность нуклеотидов. Каждый из них, в свою очередь, является последовательностью азотсодержащих нуклеиновых оснований, кодирующих информацию. Информация, содержащаяся в ДНК клеток, используется ими для управления своим поведением и синтеза белков. Эти понятия кратко иллюстрирует рис. 18.3.

---

1 Мы подробно поговорим об этом в разделе 18.2.

2 За исключением вирусов, которые представляют собой просто молекулы ДНК или РНК, заключенные в белковую оболочку.

3 Примерно один и тот же, так как по разным причинам могут иметь место небольшие локальные вариации, в том числе из-за ошибок копирования.

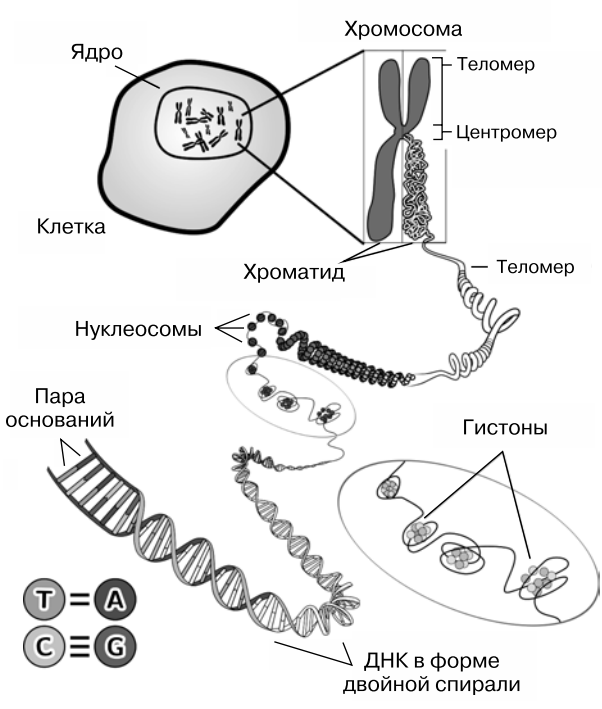


Рис. 18.3. От клеток к ДНК: геном клетки содержится в ее ядре, внутри структур, называемых хромосомами, чьи теломеры при распутывании обнаруживают двойные спирали ДНК, последовательности, состоящие всего из четырех азотистых оснований (следовательно, ДНК можно считать кодировкой по основанию 4). Источник: https://commons.wikimedia.org/w/index.php?curid=10856406, автор: KES47

Проще говоря, хромосомы кодируют информацию, определяющую поведение организма *(генетика)* и посредством этого способность организма адаптироваться к окружающей среде, выживать и производить потомство *(естественный отбор)*.

Ученый-информатик Джон Холланд (John Holland), вдохновленный этим механизмом, в начале 1970-х разработал[1] способ использования этих двух принципов развития для искусственных систем. Один из его учеников, Дэвид Голдберг (David Goldberg, долгое время считавшийся авторитетным экспертом по этой теме), в конце 1980-х годов[2] посвятил популяризации этого подхода свое исследование и свою книгу.

---

[1] Книга Холланда была первоначально опубликована в 1975 году. В настоящее время можно найти издание MIT Press 1992 года: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence, MIT Press, 1992.

[2] Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Professional, 1988.

Основная идея, показанная на рис. 18.4 следующего раздела, состоит в том, чтобы закодировать решение как хромосому и присвоить каждую хромосому организму. Качество этого решения естественным образом определяет, насколько соответствующий организм приспособлен к окружающей среде. Приспособленность, в свою очередь, является косвенным показателем шансов организма воспроизводить и передавать свой генетический материал следующему поколению. Как и в природе, альфа-индивидуумы популяции, более сильные (или более быстрые, или более умные), обычно имеют более высокие шансы прожить дольше и найти себе пару.

Но самый лучший, пожалуй, способ понять генетические алгоритмы — увидеть их в действии. С этой целью и для большей конкретики, описывая строительные блоки данного метаалгоритма, мы попутно рассмотрим пример, чтобы вы могли наглядно представить, как работает каждая компонента. Помните задачу о рюкзаке 0-1? В главе 1 мы использовали эту задачу для моделирования ситуации «подготовки к полету на Марс». Там упоминалось, что существует псевдополиномиальный динамический алгоритм, способный дать абсолютно лучшее решение, но он может быть слишком медленным, если вместимость рюкзака окажется чрезмерно велика. Существуют приближенные алгоритмы ветвей и границ, которые могут вычислять решения, близкие к оптимальным (с некоторыми гарантиями верхней границы и за разумное время), хотя их эффективность достигается за счет большой сложности[1].

Для создания быстрого, ясного и простого алгоритма оптимизации за основу возьмем генетический алгоритм, способный находить приближенные к оптимальным решения для задачи о рюкзаке.

Прежде чем углубиться в детали генетических алгоритмов, я предлагаю немного освежить память и быстро повторить определение задачи.

ПРИМЕЧАНИЕ

В задаче о рюкзаке 0-1 у нас есть контейнер с ограниченной вместимостью M и набор предметов, каждый из которых характеризуется весом (или любой другой мерой вместимости) w и стоимостью v. Положить предмет в рюкзак (то есть в контейнер, который мы решили использовать!) можно только целиком; предметы нельзя дробить на части. Сумма весов всех имеющихся предметов превышает вместимость рюкзака, поэтому нужно выбрать такое их подмножество, которое максимизирует суммарную ценность содержимого рюкзака.

В подразделе 1.4.1 решалась более конкретная задача: выбирая упаковки с продуктами из списка, показанного (для удобства) в табл. 18.1, мы должны были наполнить контейнер, вмещающий не более 1 т, но так, чтобы получить максимальную общую энергетическую ценность.

---

[1] Например, алгоритм Мартелло — Тота, являющийся одним из самых современных решений: *Martello S., Toth P.* A bound and bound algorithm for the zero-one multiple knapsack problem // Discrete Applied Mathematics, 1981. — № 3, 4. — P. 275–288.

То есть можно было легко убедиться, что пшеничная мука, рис и помидоры дают в сумме максимальный допустимый вес, но не дают максимального количества калорий.

В главе 1 кратко описано ключевое соображение, используемое эвристикой ветвей и границ для аппроксимации этой задачи. Она вычисляет отношение энергетической ценности к весу для каждого продукта и отдает предпочтение продуктам с более высоким соотношением. Однако, несмотря на использование такой эвристики алгоритмом Мартелло — Тота в качестве отправной точки, простой выбор продуктов в порядке убывания количества калорий на килограмм не дает наилучшего возможного решения.

И действительно, алгоритм динамического программирования, который точно решает задачу о рюкзаке 0-1, даже не будет вычислять это отношение. Мы не будем его использовать и здесь, поэтому оно даже не показано в табл. 18.1.

Таблица 18.1. Список продуктов, которые можно взять в полет на Марс, с их весами и энергетической ценностью

Продукт	Вес (кг)	Общее число калорий
Картофель	800	1 502 000
Пшеничная мука	400	1 444 000
Рис	300	1 122 000
Фасоль (консервированная)	300	690 000
Помидоры (консервированные)	300	237 000
Клубничное варенье	50	130 000
Арахисовое масло	20	117 800

Чтобы вспомнить более тонкие детали, вернитесь к разделу 1.4. А теперь перейдем к обсуждению основных компонент генетических алгоритмов (ГА).

Для полного определения алгоритма оптимизации необходимо задать следующие компоненты:

- *как кодировать решение* — хромосомы, для ГА;
- *операторы переходов* — скрещивание и мутации;
- *способ измерения стоимости* — функция оценки приспособленности;
- *как эволюционирует система* — поколения и естественный отбор.

18.1.2. Хромосомы

Как уже упоминалось, хромосомы кодируют решение задачи. В оригинальной работе Холланда предполагалось, что хромосомы представлены цепочками битов, последовательностями нулей и единиц. Не все задачи можно закодировать последовательностями двоичных битов (мы увидим это на примере задачи о коммивояжере в разделе 18.2), поэтому позднее была предложена более общая версия, позволяющая представлять гены (значения в хромосомах) действительными значениями.

Концептуально эти версии можно считать эквивалентными, но всякий раз, когда для представления хромосом используются последовательности действительных чисел, а не двоичных битов, может потребоваться наложить ограничения на допустимые значения, которые они могут хранить; операторы, действующие на организмы, должны быть скорректированы так, чтобы они проверяли и поддерживали эти ограничения.

К счастью для нас, задача о рюкзаке 0-1 — отличный пример для обсуждения оригинального генетического алгоритма, потому что решение этой задачи можно закодировать точно в виде последовательности битов. Ноль означает, что данный продукт остается на Земле, а единица, — что он добавляется в трюм корабля, отправляющегося на Марс.

На рис. 18.4 показано двоичное представление хромосомы, которое можно использовать в нашем примере. Два разных решения (или *фенотипа*, если говорить терминами биологии) могут быть представлены двумя хромосомами (или *генотипами*): их различия сводятся к различиям между двумя последовательностями битов.

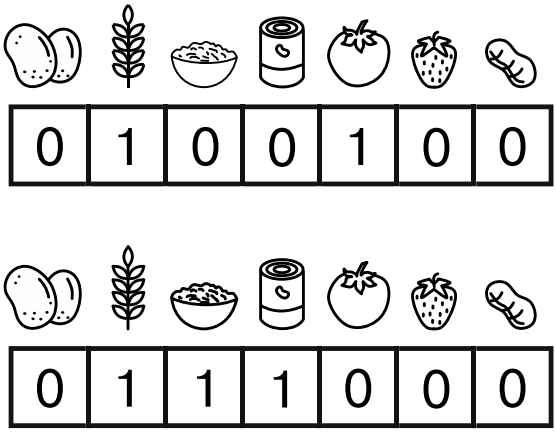


Рис. 18.4. Примеры кодирования хромосом для задачи о рюкзаке (конкретный пример описан в разделе 1.4). Различия в генотипах (две последовательности) отражаются в фенотипах (наборах продуктов, выбранных для доставки на Марс: пшеничная мука и помидоры против пшеничной муки, риса и бобов). Вторая хромосома представляет наилучшее решение для задачи, описанной в табл. 18.1

Стоит отметить, что в этом примере мы имеем отображение 1:1 между генотипом и фенотипом: решение (фенотип) однозначно идентифицируется своим представлением (генотипом), поэтому два организма с одним и тем же генотипом будут преобразованы в одно и то же решение и в одно и то же значение приспособленности (объяснение этого будет приведено в нескольких следующих разделах).

В контексте примера с рюкзаком 0-1 если два организма имеют одинаковый генотип, то оба решения приведут к добавлению в трюм корабля одного и того же набора продуктов.

Однако в природе, где разница между ними очевидна, генотип и фенотип не эквивалентны, потому что геном кодирует только правила развития, а не точные результаты, и, кроме того, организм определяется его взаимодействием с окружающей средой[1].

Точно так же в некоторых задачах имитации приспособленность не всегда полностью определяется генотипом. Одним из первых примеров применения генетических алгоритмов, который я изучал, было прикладное исследование[2] Флореано (Floreano) и Мондада (Mondada) из EPFL. В нем авторы разработали нейронные сети небольших двухколесных роботов, моделируя эволюцию собирателей, а затем и популяций хищников и жертв (своего рода игра в прятки).

В их эксперименте генотип представлял набор весов для нейронной сети робота, которые, кстати, также полностью определяли фенотип. Однако приспособленность каждого организма позже определялась его взаимодействием с окружающей средой и другими роботами.

Если продолжить развитие этого исходного эксперимента, разрешив роботам самообучаться (в настоящее время нетрудно представить применение стохастического или мини-пакетного обратного распространения для изменения весов нейронной сети на основе входных данных из окружающей среды), то их фенотипы уже не будут определяться полностью генотипом, по крайней мере не после первого обновления, потому что взаимодействие робота с окружающей средой в процессе самообучения сильно влияет на изменение упомянутых весов.

В отношении рис. 18.4 необходимо также прояснить еще одну деталь: фактическое представление хромосом зависит от конкретной задачи. В данном случае оно зависит от конкретного экземпляра рюкзака 0-1 (хотя можно и нужно построить общее представление для всех рюкзаков 0-1).

Чтобы создать код реализации главного метода генетического алгоритма, следует заняться проектированием организмов, например их моделированием с помощью класса, которому будет передана фактическая хромосома (и, как мы увидим, методы, модифицирующие ее) во время выполнения через аргументы (используя шаблон разработки «Стратегии») или во время компиляции путем наследования (шаблон разработки «Макет»).

ПРИМЕЧАНИЕ

Генетический алгоритм — это метаалгоритм, шаблон, который наполняется кодом, характерным для каждой конкретной задачи. Соответственно, одна часть кода является универсальной — структурный код, определяющий метод оптимизации, — а другая часть специфической для конкретной задачи. На этом этапе описания компонент генетических алгоритмов рассмотрим псевдокод шаблона. А поскольку в качестве примера используется задача о рюкзаке, также рассмотрим некоторый псевдокод с реализацией конкретных методов для алгоритма решения задачи о рюкзаке 0-1 с четко обозначенными фрагментами.

---

[1] Представьте, например, пару близнецов, у которых общая ДНК, но каждый из них — отдельное и уникальное существо.

[2] Например, Evolutionary neurocontrollers for autonomous mobile robots // Neural Networks, 1998. — Vol. 11, Issues 7–8, October — November. — P. 1461–1478.

В листинге 18.1 показана возможная реализация класса `Individual`, моделирующая каждый отдельный организм, эволюционирующий в популяции.

Листинг 18.1. Класс Individual

```
Класс Individual моделирует эволюцию
каждого организма в популяции
class Individual
  #type array
  chromosome                          Геном каждого индивидуума состоит из хромосомы
                                      (теоретически возможны и многохромосомные
                                      организмы, хотя и не так часто)
  function Individual(chromosome)     Предполагается, что хромосомы передаются
                                      в вызов конструктора в виде аргументов
                                      (тривиальная реализация не показана). Кроме
                                      того, в этом классе может быть статический
                                      метод-генератор
  function fitness()
                                      От каждого индивидуума можно получить информацию о его
                                      приспособленности (этой функции может потребоваться передать
                                      аргументы, если приспособленность зависит от окружающей среды)
```

18.1.3. Популяция

Наиболее очевидная разница между имитацией отжига и генетическими алгоритмами заключается в том, что вместо настройки одного решения генетические алгоритмы развивают популяцию особей.

В листинге 18.2 показана возможная реализация метода, инициализирующего популяцию, к которой будет применяться генетический алгоритм. Это тоже шаблонный метод, потому что ему должна передаваться конкретная функция инициализации отдельных хромосом (которая опять же зависит от конкретной решаемой задачи).

Листинг 18.2. Метод initPopulation

```
Метод initPopulation принимает размер создаваемой популяции
(предполагаемый или проверенный как достаточный) и функцию
инициализации отдельных хромосом. Возвращает вновь созданную популяцию
function initPopulation(size, chromosomeInitializer)
  population ← []    Инициализируем популяцию пустым списком
  for i in {1,…,size} do
    population.add(new Individual(chromosomeInitializer())
  return population
                                      Добавляем необходимое количество
                                      индивидуумов, каждый со своим геномом
```

Существует несколько разных стратегий инициализации. Если задача допускает это, то можно положиться на случайную инициализацию, чтобы обеспечить разнообразие в исходной популяции. Это оригинальный и до сих пор наиболее распространенный подход, но в некоторых ситуациях можно также добавить ограничения на хромосомы (чтобы избежать неверных решений) или даже передать готовую начальную популяцию (полученную, например, в результате применения другого алгоритма) или популяцию, полученную в предыдущей итерации.

В общем случае для задачи о рюкзаке 0-1 можно использовать генератор случайных чисел, как показано в листинге 18.3, потому что хромосомы — это просто последовательности битов.

Листинг 18.3. Задача о рюкзаке 0-1: генератор хромосом

Метод knapsackChromosomeInitializer принимает размер создаваемой хромосомы (предполагаемый или проверенный как достаточный) и возвращает случайную последовательность битов (или в данном случае массив битов)

```
function knapsackChromosomeInitializer(genesNumber)
  chromosome ← []
  for i in {0,…,genesNumber-1} do
    chromosome[i] ← randomBool().toInt()
  return chromosome
```

Генерируем необходимое количество случайных битов. Здесь применяется Java-подобный шаблон, но также можно использовать метод, возвращающий случайные целые числа от 0 до 1

Инициализируем хромосому пустым списком

Еще одно важное замечание: не все случайно сгенерированные последовательности являются приемлемыми решениями для данного экземпляра задачи о рюкзаке. Например, последовательность со всеми единицами означает (для нетривиального примера) нарушение ограничения по весу. Как увидим далее, решениям, нарушающим это ограничение, можно присвоить низкую приспособленность; другой вариант: генерируя хромосомы, можно проверять, чтобы они не нарушали это ограничение.

18.1.4. Приспособленность

В тесной связи с определениями хромосомы и организма находится понятие *приспособленности* (fitness) индивидуума. Как можно заметить в листинге 18.1, класс `Individual` имеет метод `fitness`, он возвращает значение, измеряющее приспособленность организма к окружающей среде.

Термин *«приспособленность»*, естественно, подразумевает задачу максимизации, потому что чем выше приспособленность, тем лучше чувствует себя организм в окружающей среде. Однако для многих задач предпочтительнее выражать цели в виде *функций стоимости*, которые должны быть минимизированы, поэтому у нас на выбор есть два варианта: либо реализовать шаблон генетического алгоритма так, чтобы более низкое значение приспособленности означало более высокое качество, либо переформулировать функцию стоимости в соответствии со сделанным выбором. Например, если нужно найти точку наибольшей высоты на карте, можно либо реализовать алгоритм максимизации, либо установить функцию приспособленности как `f(P)=-elevation(P)` (где `elevation()` — функция, возвращающая высоту в заданной точке) и придерживаться обычной стратегии минимизации (как было показано в главах 16 и 17, где обсуждались градиентный спуск и имитации отжига).

Вернемся к примеру с задачей о рюкзаке 0-1: цель естественным образом выражается функцией энергетической ценности, которую требуется максимизировать, — суммой энергетической ценности продуктов, добавленных в трюм.

Первая хромосома на рис. 18.4, например, имеет вид `0100100` и означает, что необходимо сложить энергетическую ценность риса и помидоров, что дает в результате 1 359 000 калорий. Но что произойдет, если будет превышена вместимость трюма, например, с хромосомой `1111111`? Для обнаружения таких пограничных случаев нужно проверить общий вес при вычислении приспособленности и присвоить особое значение (например, `0`, как в данном случае) решениям, нарушающим ограничение.

В качестве альтернативы можно убедиться в невозможности такой ситуации, проверяя ограничения в операторах, создающих и изменяющих индивидуумов (инициализация, скрещивание и мутация, с которыми мы вскоре познакомимся). Это решение, однако, может сказаться на процессе оптимизации, искусственно уменьшая множество признаков в совокупности.

Следующий важный момент. Если шаблонный метод, который мы внедряем в генетический алгоритм, пытается максимизировать приспособленность индивидуумов, то все в порядке. Но что, если наша реализация будет стремиться к более низкой приспособленности? Для этого конкретного случая можно предложить простое решение: суммировать ценность отброшенных продуктов, соответствующих нулям в хромосоме. Для любого заданного решения чем меньше эта сумма, тем выше будет сумма калорий продуктов, загруженных в трюм. Конечно, если пойти по этому пути, то решениям, превышающим ограничение по весу, придется присваивать очень высокое значение (может быть, даже бесконечность).

18.1.5. Естественный отбор

Теперь у нас есть все необходимое для создания одного организма, целой популяции организмов и проверки индивидуальной приспособленности. А это, в свою очередь, позволяет имитировать основной естественный процесс, управляющий эволюцией популяций в дикой природе: *естественный отбор*.

Естественный отбор можно видеть в природе повсюду: самые сильные и быстрые особи, которые лучше всего умеют охотиться или прятаться, как правило, живут дольше, и, в свою очередь (или вдобавок к этому), у них больше шансов спариться и передать свои гены следующему поколению.

Цель генетических алгоритмов состоит в том, чтобы использовать аналогичный механизм для получения хороших решений. В этой аналогии гены индивидуумов с высокой приспособленностью[1] кодируют черты, дающие им преимущество перед конкурентами.

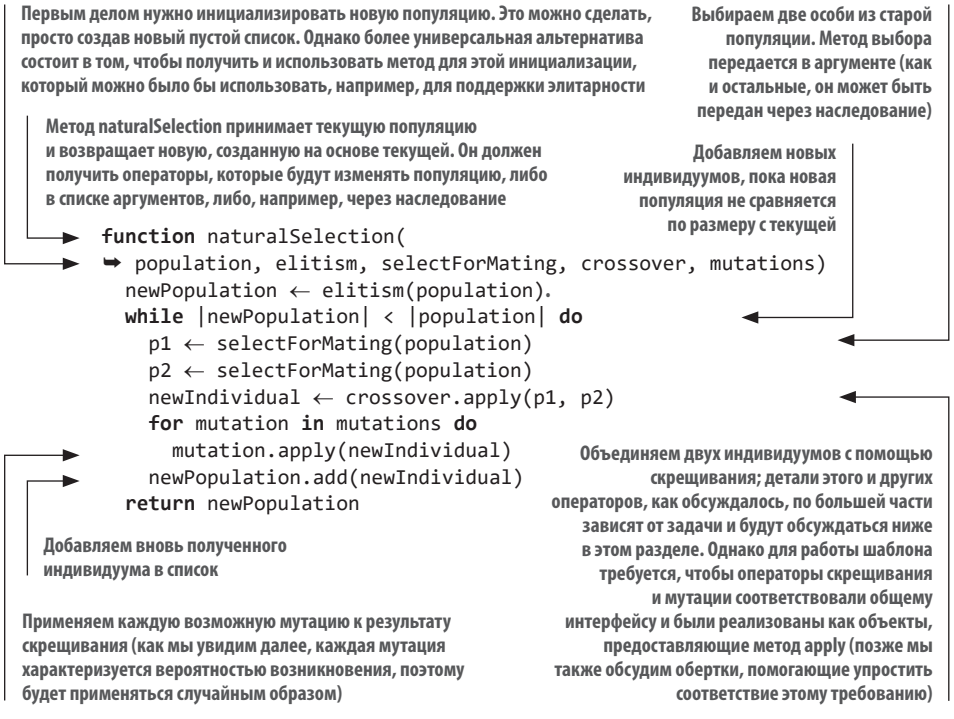
В примере с рюкзаком хорошей чертой может быть включение продуктов с высоким соотношением калорий к весу. Решения, включающие картофель, будут

[1] В оставшейся части главы будем использовать понятие высокой приспособленности обобщенно, отделяя его от фактической реализации. Оно будет означать большие значения для задач, которые максимизируют функцию, и малые значения, когда целью оптимизации является минимизация.

индивидуумами с низкой приспособленностью, и поэтому им будет трудно передать свои гены следующим поколениям. Например, в модели «хищник — жертва», упомянутой в подразделе 18.1.2, жертва, освоившая способность прятаться за объектами, эффективнее убегала от хищников.

В листинге 18.4 на примере псевдокода показано, как работает естественный отбор (в генетических алгоритмах). Этот метод регулирует переход популяции от текущего поколения к следующему, принимая старую популяцию и возвращая новую, случайно созданную из исходных индивидуумов с помощью набора операторов.

Листинг 18.4. Естественный отбор



Этот процесс естественного отбора применяется на каждой итерации алгоритма. Когда алгоритм запускается, он всегда получает полностью сформированную популяцию, возвращаемую методом инициализации (ее размер, как мы видели, определяется во время выполнения с помощью значения, получаемого в аргументе). Индивидуумы во входной популяции оцениваются на предмет их приспособленности, а затем проходят процесс отбора, который определяет, какие из них просто перейдут в следующее поколение без изменений, а какие передадут свои гены посредством спаривания.

В простейшей форме генетического алгоритма новая популяция инициализируется пустым списком (строка 2), который затем *заполняется* новыми индивидуумами, полученными в результате отбора и спаривания (строки 3–6).

Что касается спаривания, то здесь есть отличия от биологической аналогии. Прежде всего, в генетических алгоритмах обычно используется половое размножение между двумя *бесполыми* организмами[1]. Более того, для рекомбинации генетического материала двух родителей не используются никакие биологические правила, а реальные детали скрещивания по большей части зависят от конкретной задачи.

После скрещивания в генетический материал добавляются мутации, то есть скрещивание и мутирование реализуются как отдельные фазы.

Рисунок 18.5 иллюстрирует работу механизма, опираясь на аналогии с биологическим процессом.

В начальный момент индивидуумы в популяции имеют общую основу и некоторые специфические черты. Разнообразие и изменчивость популяции в значительной степени зависит от того, на какой стадии эволюции она находится. Вы увидите далее, что наша цель — получить как можно более разнообразную популяцию в начале процесса, а затем свести популяцию к нескольким однородным высокоприспособленным группам.

В примере популяция уток демонстрирует несколько уникальных особенностей, от формы клюва до окраски и формы крыльев и хвостов. Если присмотреться, то можно заметить эти особенности и увидеть, как они закодированы в хромосомах, потому что, как говорилось выше, каждая особенность фенотипа индивидуума определяется разницей в генотипе, когда единица становится нулем или наоборот (может быть, меняются сразу несколько битов, или даже, если отойти от двоичных последовательностей, гену будет присвоено другое действительное число).

Чтобы выполнить отбор, нужно оценить приспособленность каждой утки. В одних случаях для этого может быть достаточно просто применить функцию к каждой хромосоме, а в других могут потребоваться многочасовое моделирование (как в описанной выше ситуации «хищник — жертва»), чтобы увидеть, какие особи выживают дольше или лучше справляются с реальной задачей.

О выборе индивидуумов для спаривания мы поговорим в следующем подразделе, но уже сейчас можно с уверенностью сказать, что, независимо от деталей механизма отбора, наибольшие шансы быть выбранными для передачи своих генов следующему поколению будут иметь организмы с более высокой приспособленностью. Это очень важный, вероятно, единственный по-настоящему важный аспект, потому что без этой «меритократии» никакая оптимизация не возможна.

---

[1] В свое время предпринимались попытки добавить понятие половых хромосом и половых подгрупп, но, насколько мне известно, возможные улучшения в эффективности или действенности алгоритма, достигнутые таким образом, не превышают величины оптимизации. В качестве примера можно привести публикацию: *Zhang M.-m., Zhao S.-g., Wang X.* Sexual Reproduction Adaptive Genetic Algorithm Based on Baldwin Effect and Simulation Study [J] // Journal of System Simulation, 2010. — № 10.

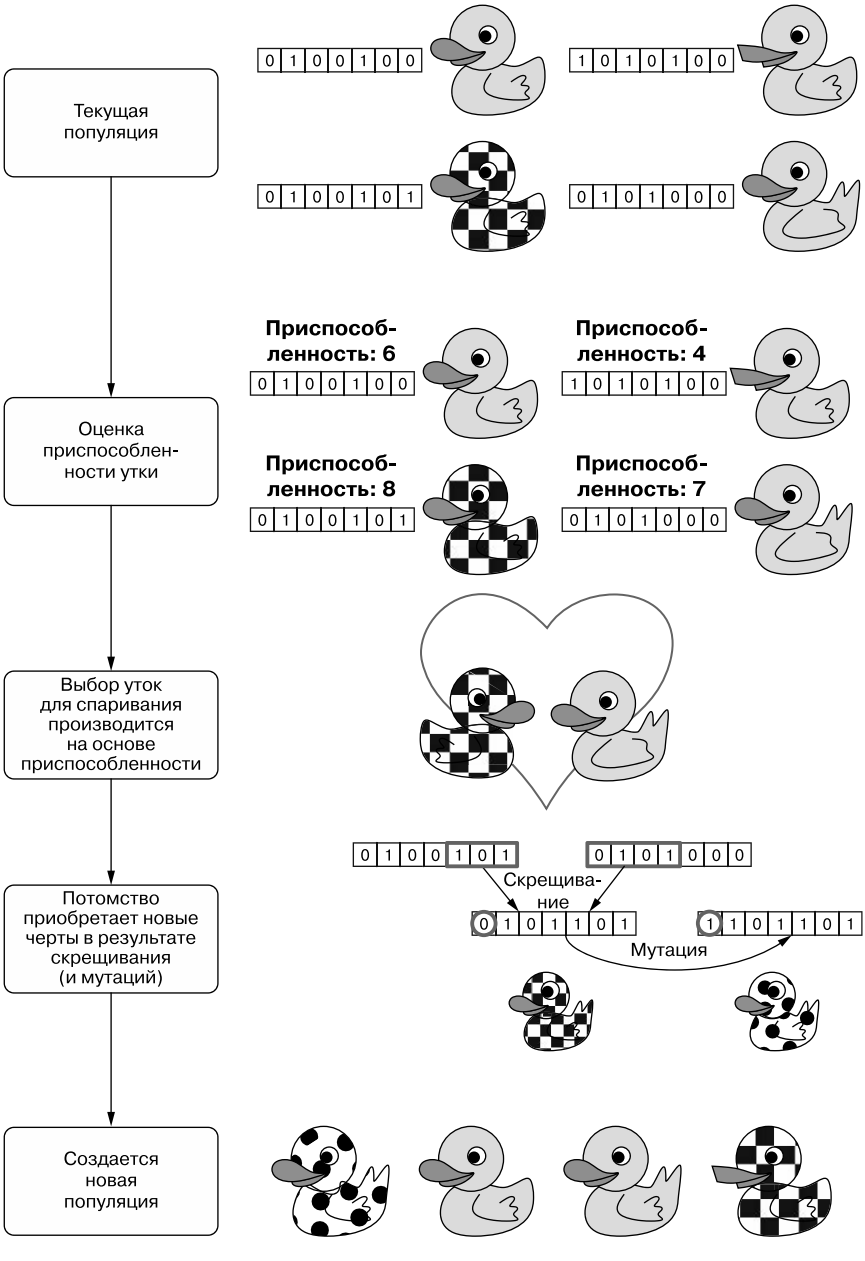


Рис. 18.5. Естественный отбор в генетических алгоритмах действует по аналогии с биологическим процессом

После выбора двух особей нужно рекомбинировать их генетический материал. Скрещивание и мутации мы тоже рассмотрим в одном из разделов ниже, но, как уже

упоминалось, эти операторы, действующие на хромосомы, во многом определяются конкретной решаемой задачей.

В примере на рис. 18.5 наш маленький утенок получает окраску оперения от одного родителя, а форму хвоста — от другого. Затем происходит одна мутация, изменяющая окраску с шахматным узором на узор в горошек.

Новая популяция, полученная в результате многократного повторения этого процесса, должна демонстрировать присутствие большего количества генов/признаков, унаследованных от высокоприспособленных индивидуумов в исходной популяции.

Теперь, ознакомившись с работой обобщенного процесса естественного отбора, можно заняться особенностями отдельных этапов.

18.1.6. Отбор индивидуумов для спаривания

Начнем с выбора индивидуумов. Чтобы лучше его объяснить, вернемся к задаче о рюкзаке.

Приведу несколько методов, которые можно использовать для выбора в каждой итерации индивидуумов, которые будут спариваться или прогрессировать:

- элитарность;
- пороговое значение;
- турнирный отбор;
- рулетка.

Конечно, кроме этих, есть множество других методов, широко используемых в практике.

Первые два из приведенного ограниченного списка — это фильтры для старой популяции, а два последних — собственно методы отбора.

Элитарность и пороговое значение, показанные на рис. 18.6, помогают решить, какие организмы могут или не могут передать свои гены следующему поколению.

Элитарность позволяет лучшим индивидуумам переходить в следующее поколение в неизменном виде. Применять этот принцип или нет — решать разработчику алгоритма. Как обычно, в одних задачах он может проявлять себя лучше, в других хуже.

Без элитарности все организмы текущей итерации были бы заменены в следующей и «прожили бы» ровно одно поколение, а с помощью этого обходного пути можно смоделировать более длительную продолжительность жизни для особенно хорошо приспособленных индивидуумов — чем выше их приспособленность по сравнению со средней по популяции, тем дольше они будут сохраняться.

Это также гарантирует, что гены *альфа-организмов* сохранятся нетронутыми и в следующем поколении (хотя может оказаться так, что в следующем поколении они уже не будут наиболее приспособленными индивидуумами).

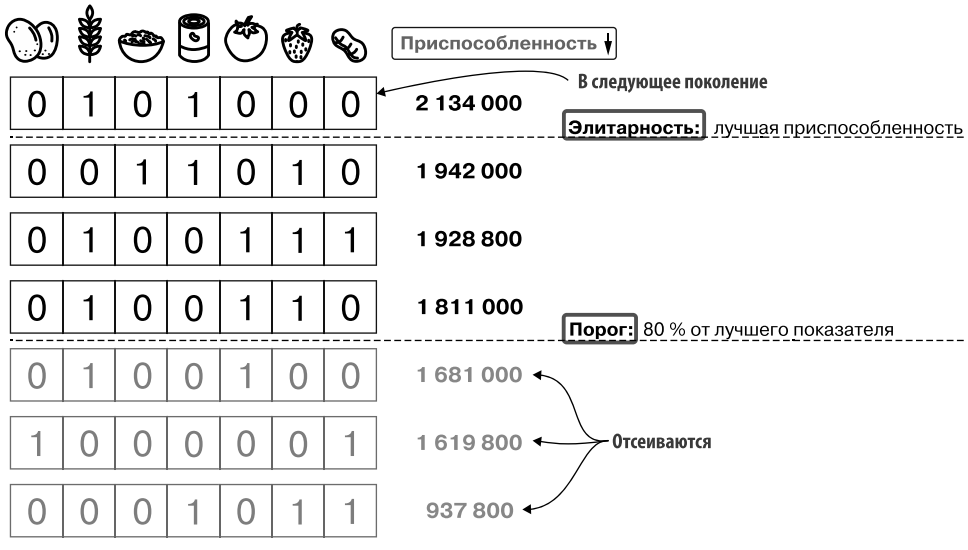


Рис. 18.6. Элитарность и порог на примере задачи о рюкзаке 0-1. Элитарность предполагает отбор лучших индивидуумов (одного или нескольких наиболее приспособленных) для переноса их в следующее поколение без спаривания и/или мутаций. Порог решает противоположную проблему: это жесткий барьер, не позволяющий индивидуумам с низкой приспособленностью передать свои гены следующему поколению. Он предполагает отсеивание определенного фиксированного количества или всех индивидуумов с приспособленностью ниже некоторого порогового значения. Например, в этом примере (где максимизируется значение приспособленности) все особи со значением приспособленности ниже 80 % от значения приспособленности лучшей особи будут отброшены и никогда не будут выбраны для спаривания

Одним важным эффектом использования элитарности является монотонное улучшение приспособленности лучшего представителя в популяции из поколения в поколение (что, однако, может не относиться к средней приспособленности).

Порог имеет противоположную цель: он предотвращает передачу в следующее поколение генов организмами с самой низкой приспособленностью, что, вполне логично, уменьшает вероятность исследования алгоритмом менее перспективных областей в ландшафте функции приспособленности.

Принцип работы прост. Устанавливаем порог приспособленности индивидуумов, которых разрешено выбирать для спаривания, и игнорируем тех, которые оказались ниже этого порога. Можем отбросить фиксированное количество организмов, например пять или переменное количество, с худшими значениями приспособленности.

В последнем сценарии порог приспособленности обычно устанавливается динамически (изменяется в каждом поколении) на основе значения приспособленности лучшего индивидуума в этом поколении. При работе со статическим абсолютным значением (что определенно требует хорошего знания предметной области) есть риск,

что ограничение окажется неэффективным (приведет к сохранению всей популяции) или, что еще хуже, к фатальным последствиям, когда из-за слишком строгого порога на ранних стадиях отфильтруются почти все индивидуумы.

В примере на рис. 18.6 было предложено установить пороговое значение равным 80 % от значения приспособленности лучшего индивидуума[1]. Решение о применении порога и выбор его значения полностью зависят от конкретной решаемой задачи.

После фильтрации начальной популяции и, возможно, переноса лучших особей в следующее поколение приходит очередь задачи воссоздания оставшихся особей в новой популяции. Для этого используется оператор *скрещивания*, который будет обсуждаться в следующем подразделе (он создает новый организм из двух родителей), и оператор *мутации* (он обеспечивает генетическое разнообразие в новых поколениях). Таким образом, первым шагом перед скрещиванием является выбор двух родителей. Как показано в листинге 18.4, этот выбор выполняется несколько раз в каждой итерации.

Существует множество способов отбора индивидуумов для спаривания. Обсудим два наиболее распространенных.

Турнирный отбор — один из самых простых (и легко реализуемых) методов отбора, используемых в генетических алгоритмах, как показано на рис. 18.7. Суть его состоит в том, чтобы случайным образом отобрать несколько организмов, а затем заставить их «состязаться» в турнире, где только победитель получает право на спаривание. Подобное можно наблюдать повсюду в дикой природе, где (обычно) взрослые самцы состязаются за право спариться с самками (возможно, вам приходилось видеть документальные фильмы об оленях, устраивающих дуэли во время брачного сезона).

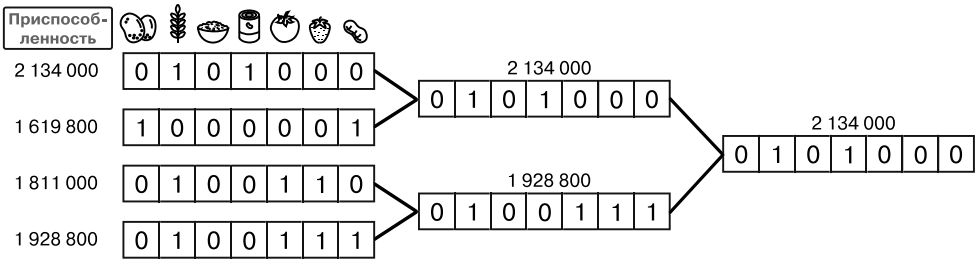


Рис. 18.7. Турнирный отбор на примере задачи о рюкзаке 0-1

На самом деле нет нужды реализовать турнирный отбор, если только вы не собираетесь имитировать состязания![2] Как показано в листинге 18.5, мы просто отбираем

---

[1] Потому что мы пытаемся максимизировать функцию приспособленности для задачи о рюкзаке 0-1; в противном случае, когда функция должна минимизироваться, можно было бы установить порог равным 120 или 105 %.

[2] Например, для роботов в модели «хищник — жертва» или любого другого сценария эволюции системы в физическом мире. В таких случаях можно добавить такую имитацию и оценивать индивидуумы на основе того, как они решают реальные задачи.

k случайных индивидуумов и из них выбираем одного с наилучшей приспособленностью. Точное число k может меняться в зависимости от задачи и размера популяции, но обычно это значение лежит в диапазоне от 3 до 5. Учтите, что чем большее количество индивидуумов участвует в турнирах, тем меньше шансов, что будут выбраны индивидуумы с низкой приспособленностью.

Листинг 18.5. Турнирный отбор

Повторяем k раз

Метод tournamentSelection принимает текущую популяцию и количество индивидуумов, которые должны участвовать в каждом турнире. Возвращает индивидуума, лучшего среди выбранных для турнира

Инициируем временную переменную с максимальной оценкой приспособленности. Поскольку это обобщенный метод и неизвестно, будет ли в конкретных задачах лучшая приспособленность иметь самое высокое или самое низкое значение, используем вспомогательную функцию, возвращающую «самую низкую» возможную приспособленность, 0 для максимизируемых функций и бесконечность для минимизируемых

```
function tournamentSelection(population, k)
  bestFitness ← lowestPossibleFitness()
  for i in {1,..,k} do
    j ← randomInt(|population|)
    if isHigher(population[j].fitness, bestFitness) then
      bestElement ← population[j]
      bestFitness ← bestElement.fitness
  return bestElement
```

Проверяем, лучше ли этот индивидуум текущего лучшего; и снова здесь используется вспомогательная функция, чтобы абстрагироваться от значения хорошей/высокой приспособленности

Выбираем случайный индекс между 0 и размером популяции. Косвенно это действие выбирает индивидуума. В этой реализации не предпринимаются меры для предотвращения появления дубликатов, что может быть допустимо, когда размер популяции велик (и, следовательно, вероятность дублирования мала), и все же стоит помнить о такой возможности

Если встречен лучший индивидуум, чем текущий, то просто обновляем временные переменные

Например, для выбора третьего лучшего индивидуума нельзя выбрать ни лучшего, ни второго по приспособленности индивидуума (и, конечно, нужно иметь третьего лучшего), поэтому вероятность, что это произойдет с пулом, насчитывающим k индивидуумов, равна[1]:

$$1 / n \times [(n - 2) / \mathrm{n}]^{k-1},$$

где n — размер популяции.

Для индивидуума с самой низкой приспособленностью (возможно, после применения порогового значения) она равна:

$$1 / n \times [1 / n]^{k-1} = 1 / n^k.$$

[1] Приблизительно и с некоторыми упрощениями: фактическое значение зависит от деталей реализации, например от возможности выбора одного и того же индивидуума несколько раз. Это хорошо, потому что нас на самом деле интересуют не точные значения вероятностей, а понимание порядков их величин и получение представления о том, как они меняются с изменением k.

Общая формула вероятности выбора m-го лучшего индивидуума имеет вид:

$$1 / n \times [(n - m + 1) / n]^{k-1}.$$

Как видите, помимо деталей и фактической точной вероятности, шансы любого индивидуума (кроме первого) уменьшаются экспоненциально с увеличением k (и полиномиально с m).

Очевидно, что турнирный отбор нужно применить дважды, чтобы получить пару родителей для создания одного индивидуума в новой популяции.

Отбор рулеткой, безусловно, сложнее турнирного отбора, но общая идея та же: шансов быть выбранными у индивидуумов с более высокой приспособленностью должно быть больше.

Как мы уже видели, при турнирном отборе вероятность отбора индивидуума с низкой приспособленностью полиномиально уменьшается с его рангом (положением в списке организмов, отсортированных по приспособленности); в частности, поскольку вероятность будет равна $O([(n - m) / n]^k)$, уменьшение будет суперлинейным, потому что k заведомо больше *1*.

Если бы, напротив, требовалось, чтобы менее приспособленные индивидуумы имели реальный шанс быть выбранными, можно было бы прибегнуть к более справедливому методу отбора. Один из вариантов — присвоить одинаковую вероятность всем индивидуумам, но тогда индивидуумы с более высокой приспособленностью не получат никаких преимуществ[1]. Хорошим балансом было бы, например, убедиться, что вероятность выбора каждого индивидуума пропорциональна его приспособленности.

Если выбрать последнее, то можно использовать отбор рулеткой. Каждому организму назначается участок на «колесе рулетки», угол которого (и, в свою очередь, длина стягиваемой дуги) пропорционален приспособленности организма.

Один из способов реализовать это — вычислить сумму приспособленностей всех индивидуумов, а затем для каждого получить процент совокупной приспособленности всей популяции, которую он составляет[2]. Например, на рис. 18.8 показано, как применить этот способ к популяции в задаче о рюкзаке. Угол сектора для каждого организма вычисляется по формуле $\theta = 2\varpi \times f$, где f — нормализованная приспособленность данного индивидуума, то есть доля приспособленности индивидуума в общей сумме для всей популяции.

---

[1] В некоторых задачах агрессивное использование элитарности может компенсировать чистую случайность отбора.

[2] Конечно, это решение пригодно, только когда более высокая приспособленность определяется более высоким значением. Его можно адаптировать к другому случаю, используя, например, инверсию значений приспособленности.

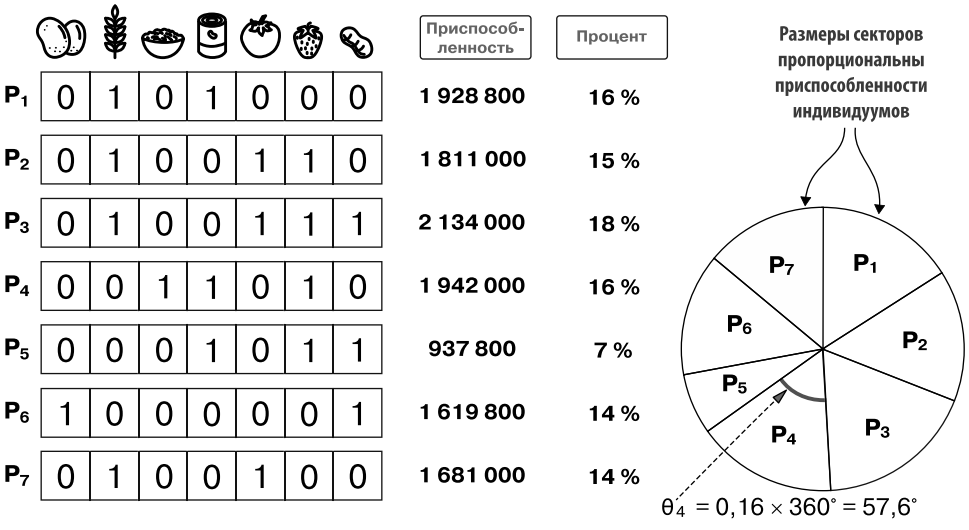


Рис. 18.8. Колесо рулетки применительно к задаче о рюкзаке 0-1, обсуждаемой в этом разделе

Каждый раз, когда нужно выбрать нового родителя для спаривания, вращаем колесо и смотрим, где оно остановится: более высокая приспособленность имеет бóльшую вероятность быть выбранной, но и индивидуумы с более низкой приспособленностью все еще имеют шанс.

ПРИМЕЧАНИЕ

Обратите внимание, что, если использовать ранг вместо приспособленности, нужно сначала отсортировать всю совокупность особей (требуемое время: O(n log(n))). При использовании приспособленности достаточно линейного количества операций для вычисления суммы и процентов, а поскольку вычисления выполняются в каждой итерации, используя ее для больших популяций, можно получить существенную экономию. Кстати, турнирный отбор тоже не требует предварительной сортировки популяции.

Однако в реальной реализации этого приема не требуется явно конструировать колесо (и даже его модель)!

Тот же результат можно получить, построив массив, подобный изображенному на рис. 18.9, где *i*-й элемент содержит сумму нормализованных[1] приспособленностей первых *i* индивидуумов в текущей популяции (взятых в том порядке, в котором они хранятся в массиве с популяцией).

[1] Приспособленность каждого индивидуума нормализуется делением ее значения на общую сумму приспособленностей в популяции. В результате каждая нормализованная приспособленность получит значение в диапазоне от 0 до 1, а их сумма для всей популяции равна 1.

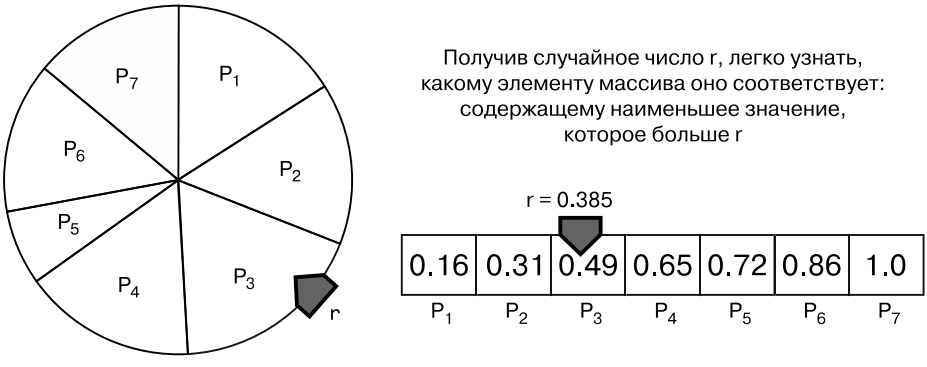


Рис. 18.9. Отбор методом рулетки на практике. Роль колеса рулетки играет массив с накопленными процентами; разность между A[i] и A[i – 1] в точности равна отношению приспособленности i-го индивидуума к сумме приспособленностей всех индивидуумов

Например, для популяции, изображенной на рис. 18.8, первый элемент массива-колеса хранит значение `population[0].normalizedFitness = 0.16`, второй элемент `population[0].normalizedFitness + population[1].normalizedFitness = 0.16 + 0.15` и т. д., как показано на рис. 18.9. Обратите внимание, что последний элемент хранит общую сумму, точно равную 1. При желании можно представить, что в массиве есть скрытый первый элемент, не показанный на рис. 18.9, значение которого равно 0 (мы увидим, что такое представление помогает написать более понятный и лаконичный код).

Чтобы выбрать элемент массива, из диапазона действительных чисел от 0 до 1 (не включая последнее) извлекается случайное число r или, если следовать аналогии с колесом, случайный угол θ в диапазоне от 0 до 360° (не включая последнее) и определяется, на какой угол нужно повернуть колесо. Числа r и θ связаны отношением $\theta = r \times 360$.

В листинге 18.6 показан метод создания массива-колеса, подобного изображенному на рис. 18.9.

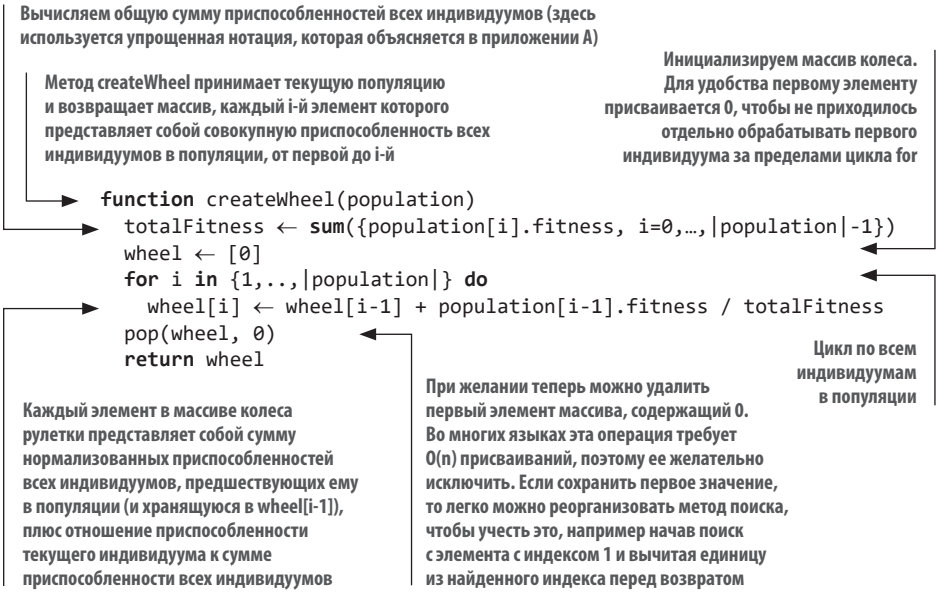
Чтобы определить сектор, на который указывает стрелка, нужно отыскать в массиве наименьший элемент, превышающий r — случайное число. Для решения этой задачи можно применить алгоритм двоичного поиска и сократить время выбора одного элемента до $O(\log(n))$. Поэтому в каждой итерации придется выполнить $O(n \times \log(n))$ операций, чтобы выбрать родителей и применить оператор скрещивания.

Можно также использовать линейный поиск. Его проще реализовать, и меньше вероятность, что он будет содержать ошибки, но это увеличит время выполнения до $O(n)$ и $O(n^2)$ соответственно.

Выбирайте тот или иной вариант, учитывая ограничения по времени и опираясь на знания в предметной области. Я оставляю реализацию выбранного метода поиска вам на проработку в качестве самостоятельного упражнения и настоятельно рекомендую

начать с самого простого варианта, тщательно протестировать его, а затем, если вам действительно понадобится ускорение, попытаться написать метод двоичного поиска и сравнить полученные результаты.

Листинг 18.6. Создание колеса рулетки для отбора родителя



18.1.7. Скрещивание

После выбора пары родителей можно сделать следующий шаг: скрестить их, как показано в листинге 18.4.

Выше уже упоминалось об этом, но на всякий случай кратко напомню, что скрещивание имитирует спаривание и половое размножение животных[1] в природе и способствует развитию разнообразия в потомстве, позволяя рекомбинировать подмножества признаков обоих родителей в каждом потомке.

В аналогичном алгоритмическом процессе скрещивание соответствует поиску в ландшафте функции приспособленности, потому что обычно рекомбинируются большие участки генома (и, следовательно, решения), которые несет каждый родитель. Это эквивалентно крупному скачку в пространстве задачи (и в ландшафте функции стоимости).

Например, возьмем решаемую сейчас задачу: рюкзак 0-1 и выбор продуктов для полета в космос. На рис. 18.10 показано возможное определение метода скрещивания для этой задачи: он выбирает единственную точку скрещивания — индекс, по которому

[1] Некоторые цветковые растения тоже размножаются половым путем посредством опыления.

рассекаются обе хромосомы; затем одна часть каждой хромосомы (на противоположных сторонах от точки скрещивания) используется для рекомбинации. В данном случае две части просто «склеиваются».

Случайность есть и должна быть важной частью скрещивания. В рассматриваемом примере она присутствует на этапе выбора точки скрещивания. Некоторые варианты выбора (как на рис. 18.10, *А*) приведут к улучшению приспособленности потомка, а некоторые (как на рис. 18.10, *Б*) — к ухудшению и даже к катастрофическим последствиям.

На самом деле, если вы заметили, метод с одной точкой скрещивания создает две пары противоположных подпоследовательностей (левая-правая и правая-левая от точки пересечения). В нашем примере используется только одна из пар, а вторая отбрасывается, однако некоторые реализации генетического алгоритма используют обе пары и создают двух потомков со всеми возможными комбинациями[1] (либо оставляя их обоих, либо оставляя того, который лучше приспособлен).

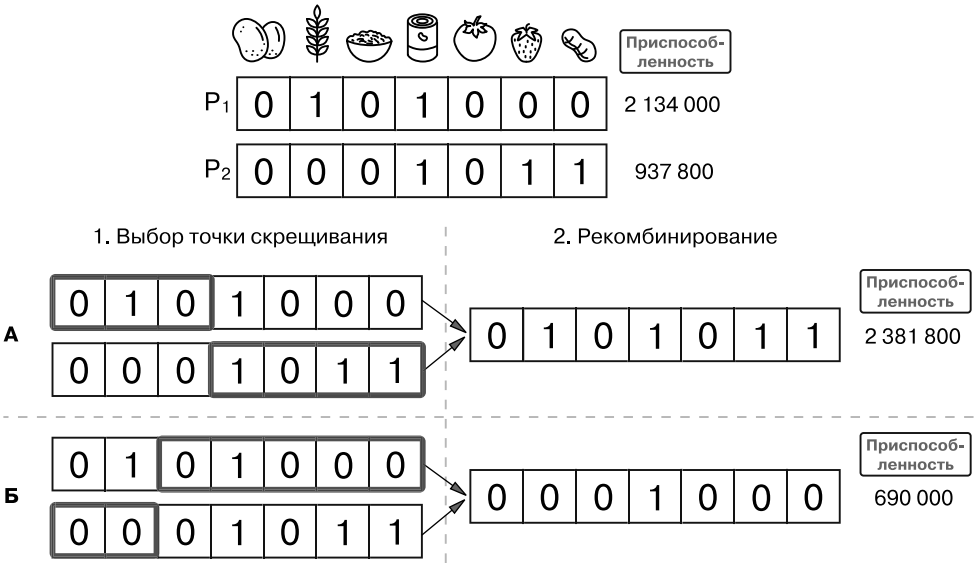


Рис. 18.10. Примеры скрещивания для задачи о рюкзаке. После отбора двух родителей выбирается точка скрещивания, а затем два генома рекомбинируются. В зависимости от проведенного выбора потомок может получиться лучше *(А)* или хуже *(Б)* приспособленным

Но никто не говорит, что необходимо придерживаться одноточечного скрещивания; это всего лишь один из многих вариантов. Например, скрещивание можно производить по двум точкам и выбирать один сегмент из первой хромосомы (и неявно — один или

[1] Этот процесс концептуально похож на ранние фазы мейоза, механизма, используемого клетками для полового размножения.

два оставшихся сегмента из другой) или вообще случайно выбирать каждое значение из той или другой родительской хромосомы. Оба примера показаны на рис. 18.11.

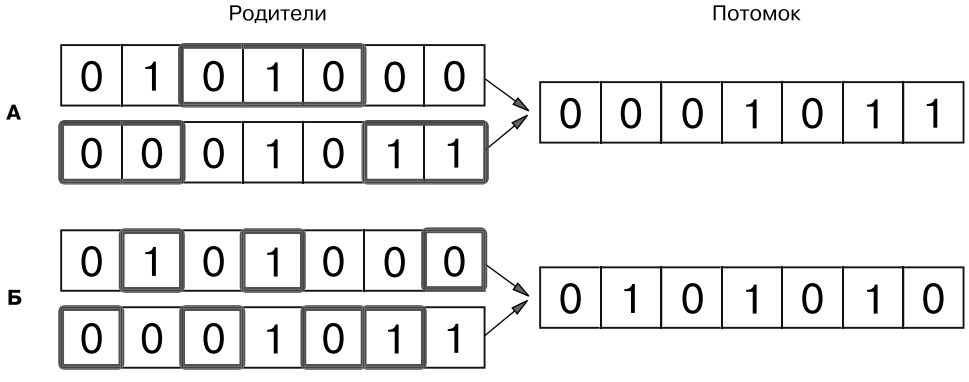


Рис. 18.11. Еще примеры скрещивания для задачи о рюкзаке. *А*. Двухточечная рекомбинация: из первой хромосомы выбирается один сегмент, а остальные — из другой. *Б*. Для каждого гена (также известного как значение) случайным образом решается, из какого родителя он будет скопирован

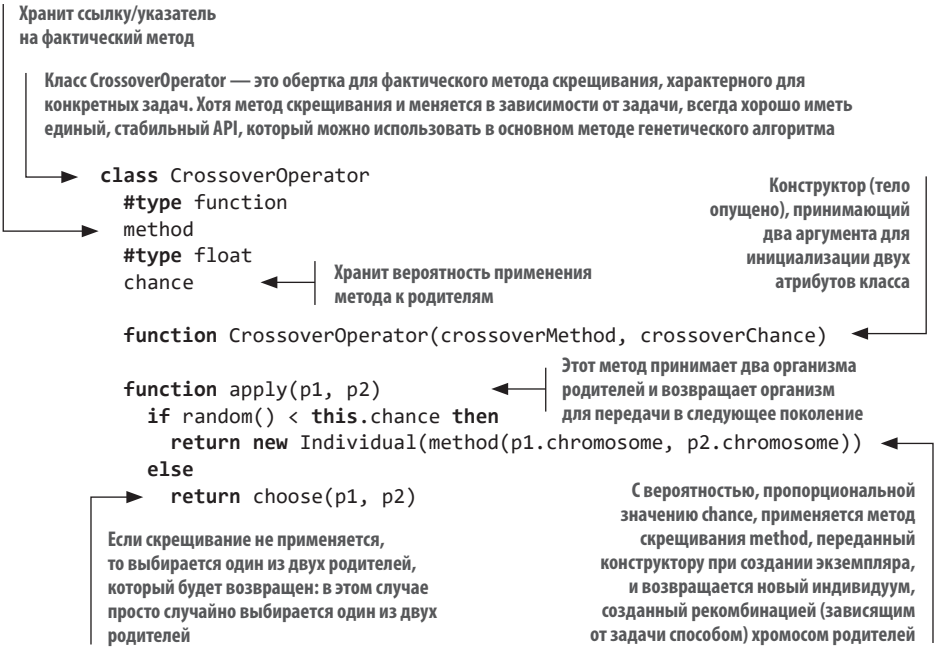
Возможны и многие другие альтернативы: как уже говорилось, скрещивание можно определить только на реальных хромосомах, и в зависимости от их структуры и ограничений (и в конечном итоге от структуры решения-кандидата) возможны различные способы рекомбинации хромосом. Ниже в этой главе нам с вами еще предстоит рассмотреть множество примеров.

И последнее: шанс скрещивания, также известный как *коэффициент скрещивания*, принято связывать с операторами скрещивания. По сути, это вероятность, что скрещивание/спаривание действительно произойдет между выбранными индивидуумами. Например, при коэффициенте скрещивания 0,7 существует 70%-ная вероятность, что скрещивание произойдет с любой парой организмов, выбранных для спаривания.

Как показано в листинге 18.7, каждый раз, когда применяется скрещивание, выбор предпринимаемого действия основывается на вероятности скрещивания. Если выбирается неприменение скрещивания, то рассматривается несколько альтернатив. Чаще всего случайным образом выбирается один из родителей, который переходит на следующий этап. Обратите внимание, что это действие отличается от рассмотрения элитарности, потому что организм, полученный в результате скрещивания, все равно будет подвергнут мутации, в то время как любой индивидуум, выбранный по элитарности, просто копируется в новую популяцию без каких-либо изменений.

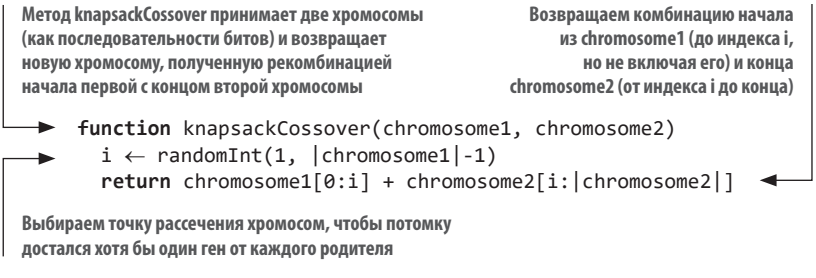
Напомню, что в листинге 18.7 показана обертка для фактического оператора скрещивания, совместимая с универсальным шаблоном естественного отбора, представленным в листинге 18.4. Фактический метод будет определяться и передаваться каждый раз при обращении к конкретному экземпляру задачи.

Листинг 18.7. Класс-обертка для оператора скрещивания



В листинге 18.8 показан оператор одноточечного скрещивания, упоминавшегося в обсуждении задачи о рюкзаке 0-1.

Листинг 18.8. Оператор скрещивания для задачи о рюкзаке 0-1



18.1.8. Мутации

После создания нового организма путем скрещивания генетический алгоритм применяет мутации к его геному. В природе, как мы знаем, рекомбинация родительских геномов и мутации происходят одновременно[1] в ходе полового размножения.

[1] Мутации также происходят спонтанно с околонулевой вероятностью во время митоза, механизма бесполого размножения всех клеток (гамет и соматических клеток). Когда эти мутации происходят в гаметах — клетках, участвующих в половом размножении, они могут играть важную роль в эволюции организмов и видов.

В генетических алгоритмах мутации работают аналогично, даже притом что это два отдельных оператора, выполняемых независимо. Если скрещивание можно сравнить с поиском в широком диапазоне, то мутации — это аналог локального поиска.

Мы видели, что скрещивание и так способствует разнообразию в новой популяции, так зачем нам нужны мутации?

Ответ прост, и лучше всего проиллюстрировать его примером (рис. 18.12). Популяция, рассматриваемая в нем, имеет одинаковые значения в трех генах. Это означает, что, как бы ни рекомбинировались гены организмов на этапе скрещивания, все получившиеся потомки будут включать в решение фасоль и клубничное варенье и ни один не будет включать картофель. Возникает реальный риск появления проблем, когда хромосомы большие (несущие много информации), особенно если длина хромосомы сравнима с размером популяции. Опасность заключается в том, что, независимо от продолжительности симуляции или количества итераций и скрещиваний, некоторые гены никогда не смогут поменять свое значение. А это, в свою очередь, означает, что пространство задачи, доступное для исследования, будет ограничено выбором начальной популяции. Ужасное ограничение.

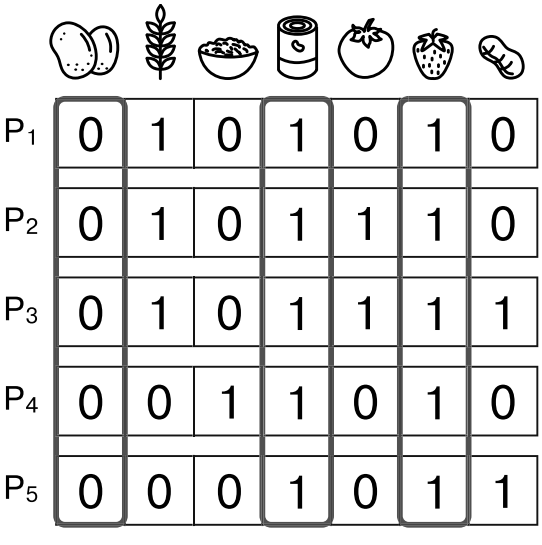


Рис. 18.12. Пример популяции, в которой без мутаций невозможно достаточное генетическое разнообразие. Все индивидуумы имеют одинаковое значение в трех генах

Чтобы предотвратить проблему и увеличить разнообразие геномного пула популяции, добавим механизм, способный изменять любые гены и в любом организме независимо.

В оригинальной работе Холланда, как уже упоминалось, хромосомы были представлены последовательностями битов и мутации считались не зависящими от предметной области. Оператор мутации будет применяться к каждой хромосоме организма,

и для каждого гена (то есть каждого бита) будет подбрасываться монета[1] и решаться, следует ли изменить этот бит. Применение мутации показано на рис. 18.13.

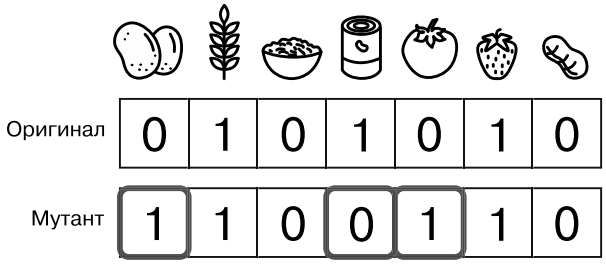


Рис. 18.13. Пример мутации для задачи о рюкзаке 0-1. Частота мутаций здесь чрезмерно завышена (для наглядности). В реальном сценарии она, вероятно, будет установлена где-то между 1 и 0,1 % (с учетом размера хромосом). Слишком большой коэффициент мутаций может нарушить стабильность сходимости алгоритма к минимуму, сводя на нет преимущества скрещивания и естественного отбора

Однако в современных генетических алгоритмах хромосомы могут принимать различную форму и иметь свои ограничения, поэтому мутации рискуют приобрести довольно сложную реализацию или применяться к хромосоме в целом, а не к отдельным генам. Как будет показано на примере задачи о коммивояжере, оператор мутации можно применить (с определенной вероятностью) только один раз ко всей хромосоме (но не к каждому ее гену) и поменять местами две вершины в маршруте.

Класс-обертка оператора мутации идентичен классу оператора скрещивания, показанному в листинге 18.7, за исключением незначительных отличий в обработке аргументов, особенно тех, которые относятся к обернутым методам: оператору мутации передается только один организм, но было бы неплохо также передать в обернутый метод шанс мутации. Это необходимо для побитовых мутаций, подобных той, что обсуждалась в задаче о рюкзаке и реализованной в листинге 18.9.

Листинг 18.9. Оператор мутации для задачи о рюкзаке 0-1

Метод knapsackMutation принимает хромосому (как последовательность битов) и вероятность мутации (как действительное число от 0 до 1) и применяет мутации к хромосоме. Метод в этой реализации изменяет аргумент, но возможно и часто проще клонировать входную хромосому и возвращать новый объект в качестве результата, если только это клонирование не становится узким местом. Взвесьте все за и против и проведите профилирование, прежде чем сделать выбор

```
function knapsackMutation(chromosome, mutationChance)
  for i in {0,..,|chromosome|} do
    if random() < mutationChance then
      chromosome[i] ← 1 - chromosome[i]
  return chromosome
```

Цикл по всем генам (битам) в хромосоме

С вероятностью mutationChance(×100) изменяем бит

[1] Несправедливая монета — дающая вероятность применения мутации намного меньше 1/2.

18.1.9. Шаблон генетического алгоритма

В листинге 18.10 демонстрируется реализация основного метода шаблона генетического алгоритма и завершается обсуждение задачи о рюкзаке 0-1. Теперь мы с вами имеем все, что нужно, чтобы запустить алгоритм на нашем экземпляре задачи и выяснить, что наилучшее возможное решение — взять в полет пшеничную муку, рис и бобы. Чтобы увидеть работу алгоритма в действии и, возможно, применить его к другим экземплярам задачи с сотнями возможных продуктов, можно использовать реализацию генетического алгоритма, предоставленную в JsGraphs на GitHub[1], и разработать для нее свои методы скрещивания, мутации и случайной инициализации, которые обсуждались в этом разделе. Это будет отличным упражнением, которое поможет вам проверить понимание техники и углубиться в ее детали.

Листинг 18.10. Обобщенная реализация генетического алгоритма

geneticAlgorithm — это шаблонный метод, реализующий основу генетического алгоритма. Его можно адаптировать для работы с несколькими задачами путем передачи специализированных методов для конкретных задач

```
function geneticAlgorithm(
➥ populationSize, chromosomeInitilizer, elitism, selectForMating,
➥ crossover, mutations, maxSteps)
population ← initPopulation(populationSize, chromosomeInitilizer)
  for k in {1..maxSteps} do
    population ← naturalSelection(
    ➥ population, elitism, selectForMating, crossover, mutations)
  return findBest(population).chromosome
```

Повторяем maxSteps раз. В каждой итерации создается новое поколение в популяции

Пусть естественный отбор идет своим чередом...

Инициализируем популяцию

В конце выбираем лучшего индивидуума в популяции и возвращаем его хромосому (представляющую лучшее найденное решение). Метод findBest тоже можно передавать через аргументы, если потребуется

18.1.10. Когда генетический алгоритм работает лучше всего

Подведем итог обсуждению. В главах 16 и 17 было представлено несколько методов, которые можно использовать в задачах оптимизации для поиска решений, близких к оптимальным, без исследования всего пространства задачи. У каждого из представленных алгоритмов есть свои сильные и слабые стороны.

- *Градиентный спуск* (см. главу 16) быстро сходится, но имеет тенденцию застревать в локальных минимумах. Он также требует, чтобы функция стоимости была дифференцируемой (поэтому его нельзя использовать в условиях экспериментов или в теории игр, например в эксперименте по эволюции роботов «хищник — жертва», описанном выше в этом разделе).
- *Случайная выборка* (см. главу 16) не предъявляет требований к дифференцируемости и не испытывает проблем с локальными минимумами, но этот метод (невыносимо) медленно сходится (если ему вообще удается сойтись).

[1] http://mng.bz/nMMd.

- *Имитация отжига* (см. главу 17) имеет те же преимущества, что и случайная выборка, но сходится более контролируемым и устойчивым путем. Тем не менее сходимость может быть медленной, и методу все-таки трудно найти минимумы, лежащие в узких долинах.

В начале главы отмечалось, что генетический алгоритм способен преодолеть многие проблемы, свойственные этим эвристикам; например, он может сходиться быстрее, чем имитация отжига, сохраняя и развивая пул решений.

В заключение этого раздела я хотел бы показать еще один критерий, который может помочь выбрать метод оптимизации. Он влечет за собой еще один термин, заимствованный из биологии, — *эпистаз*. Этот термин обозначает взаимозависимость генов и в нашей алгоритмической аналогии выражает наличие зависимых переменных, то есть переменных, значения которых зависят от других переменных.

Чтобы было понятнее, рассмотрим пример.

Каждый ген в хромосоме можно рассматривать как отдельную переменную, принимающую значения из определенного диапазона. В задаче о рюкзаке 0-1 каждый ген является независимой переменной, потому что принимаемое им значение не влияет напрямую на другие переменные. (Однако если наложить ограничение на вес каждого решения, то изменение значения гена с 0 на 1 может вынудить изменить значение одного или нескольких других генов с 1 на 0. В любом случае наблюдается слабая косвенная зависимость.)

В задаче о коммивояжере, как увидим далее, каждому гену назначается вершина и задается ограничение на появление дубликатов. Таким образом, присваивание значения переменной будет накладывать ограничения на все остальные переменные (которым нельзя присвоить то же значение), соответственно, мы получим более прямую, хотя и слабую зависимость.

Если бы решаемой задачей была оптимизация энергоэффективности проектируемого дома, а переменными были его площадь, количество комнат и объем древесины, необходимый для полов, то последняя переменная зависела бы от двух других, потому что площадь деревянного пола зависит от площади дома и количества комнат в нем. Изменение размера дома немедленно повлечет изменение объема необходимой древесины, даже если толщина пола останется неизменной (возможность изменения толщины делает целесообразным добавление отдельной переменной для этого параметра).

В отношении задачи о рюкзаке 0-1 мы говорим, что она имеет низкое взаимодействие переменных и, следовательно, низкий эпистаз; оптимизация дома имеет высокий эпистаз, тогда как для задачи о коммивояжере эпистаз находится где-то посередине.

Интересно отметить, что степень взаимодействия переменных влияет на форму ландшафта целевой функции: чем выше эпистаз, тем более волнистым будет ландшафт.

При высокой зависимости изменение одной переменной также меняет значения других переменных, а это означает бóльший скачок в ландшафте функции стоимости и пространстве задачи, что затрудняет исследование окрестностей решений и тонкую настройку результатов алгоритма.

Знание эпистаза задачи может помочь выбрать лучший алгоритм для применения:

- при низком эпистазе лучше всего работают алгоритмы поиска минимума, такие как градиентный спуск;
- при высоком эпистазе лучше предпочесть случайный поиск, то есть имитацию отжига или случайную выборку;
- а как насчет генетических алгоритмов? Оказывается, они лучше всего работают в широком диапазоне от среднего до высокого эпистаза.

Поэтому, разрабатывая функцию стоимости/приспособленности для задачи (что, хочу еще раз четко заявить, является одним из важнейших шагов на пути к хорошему решению), нужно быть осторожными со степенью взаимозависимости и выбирать лучший подходящий прием.

На этапе проектирования также можно попытаться уменьшить количество зависимых переменных, если такое возможно. Это позволит использовать более мощные методы и в конечном итоге получить лучшее решение.

Теперь, завершив обсуждение компонент и теории генетических алгоритмов, мы с вами готовы углубиться в практическое их применение, чтобы увидеть, насколько мощной является эта техника.

18.2. ЗАДАЧА О КОММИВОЯЖЕРЕ

В главе 17 вы познакомились с задачей о коммивояжере и рассмотрели ее решение, основанное на имитации отжига.

На рис. 18.14 показан тот же пример, что приводился в предыдущей главе, — экземпляр задачи с десятью городами, лучшим решением которой был маршрут с общей протяженностью 1625 миль:

```
["New Carthage", "Syracuse", "Buffalo", "Pittsburgh", "Harrisburg",
➥ "Opal City", "Metropolis", "Gotham City", "Civic City" ]
```

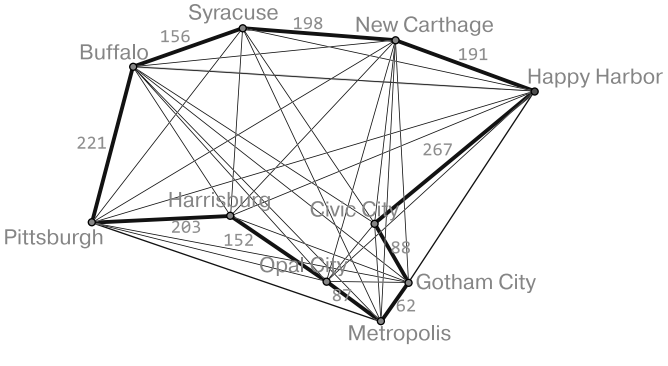


Рис. 18.14. Пример задачи о коммивояжере: лучшее решение, показанное на этом рисунке, имеет стоимость 1625

Теперь попробуем решить эту задачу с помощью генетического алгоритма и посмотрим, можно ли ускорить сходимость к хорошему решению (и улучшить среднее решение: не забывайте, что эти алгоритмы оптимизации выдают решения, близкие к оптимальным, поэтому не дают никаких гарантий, что всегда будут обнаруживать наилучшее решение из возможных).

18.2.1. Приспособленность, хромосомы и инициализация

В общем случае хорошей отправной точкой для любых задач является разработка способа кодирования решений и функции стоимости. В задаче с рюкзаком 0-1 требовалось максимизировать суммарную энергетическую ценность продуктов, укладываемых в рюкзак. Но задача о коммивояжере — это задача минимизации, поэтому будем предполагать использование алгоритма минимизации (как уже упоминалось, в противном случае можно просто максимизировать противоположное или обратное значение стоимости).

Следовательно, в роли функции приспособленности генетического алгоритма можно использовать функцию стоимости, которая была определена в разделе 17.2 и реализована в листинге 17.2: простую сумму расстояний между соседними вершинами, как показано на рис. 18.14.

Есть даже возможность использовать ту же самую кодировку для решений: они будут представлять перестановки вершин графа. Сказанное легко перевести на язык хромосом, которые в этом случае больше не будут последовательностями битов. Нам нужно, чтобы каждый ген имел целочисленное значение (индекс одной из вершин) с неявным ограничением, не позволяющим существовать двум генам с одинаковым значением.

Любое решение можно инициализировать как случайную перестановку индексов от 0 до $n - 1$, где n — количество вершин в графе, как это делалось в примере использования имитации отжига. Резонно даже передать тот же метод в аргументе `chromosomeInitializer` в вызов метода `initPopulation`, показанного в листинге 18.2.

18.2.2. Мутации

Даже в реализации мутаций является обоснованным (по крайней мере для первоначальной оценки) в значительной степени ограничиться повторным использованием методов, разработанных в главе 17 для локального поиска, а учитывая наше желание сравнить работу имитации отжига и генетического алгоритма, новые виды мутаций не должны добавляться в принципе.

Однако есть несколько отличий от упомянутых способов. В листинге 17.3 мы реализовали ансамблевый метод, который с некоторой вероятностью применяет один из возможных переходов. В генетических алгоритмах такой ансамблевый механизм уже неявно используется для применения мутаций, поэтому необходимо реализовать каждую мутацию (или локальный поиск) в виде отдельного метода.

Еще одно отличие: мутация, воссоздающая нового индивидуума с нуля, не имеет смысла для генетического алгоритма. Это противоречит принципам естественного отбора, а из предыдущего материала известно, что разнообразие генома в популяции уже поддерживается скрещиванием и локальными мутациями.

Следовательно, из листинга 17.3 можно извлечь два метода: один для перестановки соседних вершин и один для перестановки случайных вершин — и создать из них два оператора мутации. (Оба показаны в листинге 18.11. Будьте внимательны: ради простоты эти методы изменяют входные аргументы — для мутаций в генетических алгоритмах это нормальное явление. Вы должны знать об этом обстоятельстве и четко указать его в документации с описанием метода.) Чтобы увидеть эти алгоритмы в действии, обращайтесь к рис. 17.13 в предыдущей главе.

Листинг 18.11. Мутации для задачи о коммивояжере

```
Метод swapAdjacent принимает хромосому — вариант решения P
(точку в пространстве задачи, то есть перестановку списка вершин в графе)
и возвращает ту же хромосому после перестановки пары соседних вершин

function swapAdjacent(P)
  i ← randomInt(0, |P|)      Выбираем случайный индекс
  swap(i, (i+1) % |P|)       элемента массива и меняем
  return P                   местами с соседним элементом

function swapAny(P)
  i ← randomInt(0, |P|)      Выбираем случайные вершины для перестановки.
  j ← randomInt(0, |P|)      Обратите внимание, что два индекса могут
  swap(i, j)                 оказаться одинаковыми
  return P

Метод swapAny принимает хромосому и возвращает ту же хромосому
после перестановки случайно выбранных вершин
```

И последнее, хотя и не менее важное соображение: вероятность мутации в генетических алгоритмах, как правило, намного меньше отношения, использованного в методах имитации отжига (поскольку в имитации отжига эти «мутации» были единственным способом исследовать пространство задачи). Однако здесь мы попытались использовать те же вероятности, чтобы провести более справедливое сравнение. Через пару разделов обсудим полученные результаты.

18.2.3. Скрещивание

В скрещивании кроется настоящая инновация, по сравнению с имитацией отжига: возможность объединения двух хороших решений. По правде говоря, в имитации отжига изначально не было даже возможности сохранить несколько решений!

Комбинировать две перестановки одной и той же последовательности можно большим количеством способов. Хорошо, может быть, и не большим, но их определенно больше одного. Чтобы не усложнять, выберем самый простой способ, по крайней мере самый простой, который я смог придумать.

Все так же выберем единственную точку рассечения, как в задаче о рюкзаке 0-1, но дальше предпримем кое-что интересное. Чтобы лучше понять, рассмотрим небольшой пример. На рис. 18.15 показан в действии метод скрещивания для примера, который использовался, начиная с главы 17, — графа с десятью вершинами.

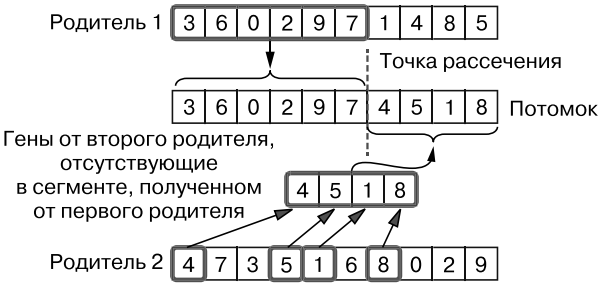


Рис. 18.15. Метод скрещивания для задачи о коммивояжере. Первую половину новой хромосомы можно напрямую скопировать у первого родителя, но, чтобы получить вторую половину у второго родителя, потребуется приложить больше усилий

После выбора случайной точки рассечения в новую хромосому копируются все гены из сегмента, полученного от первого родителя, начиная с гена с индексом 0 и до гена, предшествующего точке рассечения.

Теперь нужно заполнить остальную часть хромосомы потомка, но просто скопировать последовательность после точки рассечения (как это делалось в задаче о рюкзаке с последовательностями битов) нельзя, потому что последние четыре гена в хромосоме второго организма содержат вершины 0, 2 и 9, уже присутствующие в генах в первой половине новой хромосомы.

Вместо этого следует выполнить обход всей второй хромосомы от начала до конца и каждый раз, обнаруживая одну из вершин, которая не была скопирована из хромосомы родителя 1 в хромосому потомка (вершины 1, 4, 8 и 5), добавлять эту вершину в новую хромосому. В этом случае конечным результатом будет хромосома, включающая целиком часть хромосомы первого родителя от начала до точки рассечения, а вторая часть будет содержать вершины в том же порядке, в каком они встречаются в хромосоме второго родителя (где, однако, они могли находиться не по соседству друг с другом).

В листинге 18.12 показана возможная реализация этого метода.

Листинг 18.12. Метод скрещивания для задачи о коммивояжере

```
Выбираем точку рассечения так, чтобы от каждого родителя был взят хотя бы один ген
Инициализируем новую хромосому первой частью из chromosome1

Метод tspCossover принимает две хромосомы (как массивы) и возвращает новую хромосому, полученную путем рекомбинации входных данных

function tspCossover(chromosome1, chromosome2)
  i ← randomInt(1, |chromosome1|-1)
  newChromosome ← chromosome1[0:i]
```

```
genesFromChromosome1 ← new Set(newChromosome)
for j in {0,..,|chromosome2|-1} do
  if not chromosome2[j] in genesFromChromosome1 then
    newChromosome.add(chromosome2[j])
return newChromosome
```

Если значение j-го гена еще не добавлено в новую хромосому, то добавляем его в конец. Для большей эффективности используется множество, выполняющее поиск за амортизированное постоянное время. Также обратите внимание, что нет нужды обновлять множество, потому что оно не допускает появления дубликатов

Чтобы заполнить оставшиеся гены новой хромосомы, нужно последовательно перебрать гены в chromosome2

Сохраняем в множестве гены, полученные из chromosome1

18.2.4. Настройка результатов и параметров

После определения и реализации всех частей генетического алгоритма хотелось бы получить ответы на пару вопросов.

- Дает ли применение генетического алгоритма какое-то улучшение по сравнению с имитацией отжига?
- Насколько сильно скрещивание и мутации влияют на эффективность алгоритма?

Как вы, наверное, догадались, пришло время заняться профилированием!

Ответить на первый вопрос проще. Имитации отжига в тестах подраздела 17.2.4, основанных на реализации в JsGraphs, потребовалось примерно 600 итераций, чтобы сойтись к лучшему решению, и мы смогли получить среднюю стоимость, равную 1668,248 (напомню, что алгоритмы Монте-Карло, подобные имитации отжига и генетическим алгоритмам, не всегда дают наилучший результат).

На GitHub желающие могут увидеть реализацию генетического алгоритма для решения задачи о коммивояжере на основе JsGraphs[1]. Для популяции с десятью индивидуумами, с теми же скоростями мутаций, которые использовались в имитации отжига, и вероятностью скрещивания 0,7, алгоритму потребовалось в среднем породить десять поколений, чтобы сойтись к оптимальному решению (стоимость: 1625). Для корректности нужно сравнить количество итераций в имитации отжига (где сохраняется одно решение) с произведением количества итераций на размер популяции в генетическом алгоритме. Но даже в этом случае получаем снижение с 600 до 100 — очень хорошее улучшение.

Но истинное преимущество можно оценить, вычислив и сравнив среднюю стоимость одинакового количества итераций, например 1000 итераций имитации отжига и эволюционирование популяции из 25 индивидуумов на протяжении 40 поколений (или любой другой комбинации этих параметров, произведение которых равно 1000). Без особых усилий, затраченных на настройку параметров, была получена средняя стоимость, равная 1652,84, и это с использованием высокой вероятности как для

[1] https://github.com/mlarocca/jsgraphs/blob/master/src/graph/algo/genetic/tsp.mjs.

скрещивания, так и для мутации в виде перестановки вершин. Для популяции из 100 элементов среднее значение снижается до 1636,8.

Еще один интересный фактор, который хотелось бы понять: как шансы скрещивания и мутаций влияют на эволюцию популяции?

Чтобы ответить на этот вопрос, запустим алгоритм только с одним из этих операторов и нарисуем кривую изменения наилучшей приспособленности организма. Для получения более полного понимания и более ясных диаграмм используем усложненный пример задачи: полный граф с 50 вершинами (и случайными весами ребер).

На рис. 18.16 показана диаграмма с графиками тренда оптимизации для трех прогонов алгоритма на популяции со 100 организмами с эволюцией в течение 100 поколений. Первое, что можно заметить, — оператор скрещивания выходит на плато в какой-то (ранний) момент, но это не должно быть сюрпризом, верно? Мы с вами уже обсуждали роль скрещивания и его ограничения. В частности, оно может рекомбинировать геномы исходной популяции, но если ни один из организмов не имеет какого-то признака[1], то скрещивание сможет улучшить приспособленность лишь до определенного предела.

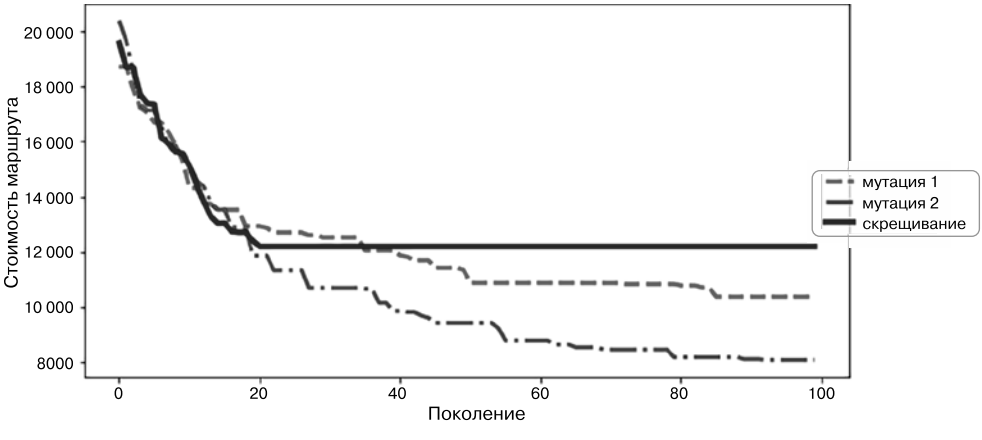


Рис. 18.16. Эволюция популяции в задаче о коммивояжере, когда разрешены только мутации или только скрещивание. Эволюция, основанная исключительно на скрещивании, достаточно быстро выходит на плато, в то время как эволюция, основанная на мутации, меняющей местами случайные пары, преуспевает больше. По оси абсцисс отложено количество поколений, а по оси Y — стоимость лучшего найденного маршрута

[1] Для представления в виде последовательностей битов, как отмечалось, это означает, что ни одна из хромосом не имеет определенного значения конкретного гена; для задачи о коммивояжере, имеющей другое представление, это означает отсутствие хромосомы с определенной подпоследовательностью вершин. Например, если ни одна хромосома не имеет вершины 1 перед вершиной 0, то оператор скрещивания не сможет добавить этот признак. Учитывая, что число пар вершин квадратичное, маловероятно, что оба возможных порядка для всех пар появятся в начальной популяции.

Можно еще заметить, что мутация 2 более эффективна, чем мутация 1, которая также в какой-то момент выходит на плато: и здесь тоже нет ничего удивительного. Если вы читали главу 17 об имитации отжига, где были представлены обе эти мутации, то видели, что перестановка только смежных вершин — это локальный поиск с малым диапазоном, он не позволяет алгоритму выйти из локальных минимумов.

Пока все хорошо, но что получится, если увеличить численность популяции? Это увеличит разнообразие геномов в исходной популяции, поэтому вполне можно ожидать увеличения эффективности скрещивания.

Диаграмма на рис. 18.17 подтверждает эту идею. Скрещивание заставляет алгоритм развиваться быстрее на начальных этапах и позволяет получить такое же хорошее решение, как и с использованием только наиболее эффективной мутации.

Но это был всего лишь счастливый случай. Результаты еще нескольких прогонов версии, выполняющей только скрещивание, показывают (рис. 18.18), что качество наилучшего найденного решения зависит от разнообразия и качества исходной популяции. Если алгоритму повезет на этапе инициализации, то он доберется до оптимального решения; в противном случае, как и ожидалось, одно только скрещивание выведет алгоритм на плато неоптимального решения. Результаты прогонов версий только с мутациями (не показаны на диаграмме) имеют меньшую дисперсию, и графики эволюции разных попыток выглядят одинаково.

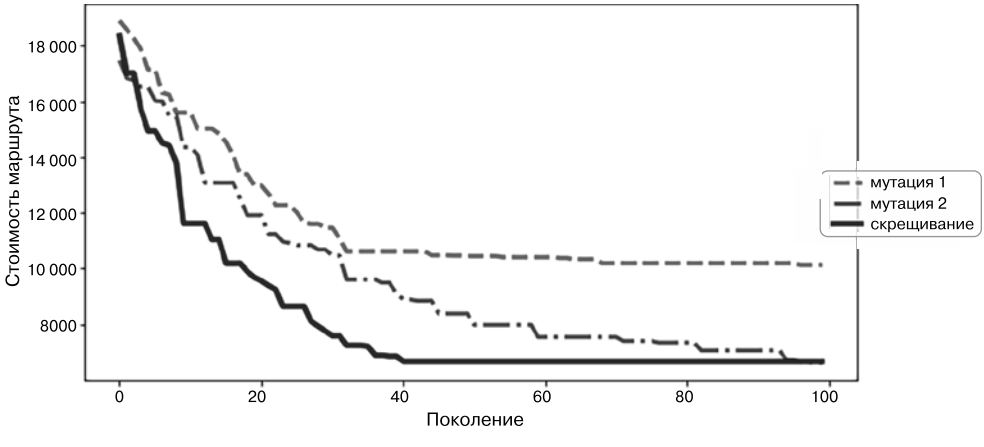


Рис. 18.17. Тот же анализ, что и на рис. 18.16, но с популяцией, увеличенной в десять раз. Для больших популяций скрещивание позволяет достичь лучших результатов в более ранних поколениях и даже приближается к результату, который дают мутации

Еще одно соображение, которое следует учесть: обе диаграммы на рис. 18.16 и 18.17 подтверждают, что мутация «перестановка соседних пар» менее актуальна для алгоритма и ее можно отбросить, оставив только мутацию «перестановка произвольных пар».

Итак, дает ли какое-то преимущество одновременное использование скрещивания *и* мутации? Чтобы выяснить это, нужно отказаться от диаграмм (они дают представление о тенденции моделирования, но не являются статистически значимыми, потому что основаны всего на нескольких прогонах) и провести более тщательный анализ, обобщенный в табл. 18.2.

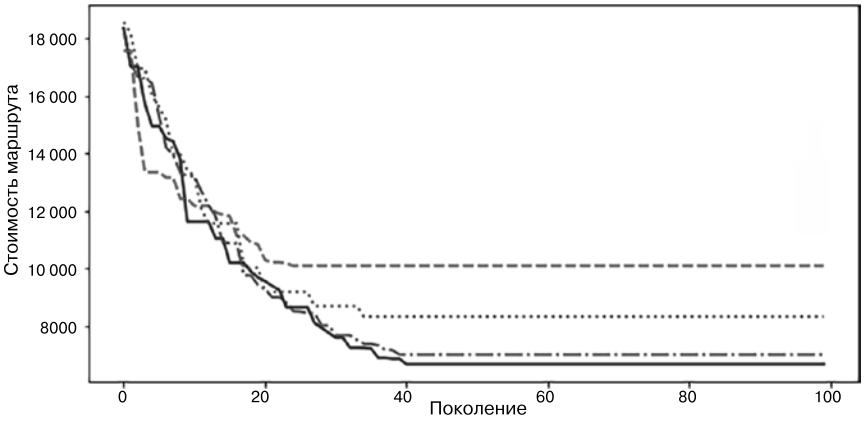


Рис. 18.18. Тот же анализ, что и на рис. 18.17. Было проведено несколько экспериментов с использованием только скрещивания (все с одинаковым шансом скрещивания 0,7). Эта диаграмма показывает, насколько велика разница в конечных результатах, которые теперь полностью зависят от выбора первоначальной популяции (из-за отсутствия мутации)

Таблица 18.2. Лучшее среднее решение генетического алгоритма для задачи о коммивояжере в зависимости от вероятностей применения операторов. Результаты отсортированы по стоимости маршрута, а наиболее значимые строки выделены

Скрещивание	Перестановка соседних вершин	Перестановка произвольных вершин	Средняя стоимость маршрута
0,2	0	0,5	6326,447
0,3	0,002	0,5	6379,628
0,2	0,02	0,5	6409,664
0,1	0,002	0,5	6428,103
0,2	0	0,2	6555,593
0,2	0,2	0,5	6753,441
0,2	0,2	0,2	6823,971
0,2	0	0,7	7016,033
0	0	0,2	7110,798
0	0	0,7	7143,873
0,7	0	0,7	7818,061
0	0,7	0	10 428,917
0,7	0	0	11 655,949
0,2	0	0	15 210,800

Как видите, наилучшие результаты получаются при объединении скрещивания и мутаций. Мутация перестановки произвольных вершин важнее скрещивания, тогда как мутация перестановки смежных пар не только менее важна, но и может даже стать вредной, когда ее вероятность слишком высока, что также относится к скрещиванию. Это может вызвать недоумение: в конце концов, самое худшее, чего можно было бы ожидать, — бесполезность мутации. Как объяснить тот факт, что высокая вероятность скрещивания и мутации 1 в среднем приводит к худшим решениям? Дело в том, что даже при использовании элитарности, если у других организмов в новых поколениях наблюдается подверженность множественным изменениям, высока вероятность, что их приспособленность ухудшится, и этот эффект проявляется все сильнее с каждым поколением, тогда как средняя приспособленность становится лучше. Если решение оказывается близко к оптимальному, то результат случайного изменения может оказать ухудшающее воздействие, а не улучшающее.

Объясняется это тем, что, в отличие от имитации отжига, отвергающей ухудшающие изменения на последних этапах симуляции, генетический алгоритм принимает их безоговорочно. Помимо небольшой доли высокоприспособленных индивидуумов, потенциально гарантированных элитарностью, остальная часть популяции может показывать снижение средней приспособленности, если одновременно происходит слишком много случайных изменений.

И последнее: к полученным здесь результатам следует относиться с некоторой долей скептицизма, потому что среднее значение вычисляется «всего лишь» по тысяче прогонов, а также потому, что область значений параметров исследована недостаточно широко.

ПРИМЕЧАНИЕ

Для точной систематической настройки параметров существует эффективный протокол, часто используемый в машинном обучении. Сначала создается множество разнородных значений, чтобы с его помощью найти порядок величины для лучшего решения (например, для вероятности мутации можно взять множество [0,01, 0,05, 0,1, 0,2, 0,5, 0,8, 1]), а затем результаты уточняются путем многократного повторения процесса вокруг наиболее многообещающих значений. (Например, предположим, что наилучшие результаты были получены со значениями 0,2 и 0,5, тогда можно проверить значения [0,15, 0,2, 0,25, 0,3... 0,45, 0,5, 0,55, 0,6].) Поскольку у нас несколько параметров и требуется измерить результаты для всех комбинаций (как показано в табл. 18.2), желательно, чтобы эти списки были как можно короче и уточнялись в несколько шагов.

18.2.5. За пределами задачи о коммивояжере: оптимизация маршрутов для всего парка грузовиков

Как упоминалось в главе 17, решение задачи о коммивояжере — это хорошо, но оно позволяет оптимизировать маршрут только одного транспортного средства. Для процветающего бизнеса естественно иметь несколько грузовиков, занимающихся доставкой товаров с разных складов (порой даже с нескольких складов сразу).

Каждый грузовик имеет ограниченную грузоподъемность, поэтому при планировании нужно учитывать вес заказов (ничего не напоминает?), а кроме того, пункт доставки каждого заказа может существенно изменить маршрут, необходимый для доставки всех заказов, закрепленных за грузовиком.

Это чрезвычайно сложная задача для оптимизации, потому что, выбирая грузовик для доставки каждого заказа, нужно оптимизировать маршруты транспортных средств одного за другим.

Для упрощения задачи район, обслуживаемый складом, часто разбивают на зоны и определяют заведомо оптимальные привязки этих зон к грузовикам. Это, однако, вынуждает соглашаться на неоптимальные решения и использовать грузовики избыточной грузоподъемности, чтобы совладать с колебаниями спроса.

В качестве альтернативы можно запустить оптимизацию задачи о коммивояжере для каждого грузовика после назначения им заказов, а затем суммировать любые используемые показатели (пройденные мили, затраченное время или израсходованное топливо).

Как нетрудно представить, оптимизировать более крупную задачу сложнее, потому что для любого переназначения заказов нужно повторно выполнить не менее двух оптимизаций задачи о коммивояжере.

С точки зрения *эпистаза* (как определено в подразделе 18.1.9) переменные функции приспособленности сильно взаимосвязаны, поэтому эффективным может оказаться не только генетический алгоритм, но и случайный поиск.

18.3. МИНИМАЛЬНОЕ ВЕРШИННОЕ ПОКРЫТИЕ

Мы с вами можем быть вполне довольны результатами применения генетического алгоритма для решения задачи о коммивояжере и переходить к новым интересным и, конечно же, *NP-трудным* задачам на графах.

Решению такого рода задач посвящено много книг и статей, и некоторые из них считаются вехами и ориентирами эпохального значения в информатике, а некоторые из них нашли широкое практическое применение в инженерии.

Минимальное вершинное покрытие (minimum vertex cover) — важная техника и с точки зрения теории, и с точки зрения практики, поэтому недопустимо было бы завершить эту книгу, не обсудив ее!

Для графа $G = (V, E)$ *вершинным покрытием* называется любое множество вершин, охватывающее (покрывающее) все его ребра; ребро $e = (u, v) \in E$ охватывается подмножеством вершин $S \subseteq V$, если хотя бы один из концов ребра, u или v, принадлежит S.

У каждого графа есть тривиальное вершинное покрытие: множество V всех вершин. Но настоящий интерес представляет поиск минимального покрытия вершин,

то есть наименьшего возможного подмножества вершин, охватывающего все ребра. На рис. 18.19 показано несколько примеров простых графов и их вершинных покрытий.

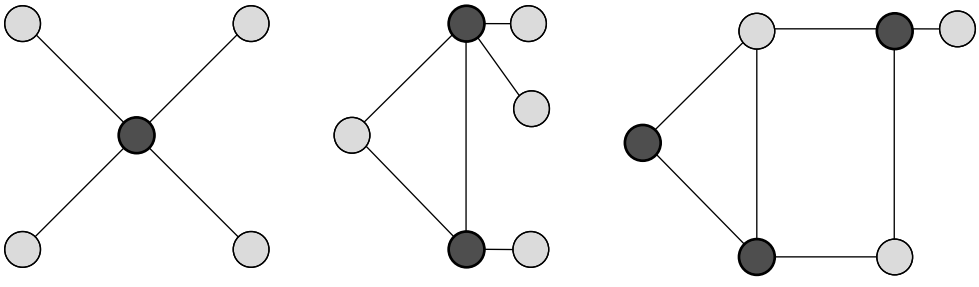


Рис. 18.19. Три примера минимального покрытия вершин в графах. Более толстые окружности (темные вершины), они и только они необходимы для покрытия всех ребер этих графов

Обратите внимание, что некоторые графы имеют единственное решение (например, первый слева на рис. 18.19), тогда как другие — несколько. Попробуйте сами найти альтернативное решение для графа справа.

На рис. 18.20, напротив, показаны примеры, *не* являющиеся минимальным вершинным покрытием.

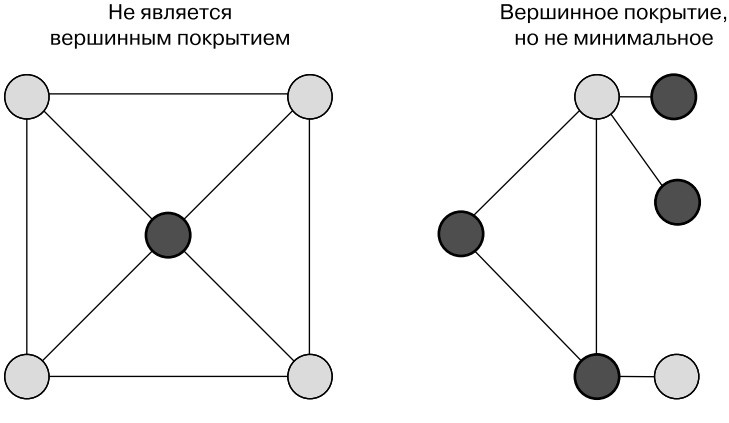


Рис. 18.20. Примеры подмножеств, не являющихся минимальным покрытием вершин для этих графов. *Слева* не все ребра охватываются вершиной в центре графа, так что эта вершина не входит в вершинное покрытие. *Справа* показан пример вершинного покрытия, не являющегося минимальным (см. рис. 18.19)

Было доказано, что задача поиска минимального вершинного покрытия (покрытия, которое удовлетворяет требованию: «данное подмножество вершин является минимальным вершинным покрытием для G») относится к NP-трудным.

Поскольку существует $2^{|V|}$ возможных подмножества V, поиск простым перебором кажется невозможным.

Существуют алгоритмы, отыскивающие приближенное решение за полиномиальное время, но они могут гарантировать только результат, находящийся в пределах коэффициента аппроксимации, близкого к 2 (чуть меньше 2) от наилучшего решения[1]. Эта задача фактически принадлежит к вычислительному подклассу NP, называемому *APX-полными задачами*, — множеству NP-задач оптимизации, для которых существует алгоритм аппроксимации за полиномиальное время с постоянным коэффициентом.

Для некоторых категорий графов, таких как двудольные графы и деревья, существуют алгоритмы, отыскивающие точное решение задачи минимального вершинного покрытия за полиномиальное время в наихудшем случае, но в других случаях приходится либо согласиться на приближенное (неоптимальное) решение, либо использовать экспоненциальный алгоритм.

Разумеется, я намекаю на первый вариант и генетический алгоритм. Но сначала поговорим о практическом применении этой задачи.

18.3.1. Практическое применение вершинного покрытия

Вернемся к нашей компании электронной коммерции: по мере развития бизнеса растут и склады. Чтобы защитить компанию от краж и гарантировать безопасность сотрудников, необходимо установить камеры, охватывающие все переулки складов. Предположим, что склады имеют форму графа, изображенного на рис. 18.20, где ребра — это переулки, а вершины расположены в точках пересечения и тупиках. Предположим также, что камеры оснащены специальными широкоугольными объективами (или поворотными механизмами, или и тем и другим) и могут охватывать до 180° или даже 360°. В таком случае минимальное количество необходимых камер определяется минимальным вершинным покрытием этих графов.

Несмотря на упрощение, это реальное практическое применение задачи, его проще всего адаптировать к нашему сценарию, но определенно, оно не единственное.

Еще одна область, где востребованы эффективные алгоритмы вершинного покрытия, — биоинформатика и, в частности, вычислительная биохимия. Часто бывает так, что последовательности ДНК в образцах представляют конфликты (точное определение конфликта зависит от контекста), которые необходимо разрешить. В таких случаях на помощь исследователям приходят графы. Можно определить граф конфликтов, в котором вершины являются последовательностями, а ребра добавляются, когда возникает конфликт между любой парой последовательностей. Обратите внимание, что этот граф потенциально несвязный.

---

[1] Если алгоритм имеет коэффициент аппроксимации a, это означает, что для задачи с решением, значение которого равно v, алгоритм аппроксимации вернет решение, не превосходящее a × v.

Поскольку цель состоит в том, чтобы разрешить все конфликты, удалив как можно меньше последовательностей, нужно отыскать минимальное вершинное покрытие графа конфликтов.

В области информатики вершинное покрытие можно применять для повышения безопасности сети. Например, решение этой задачи использовалось[1] для разработки стратегий маршрутизации, оптимальных с точки зрения комбинаторной топологии, препятствующих распространению червей-невидимок в больших компьютерных сетях.

Наконец, во многих других областях тоже можно найти похожие задачи. Но мне хотелось бы упомянуть одну уже обсуждавшуюся в этой книге: минимальное вершинное покрытие использовалось[2] для разработки эффективного классификатора ближайших соседей для обобщенных метрических пространств (не ограниченных евклидовым или гильбертовым пространством).

18.3.2. Реализация генетического алгоритма

Теперь пришло время засучить рукава и написать достойный оптимизатор для этой задачи. Самое замечательное, что мы можем повторно использовать большую часть работы, проделанной в процессе поиска решения задачи о рюкзаке 0-1!

Решениями задачи поиска минимального вершинного покрытия, по сути, являются подмножества вершин точно так же, как для задачи о рюкзаке решениями были подмножества доступных продуктов. Соответственно, есть возможность реализовать хромосомы в виде последовательностей битов и повторно использовать одни и те же операторы скрещивания и мутации!

Единственное, что поменяется, — это определение функции приспособленности. Здесь качество решения определяется количеством вершин в подмножестве (чем меньше, тем лучше) при условии покрытия всех ребер. Можно было бы реализовать ограничение отдельно и отбрасывать все подмножества, не являющиеся покрытиями вершин, или присваивать им огромное значение приспособленности. Однако в этом случае минимальное решение, охватывающее все ребра, кроме одного, будет оштрафовано и получит значение приспособленности даже большее, чем тривиальное решение, содержащее все вершины (которое является действительным вершинным покрытием). Вместо этого важно сохранить предыдущее решение в популяции, так как его можно превратить в действительное вершинное покрытие, добавив одну вершину.

Для обхода этой проблемы я предлагаю интегрировать количество непокрытых ребер в функцию приспособленности с определенным множителем (например, 2).

---

[1] *Filiol E. et al.* Combinatorial optimization of worm propagation on an unknown network // International Journal of Computer Science, 2007. — № 2. — P. 124–130.

[2] *Gottlieb L.-A., Kontorovich A., Krauthgamer R.* Efficient classification for metric data // IEEE Transactions on Information Theory, 2014. — № 60. — P. 5750–5759.

Возьмем за основу первый пример на рис. 18.20 и сравним два возможных решения, показанных на рис. 18.21. Граф слева имеет одно непокрытое ребро, поэтому его приспособленность равна 5, то есть приспособленности тривиального решения *(справа)*, которое является действительным вершинным покрытием. Этот пример также показывает, насколько важно, чтобы множитель для количества непокрытых ребер был больше 1. Если просто сложить количество непокрытых ребер с количеством вершин, то решение *слева* имело бы величину приспособленности, равную 4, но любое подмножество с четырьмя вершинами в действительности было бы более оптимальным и правильным решением.

В то же время, используя эту функцию приспособленности, мы не отбрасываем многообещающее решение, которое находится всего в двух мутациях от минимального вершинного покрытия (с тремя вершинами).

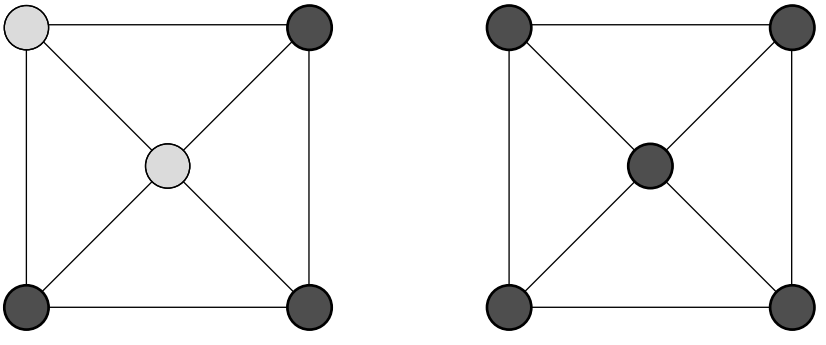


Рис. 18.21. Оценка приспособленности возможных решений. Мы складываем количество вершин с количеством непокрытых ребер, чтобы не отбрасывать многообещающие решения, которые не являются действительными вершинными покрытиями, подобными тому, что показано *слева*

В листинге 18.13 показана реализация функции приспособленности для этого метода. Но обратите внимание, что, в отличие от задачи о рюкзаке, здесь также нужно передать методу экземпляр графа, чтобы он мог проверить, какие ребра покрыты решением. Во многих языках этого можно добиться без изменения шаблона генетического алгоритма (показанного в листингах 18.10 и 18.4) путем каррирования функции приспособленности и привязки ее к экземпляру графа перед передачей основному методу генетического алгоритма.

Например, JavaScript — один из языков, который позволяет это сделать: конкретную реализацию можно найти в составе библиотеки JsGraphs на GitHub[1].

И это почти весь новый код, нужный для реализации решения задачи минимального вершинного покрытия. Но есть один тонкий момент.

[1] https://github.com/mlarocca/jsgraphs/blob/master/src/graph/algo/genetic/vertex_cover.mjs.

Листинг 18.13. Вершинное покрытие: функция приспособленности

```
function vertexCoverFitness(graph, chromosome)
  fitness ← 0
  for gene in chromosome do
    if gene == 1 then
      fitness ← fitness + 1
  for edge in graph.edges do
    if chromosome[edge.source] == 0
      and chromosome[edge.destination] == 0 then
      fitness ← fitness + 2
  return fitness
```

Инициализируем значение приспособленности нулем

Метод vertexCoverFitness принимает хромосому (последовательность битов) и возвращает значение приспособленности, связанное с заданным решением

Если ген имеет значение 1, это означает, что данная вершина уже имеется в решении, поэтому ее нужно учесть в стоимости. Напомню, что цель состоит в том, чтобы свести к минимуму количество используемых вершин, обеспечив при этом правильное покрытие вершин

Цикл по всем генам (битам) в хромосоме

Цикл по всем ребрам в графе

Если ни одна из конечных точек ребра не включена в решение, это ребро не покрыто (решение не является допустимым вершинным покрытием). Но мы не отбрасываем такое решение, а просто добавляем штраф к значению приспособленности. В этом случае добавляем 2, но точно так же можно передавать желаемую величину штрафа в виде аргумента и сделать этот метод более гибким

Чтобы сохранять уверенность в возврате действительного вершинного покрытия, нужно отсортировать конечную популяцию по приспособленности, проверить решения, начиная с того, которое имеет наименьшее значение пригодности, и вернуть первое в отсортированном списке, которое покрывает все ребра.

18.4. ДРУГИЕ ВАРИАНТЫ ПРИМЕНЕНИЯ ГЕНЕТИЧЕСКОГО АЛГОРИТМА

Как в сфере графов, так и в сфере обобщенных структур данных существует бесчисленное множество задач, которые можно сформулировать как задачи оптимизации. Для них генетические алгоритмы могли бы эффективно и за разумное время находить решения, близкие к оптимальному.

Мы с вами кратко обсудим два из них, особенно важных для практического применения: поиск максимального потока в графе и свертывание белка.

18.4.1. Максимальный поток

Задача о максимальном потоке определяется на ориентированных графах определенного типа, называемых сетями (рис. 18.22). Сеть $N = (V, E)$ — это ориентированный взвешенный граф, имеющий две специальные вершины $s, t \in V$: s называется истоком (source), а t — стоком (sink) сети. Особенность этих вершин в том, что у истока есть только исходящие ребра, а у стока — только входящие.

В сетях веса ребер называются *емкостью*. Точная характеристика пропускной способности ребер зависит от контекста. Например, для гидравлической сети ребра моделируют трубы, а их вес — максимальный объем жидкости, который может пройти через трубу в любой момент времени.

Один из вариантов применения сетей — моделирование потока через вершины, начиная с истока и заканчивая стоком, где поток ребра — это фактическое количество (например, воды), проходящее через ребро. Каждая вершина v сети имеет входящий поток (например, общее количество воды, поступающее через входящие ребра), он может быть перераспределен в исходящие ребра с оговоркой, что суммарный исходящий поток должен быть равен входящему. Например, как показано на рис. 18.22, узел D может иметь максимальный входящий поток (сумма пропускной способности всех его входящих ребер) 5 единиц и максимальный исходящий поток 6 единиц. Следовательно, реальный поток через D никогда не превысит 5, а это возможно, только когда входящий поток этого узла максимален: если в какой-то момент входящий поток D будет равен 4 по той причине, что часть потока из A была направлена в E вместо D, то исходящий поток D тоже ограничится 4 единицами. Когда максимально возможный исходящий поток больше входящего, как в этом примере, это означает необходимость управления исходящим потоком, потому что нет возможности использовать полную пропускную способность исходящих ребер. Например, учитывая, что максимальный поток от F к стоку равен 1, имеет смысл направить поток из D к G, чтобы максимизировать общий входящий поток стока t: это также является целью задачи максимального потока.

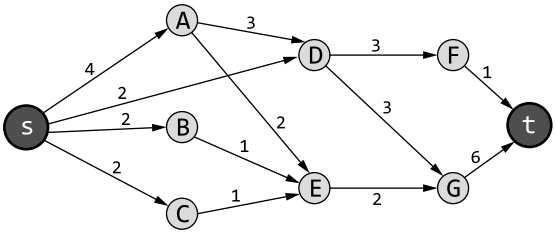


Рис. 18.22. Пример сети с максимальным потоком 6

Более формально поток в сети определяется как назначение ребрам действительных чисел, таких что:

- поток через ребро не может превышать максимальную пропускную способность этого ребра;
- общий объем потока, входящий в вершину, должен быть равен общему объему потока, исходящего из вершины (за исключением s и t, которые по определению имеют *нулевые объемы входящего* и *исходящего* потока соответственно).

Объем потока определяется как сумма потока, выходящего из вершины истока s, или, что то же самое, сумма потока, входящего в вершину стока t.

Максимальный поток для сетевого графа G — максимально возможный объем допустимого потока для G.

Обратите внимание, что задача о максимальном потоке не является NP-трудной, потому что существует несколько полиномиальных алгоритмов, дающих точное решение. Эти алгоритмы, однако, имеют сложность не ниже $O(|V|^3)$, что делает их непрактичными для очень больших графов; более того, существуют NP-полные варианты этой задачи.

Теперь, имея четкую формулировку задачи, попробуем разработать для ее решения генетический алгоритм.

Прежде всего, как обычно, нужно определить функцию приспособленности и кодировку (способ представления) хромосом. Последнее решается просто — хромосома содержит по одному гену на каждое ребро, и он может принимать любое значение от 0 до пропускной способности ребра.

Этот способ кодирования (рис. 18.23) автоматически проверяет первое ограничение в формальном определении потока, но ничего не может сделать со вторым. Оказывается, все еще нужно проверять каждое решение, потому что оно будет действительным, только если соблюдается второе ограничение.

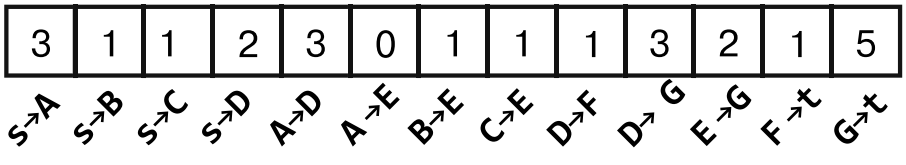


Рис. 18.23. Решение (хромосома) для задачи поиска максимального потока сети на рис. 18.22. Это правильный поток? Максимальный ли он?

Поступать с такими решениями можно двумя способами: либо отбрасывать недопустимые потоки (вероятно, не лучший выбор по тем же причинам, что и в задаче вершинного покрытия), либо добавить в функцию приспособленности член, учитывающий это.

Например, можно вычислить абсолютное значение разницы потоков для всех вершин (кроме `s` и `t`) и разделить поток решения на накопленную разность, если она не равна нулю.

Скрещивание и мутации выглядят проще. Можно использовать одноточечную рекомбинацию и точечную мутацию, которая случайным образом увеличивает или уменьшает объем потока через ребро на 1 (в пределах пропускной способности ребра).

Дочитав главу до этой точки, вы уже получили все необходимые инструменты не только для реализации этого алгоритма, но и для разработки более сложных и, возможно, лучших операторов, а также для опробования различных определений функции приспособленности.

Поэтому вместо обсуждения реализации я хотел бы сказать несколько слов о том, почему эта задача считается важной.

Есть несколько практических применений задачи поиска максимального потока в физическом мире. Например, составление расписания летного экипажа авиакомпании или, что еще интереснее, *задача обращения и спроса*. Вы наверняка помните, наша компания электронной коммерции искала способы максимизировать загрузку грузовиков и минимизировать избыточную мощность, необходимую для преодоления пиков спроса. Оказывается, эту задачу можно сформулировать в виде задачи о максимальном потоке!

Фактически в задаче обращения и спроса имеется источник товаров — наш склад и набор пунктов назначения, куда нужно доставить эти товары. Однако существуют и свои ограничения. Во-первых, максимальная грузоподъемность каждого транспортного средства является непреодолимым пределом, потому что в грузовике нельзя перевезти больше товаров, чем может в нем поместиться. А при наличии нескольких складов/заводов задача становится еще более сложной.

Но, за исключением подобных практических проблем, *максимальный поток* — классическая задача информатики и разработки программного обеспечения!

Примерами задач, которые нетрудно смоделировать как задачу оптимизации максимального потока, могут служить: сегментация изображения, планирование, подключение к сети и оптимизация в компиляторах.

18.4.2. Свертывание белков

В последних нескольких главах все наше внимание было сосредоточено на задачах с графами, которые можно решить с помощью оптимизации (мета)эвристики. Прежде чем завершить обзор практических применений генетических алгоритмов, я хотел бы упомянуть задачу другого рода, очень важную для нас в эти трудные времена.

Белки — это большие молекулы, последовательности аминокислот, закодированные в ДНК организмов (после расшифровки ДНК клетками). Они вырабатываются клетками для выполнения множества различных функций внутри организма, среди них такие: регулирование реакции на раздражители, перенос более простых молекул внутри и между клетками, поддержание и ускорение метаболических реакций и даже сама репликация ДНК. То есть белки играют важную роль в функционировании любого организма.

Два белка отличаются своими последовательностями аминокислот, определяющими трехмерную структуру белка, а она, в свою очередь, определяет активность белка.

Помимо прочего, трехмерная структура поверхностных белков определяет способность вирусов связываться с рецепторами и инфицировать определенные типы клеток. Это не что иное, как та самая способность, которая недавно выдвинула свертывание белков на первое место в списке самых насущных задач, требующих решения.

Свертывание белка — это физический процесс, посредством которого линейная цепь белка (также известная как *полипептид*) приобретает свою трехмерную конформацию.

Последовательность аминокислот в белке определяется ДНК, которая его кодирует, и ее сравнительно легче обнаружить экспериментально. Однако именно трехмерная структура белка определяет его функциональность, а вот ее определить в лаборатории гораздо сложнее (хотя разрабатываются новые методы микровизуализации).

Одним из первых и наиболее успешных вариантов применения генетических алгоритмов было определение наиболее вероятной трехмерной структуры белков[1] с известной последовательностью аминокислот. Идея аналогична алгоритмам силовой визуализации графов, что обсуждалось в главах 16 и 17 (еще одна причина, почему та тема была так важна!). Можно разработать функцию приспособленности с учетом сил притяжения и отталкивания между пептидами (последовательностями аминокислот), аминокислотами и далее вплоть до атомов и таким образом найти трехмерную конфигурацию системы с минимальной энергией.

Генетические алгоритмы и искусственные иммунные системы были теми эвристиками, которые находились в числе фаворитов для решения задачи свертывания белков. Стоит отметить, что на момент написания книги в этой области используются еще более современные вычислительные приемы, тоже основанные на алгоритмах, заимствованных из биологии, — нейронных сетях[2].

18.4.3. За пределами генетических алгоритмов

Генетические алгоритмы — это всего лишь ветвь более широкой категории *эволюционных алгоритмов* (ЭА), включающей все обобщенные метаэвристические алгоритмы оптимизации на основе популяции.

Я не мог выбрать лучшего способа завершить эту главу и книгу, чем дать краткий обзор наиболее интересных эволюционных алгоритмов в литературе по информатике. Считайте это отправной точкой для дальнейшего изучения алгоритмов оптимизации.

Меметический алгоритм[3] — это класс метаэвристик оптимизации, непосредственно полученный из генетических алгоритмов. Его особенность заключается в наличии одного или нескольких операторов мутации, фактически выполняющих эвристику локального поиска, что-то вроде случайной выборки в окрестности текущего решения, и направляющих каждого индивидуума к локальному оптимуму в каждой

---

1 *Schulze-Kremer S.* Genetic algorithms and protein folding // Protein Structure Prediction. Humana Press, 2000. — P. 175–222.

2 Проект AlphaFold, разработанный в DeepMind: http://mng.bz/Qmm4.

3 *Moscato P.* On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms // Caltech concurrent computation program, C3P Report 826, 1989.

итерации (например, такие операторы пытаются изменить каждый бит в хромосоме по отдельности и сохранить наилучший результат). Подобные качества делают этот алгоритм гибридом генетического алгоритма со случайной выборкой или, возможно, даже с градиентным спуском. Ключом в любой ситуации является использование дополнительных знаний о предметной области для выполнения локальной оптимизации.

Искусственная иммунная система[1] (Artificial immune system, AIS) — класс алгоритмов, заимствованных из биологии, моделирующих иммунную систему и ее способность обучаться и запоминать. AIS расширяет возможности генетических алгоритмов такими операторами, как *клональная селекция*, *созревание аффинности* и *негативная селекция*.

Оптимизация роя частиц[2], которая лучше подходит для численной оптимизации, использует динамику роя для изучения ландшафта функции стоимости. Механизм аналогичен механизму генетических алгоритмов с популяцией (роем) решений, которые поддерживаются и перемещаются не с помощью генетических операторов, а с помощью более простых правил, основанных на сочетании кинематики и биологии.

Оптимизация колонии муравьев[3] — вероятностный метод, действующий подобно колонии муравьев, использующей коммуникацию на основе феромонов для формирования путей к источникам пищи. На редкость хорошо подходит для задач с графами и особенно тех, которые можно свести к поиску путей в графе.

И названные примеры — лишь вершина айсберга; существует масса других эволюционных алгоритмов, основанных на самых разных биологических принципах. Алгоритмов так много, что для их описания потребовалась бы отдельная книга.

18.4.4. За пределами круга алгоритмов, рассмотренных в этой книге

Мы подошли к концу книги, но это не конец вашего путешествия по алгоритмам! Если вас заинтересовали темы этой книги, то вот список литературы, которая может помочь углубиться в некоторые из них:

- *Grokking Machine Learning*[4] Луиса Серрано (Luis Serrano); здесь подробно рассказывается о машинном обучении и некоторых алгоритмах обучения, кратко

---

1 *Kephart J. O.* A biologically inspired immune system for computers // Proc. Of The Fourth International Workshop On Synthesis And Simulation Of Living Systems, Artificial Life IV, 1994.

2 *Eberhart R. C., Shi Y., Kennedy J.* Swarm Intelligence. — Elsevier, 2001.

3 *Dorigo M., Gambardella L. M.* Ant colony system: a cooperative learning approach to the traveling salesman problem // IEEE Transactions on evolutionary computation 1.1, 1997. — P. 53–66.

4 *Серрано Л.* Грокаем машинное обучение. — СПб.: Питер, 2024.

описанных в главах 12, 13 и 16. Издание послужит отличной отправной точкой для всех, кто желает узнать, как используется градиентный спуск в линейной и логистической регрессии.

- *Grokking Artificial Intelligence Algorithms* Ришала Харбанса (Rishal Hurbans)[1]; в работе подробно рассмотрены некоторые из тем, кратко представленных в моей книге, таких как алгоритмы поиска и оптимизации и эволюционные алгоритмы, описаны передовые идеи, которые я только что упомянул, например роевой интеллект.
- *Graph-Powered Machine Learning* Алессандро Негро (Alessandro Negro); хороший выбор для тех, кто собирается на практике применить полученные здесь знания о графах и заняться машинным обучением с другой точки зрения.
- *Graph Databases in Action* Дэйва Бехбергера (Dave Bechberger) и Джоша Перримана (Josh Perryman); авторы исследуют еще одно современное применение графов. Из их книги вы узнаете, как графы позволяют улучшить моделирование отношений внутри данных и почему графовые базы данных являются предпочтительным выбором для хранения сильно взаимосвязанных данных.
- *Algorithms and Data Structures for Massive Datasets* Дзейлы Медьедович (Dzejla Medjedovic) и др.; здесь развиваются темы, начатые в моей книге, особое внимание уделяется как методам адаптации структур данных и алгоритмов к массивным наборам данных, так и современным приложениям для работы с большими данными.

А теперь, если вы решите проверить свое понимание структур данных, которые были вам здесь предложены к ознакомлению, вот вам задание от меня: реализуйте приведенные алгоритмы на любом языке по вашему выбору и добавьте реализацию в мой репозиторий[2] на GitHub! Подробные инструкции вы найдете в файле README[3].

Удачи вам в последующем обучении и я надеюсь, что мое задание вам понравится!

РЕЗЮМЕ

- Генетические алгоритмы измеряют приспособленность организмов, повышают шансы выжить и передать свой геном следующим поколениям для тех из них, которые лучше всего адаптируются к окружающей среде. Все происходит как в реальной динамике популяций.
- *Скрещивание* — это новая идея по сравнению с другими алгоритмами оптимизации, такими как имитация отжига. Оно позволяет случайным образом рекомбинировать признаки нескольких организмов.

---

1 *Харбанс Р.* Грокаем алгоритмы искусственного интеллекта. — СПб.: Питер, 2023.

2 https://github.com/mlarocca/AlgorithmsAndDataStructuresReadersContributions.

3 http://mng.bz/1rjn.

- *Мутация* — биологическая аналогия оптимизации локального поиска, действует подобно операторам перехода в имитации отжига.
- Если функция стоимости использует множество тесно связанных переменных и изменение одной из них приводит к изменению одной или нескольких других, то задача (в смоделированном виде) имеет высокий *эпистаз*.
- Когда переменные тесно взаимосвязаны, то имитация отжига может превосходить генетические алгоритмы по эффективности. Стоит попробовать оба подхода на небольших экземплярах задачи и по результатам сделать выбор в пользу одного из них.
- *Вершинное покрытие* — это задача с низким эпистазом, в которой особенно ярко проявляют себя генетические алгоритмы.

Приложения

А Краткое руководство по псевдокоду

Для описания работы алгоритмов в этой книге решено использовать псевдокод. Это решение продиктовано двумя основными соображениями:

- мной движет желание сделать книгу доступной для читателей с любым опытом и не привязывать ее к конкретному языку программирования или парадигме;
- обобщенное описание выполняемых шагов и абстрагирование от низкоуровневых деталей позволяют сосредоточиться на сути алгоритмов, не беспокоясь о причудах языка программирования.

Весь смысл использования псевдокода в том, чтобы дать обобщенное, полное и простое для понимания описание алгоритмов; следовательно, если это сделать правильно, то не потребуется дальнейшего объяснения.

В то же время даже при использовании псевдокода необходимо следовать некоторым соглашениям и быть последовательными. Более того, читателям, не знакомым ни с этим подходом, ни с выбранными обозначениями или соглашениями, может потребоваться некоторое время, чтобы приноровиться и освоиться.

По этой причине было решено добавить краткое руководство, поясняющее обозначения, которые используются на протяжении всей книги. Если вы не испытываете сложностей при чтении псевдокода, то можете смело пропустить это приложение или обращаться к нему, когда почувствуете, что вам нужны пояснения по нотации.

А.1. ПЕРЕМЕННЫЕ И ОСНОВЫ СИНТАКСИСА

Как и в любых других языках программирования, первой фундаментальной возможностью является возможность сохранения и восстановления значений: известно, что, в отличие от языков низкого уровня, таких как ассемблер, использующих регистры для хранения данных, языки более высокого уровня поддерживают понятие *переменных*.

Переменные — это именованные ссылки на память, которые можно создать, присвоить им значения, а затем прочитать их.

Некоторые языки, будучи строго типизированными, требуют явно указывать тип значений, которые сможет принимать переменная (например, целые числа или строки). Другие языки, свободно типизированные, позволяют сохранять в переменных любые значения, не ограничивая их одним типом. Подход со свободной типизацией также избавляет от необходимости объявлять переменные перед использованием: переменная автоматически создается первой операцией присваивания (но обратите внимание, что попытка прочитать ее перед присваиванием во всех языках со свободной типизацией приводит к ошибке).

Для псевдокода использование слабо типизированного подхода вполне естественно; а для нас, как уже упоминалось, желательно максимально абстрагироваться от деталей реализации.

Еще один открытый вопрос для переменных (функций и т. д.) — соглашение об именах. Это является (обычно) не непосредственным требованием языка программирования, а, скорее, соглашением, достигнутым в сообществе.

В книге применяется *верблюжьяНотация* для имен, но не по какой-то технической причине или из-за предпочтений в отношении какого-то одного языка программирования, а просто потому, что в такой нотации используется меньше символов, чем в *змеиной_нотации*, а это, в свою очередь, помогает вместить строки кода по ширине книжной страницы.

Так, например, я использую имя *doAction* вместо *do_action* для функций и что-то вроде *RedBox* для классов или объектов.

Переменные поддерживают несколько основных операций:

- *присваивание значения* обозначается стрелкой, указывающей влево; например, инструкция `index` ← 1 обозначает присваивание значения `1` переменной с именем `index`;
- *чтение значения переменной* обозначается простым упоминанием имени переменной; имя переменной будет замещаться ее значением при выполнении; например, инструкция `index` ← `size` означает чтение значения переменной `size` и присваивание его переменной `index`;

- *сравнение значений*:
  - для обозначения сравнения двух переменных или значений[1] используется пара знаков равенства: `index == size` или `index == 1` — это два примера, первый сравнивает две переменные, а второй сравнивает переменную и значение;
  - определение неравенства обозначается как <> или !=: `index <> size`.

Для сравнений «меньше», «больше» и т. д. применяются стандартные операторы: `index <= size`, например, оценивается как `true`, если значение переменной `index` меньше значения переменной `size` или равно ему.

A.2. МАССИВЫ

Массивы можно считать частным случаем переменных, потому что есть возможность присваивать/читать массивы целиком или поэлементно. Если вы не знакомы с массивами и контейнерами, загляните в приложение B.

В нашем псевдокоде мы абстрагируемся от деталей реализации массивов, типичных для конкретных языков программирования. В частности, массивы считаются динамическими структурами данных, не требующими явно выделять место для элементов перед доступом к ним.

Некоторые языки программирования изначально поддерживают концепцию динамических массивов; другие предоставляют их приблизительное подобие в своих стандартных библиотеках, часто в виде структуры данных, называемой вектором.

Кроме того, в одних языках массивы могут хранить элементы только одного типа, определяемого при создании массива, а в других — любые значения независимо от типа. Обычно это определяется строгой или слабой типизацией языка программирования.

Как упоминалось в предыдущем разделе, мы абстрагируемся от типов переменных в псевдокоде, поэтому, естественно, предполагается последний подход. Однако для большинства структур данных нам необходимы только массивы и контейнеры, содержащие элементы исключительно одного типа, и, если явно не указано иное, можно с уверенностью предположить, что это именно так. Просто помните, что массивы с элементами разных типов вполне возможны в псевдокоде, но в реальных программах для их поддержки придется приложить дополнительные усилия.

Для обозначения доступа к элементам массивов используются квадратные скобки, как и в большинстве языков программирования: $A[i]$ обозначает i-й элемент массива A, и аналогично, чтобы присвоить значение этому элементу, используется

[1] В общем случае сравниваются два выражения, состоящие из переменных и значений.

обычный синтаксис для переменных, что-то вроде $A[j] \leftarrow b$ (присваивает j-му элементу массива A значение переменной b).

Всякий раз, когда не указано иное, предполагается, что индексация элементов массива начинается с 0, поэтому $A[0]$ соответствует первому элементу в массиве A, $A[1]$ — второму и т. д.

Нам также иногда приходится выполнять определенные операции с массивами (и с контейнерами в целом). Например, может понадобиться найти максимальное значение в массиве чисел, или самое длинное слово в массиве строк, или сумму значений в числовом массиве.

Для таких операций используется что-то более близкое к математической нотации. Например, для массива A:

- `max{A}` обозначает максимальное значение в A;
- `sum{A}` — сумма всех элементов в A;
- `sum{x ∈ A | x > 0}` — сумма положительных элементов в A;
- $A[i] \leftarrow i^3\ (\forall\ i \in \{0..|A| - 1\} \mid i\ \%\ 3 == 0)$ — всем элементам в A с индексами, кратными 3, присваивает значение, равное кубу индекса.

Разумеется, в зависимости от выбранного языка программирования эти конструкции могут не иметь прямого аналога, умещающегося в одну инструкцию, и тогда придется написать и использовать вспомогательный метод.

А.3. УСЛОВНЫЕ ИНСТРУКЦИИ

Следующие фундаментальные понятия, которые понадобятся для написания осмысленной программы, — условные операторы. Большинство языков программирования реализуют условные операторы в форме конструкции *if-then-else*, поэтому и я в книге применяю аналогичный синтаксис.

Например, чтобы выразить простой алгоритм вычисления абсолютного значения числа `x` и сохранения его в другой переменной `y`, в случае псевдокода используется следующее написание:

```
if x < 0 then
  y ← -x
else
  y ← x
```

Сами условия не заключаются в круглые скобки, а для ограничения блоков не используются фигурные скобки (см. раздел А.4). Обратите также внимание, что ключевые слова выделяются жирным шрифтом.

Иногда ветвь `else` не нужна, например, когда необходимо выполнить действие, только если условие выполнено, и ничего не делать в противном случае. Так, на рис. А.1

показана альтернативная версия алгоритма получения абсолютного значения, присваивающая абсолютное значение `x` самой переменной `x`; это позволяет упростить исходный код до:

```
if x < 0 then
  x ← -x
```

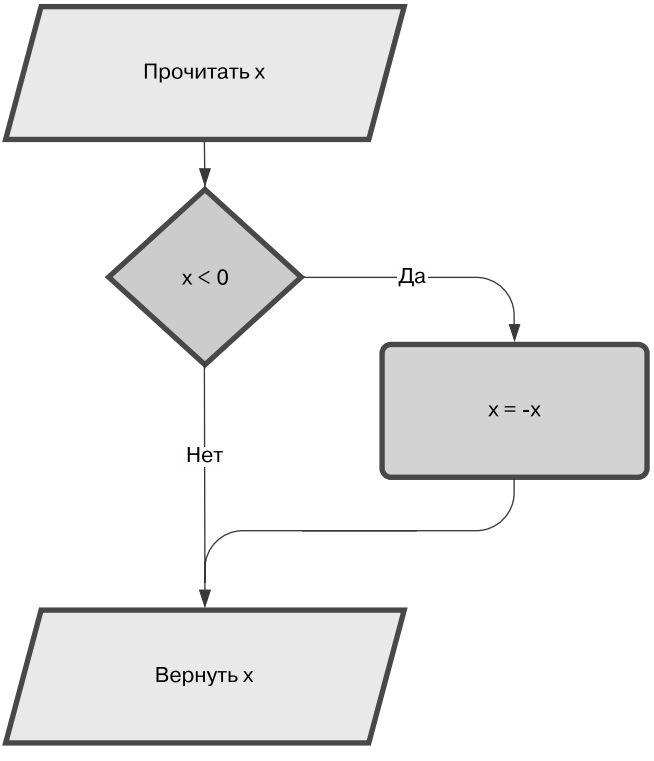


Рис. А.1. Блок-схема возможного алгоритма вычисления абсолютного значения

A.3.1. else-if

В других случаях приходится выбирать между тремя и более альтернативами. В таких случаях можно, конечно, реализовать логику, используя несколько вложенных операторов `if-then-else`, но это сделает код громоздким и трудно читаемым (из-за отступов).

Более компактное решение основано на использовании варианта `else-if`, распространенного в языках сценариев, таких как Python и Ruby.

Вот пример использования ключевого слова `elsif`:

```
if x < 0 then
  y ← -1/x
```

```
elsif x > 0 then
  y ← 1/x
else
  y ← NaN
```

Между прочим, этот фрагмент кода вычисляет абсолютное значение обратной величины числа `x`, а значение `NaN` означает not a number («не число») и является лучшим выбором для выражения того факта, что `1/0` не определено[1].

A.3.2. switch

Многие языки программирования поддерживают специальные инструкции для проверки условия, которое может принимать несколько дискретных значений. Обычно эта инструкция называется `switch`. Она позволяет перечислить возможные значения выражения и для каждого выполнить соответствующую ветвь `case` (плюс ветвь `default`, если фактически полученное значение не соответствует ни одному из предусмотренных вариантов):

```
switch x² + y²
  case 0
    label ← "origin"
  case 1
    label ← "unit circle"
  default
    label ← "circle"
```

Этот фрагмент кода вычисляет выражение $x^2 + y^2$: каждый оператор `case` соответствует своей ветке, выполняемой, когда результатом выражения является значение, следующее за ключевым словом, поэтому, если $x^2 + y^2$ дает в результате 0, то выполняется код после первого оператора `case` и переменная `label` получает значение `"origin"` (начало координат); аналогично, если в результате получается 1, то переменная `label` получает значение `"unit circle"` (единичная окружность). Для любого другого значения выполняется код после ключевого слова `default`, а переменная `label` получает значение `"circle"` (круг).

Мы используем расширенную версию этой инструкции, также позволяющую указывать диапазоны в операторах `case`:

```
switch x² + y²
  case 0
    label ← "origin"
  case [0,1]
    label ← "inside unit circle"
  case {1..10}
    label ← "multiple of unit circle"
  default
    label ← "outside unit circle"
```

[1] Другой, менее точной с точки зрения математики альтернативой могло бы быть присваивание значения inf (обозначающего бесконечность), но, поскольку знак предела зависит от направления, с которого мы приближаемся к 0, ни +inf, ни –inf не будут правильными.

В этом фрагменте кода используется четыре разных правила сопоставления:

- первое — проверка соответствия ровно одному значению (0);
- второй — проверка соответствия любому числу от 0 до 1 включительно;
- третье — проверка соответствия любому целому числу от 1 до 10 включительно;
- последнее — проверка соответствия вообще любому значению (по умолчанию).

Обратите внимание, что правила проверяются сверху вниз и выбирается первое выполнившееся. Например, даже притом что значение 0 для $x^2 + y^2$ соответствует как первому, так и второму (и четвертому!) предложению `case`, выполнено будет только первое из них.

Наконец, здесь были показаны примеры выражений, вычисляющих числовые значения, но вообще возможны сопоставления со значениями любых других типов (например, со строками).

A.4. ЦИКЛЫ

Другими фундаментальными управляющими конструкциями являются циклы.

Традиционно языки программирования предоставляют как минимум два вида циклов:

- циклы `for`, явно перебирающие некоторые индексы или элементы в контейнере;
- циклы `while` — более общие, проверяющие условие продолжения итераций.

Существует несколько вариантов этих двух видов циклов (например, `do-while`, `for-each` и `repeat-until`), но все эти циклы можно реализовать в терминах базового цикла `while`.

Мы используем цикл `for-each` и базовый цикл `while`.

В обоих случаях можно контролировать выполнение цикла с помощью ключевых слов `continue` (выполняющего переход к следующей итерации в цикле) и `break` (выполняющего немедленный выход из цикла без проверки условия).

Понять их работу проще на примерах.

A.4.1. Цикл for

Для иллюстрации цикла `for` посмотрим, как суммировать все элементы в массиве `A`. Есть как минимум три способа сделать это.

Первый — обойти все элементы массива, от первого до последнего:

```
n ← length(A)
total ← 0
for i in {0..n-1} do
  total ← total + A[i]
```

После ключевого слова `for` указывается имя переменной (`i`), хранящей значения, по которым выполняются итерации; затем следует ключевое слово `in`, а после него (упорядоченный) список значений, которые может принимать переменная `i` (в данном случае все целые числа от `0` до `n-1` включительно).

Поскольку для доступа к текущему элементу массива нужен только индекс `i`, тот же результат можно получить более кратким способом, перебирая элементы `A` напрямую:

```
total ← 0
for a in A do
  total ← total + a
```

Наконец, как было показано в предыдущем разделе, мы также используем математическую нотацию вместо цикла `for`, чтобы сделать код еще более кратким не в ущерб ясности:

```
total ← sum{A}
```

Последние два варианта, очевидно, не всегда жизнеспособны. Например, если понадобится применить сложные выражения, использующие сразу несколько элементов массива, то может потребоваться явно выполнить итерации по индексам:

```
n ← length(A)
total ← A[0]
for i in {1..n-1} do
  total ← total + A[i] + A[i-1]2
```

A.4.2. Цикл while

Как уже упоминалось, циклы `while` предназначены для более общих случаев использования. Как таковые их можно применять для реализации той же логики с циклами `for`, что представлена в примерах в предыдущем разделе. Вот эквивалент цикла `for` для вычисления суммы элементов массива:

```
n ← length(A)
i ← 0
total ← 0
while i < n do
  total ← total + A[i]
  i ← i + 1
```

Очевидно, что синтаксис `for-each` позволяет выразить ту же логику меньшим количеством кода (а также, что важно, он инкапсулирует логику перебора индексов, это делает его менее подверженным ошибкам, чем цикл `while`, где приходится явно инициализировать и увеличивать `i`).

Однако циклы `while` позволяют писать условия, которые трудно или невозможно выразить с помощью операторов `for`:

```
while not eof(file) do
  x ← read(file)
  total ← total + x
```

Приведенный фрагмент кода абстрагирует процесс чтения целых чисел из файла и их суммирования до достижения конца файла (`eof`).

В качестве более конкретного примера рассмотрим алгоритм вычисления *наибольшего общего делителя* (НОД) двух целых чисел `a` и `b`:

```
while a <> b do
  if a > b then
    a ← a – b
  else
    b ← b - a
```

В конце цикла переменная `a` будет хранить НОД для `a` и `b`.

Как видите, в обоих примерах нужно вычислять предикат, который трудно выразить при использовании цикла `for-each`.

A.4.3. break и continue

Иногда бывает нужно проверить несколько условий или отреагировать на условия, которые можно оценить только внутри тела цикла. Для этого можно использовать `break` и `continue`. Например, вот как в примере суммирования чисел в файле можно пропустить нечетные числа и использовать `0` в качестве маркера, чтобы прекратить суммирование чисел, как только будет встречен `0`:

```
while not eof(file) do
  x ← read(file)
  if x % 2 == 1 then
    continue
  elsif x == 0 then
    break
  total ← total + x
```

Всякий раз, когда очередное число, прочитанное из файла, является нечетным, выполняется переход к следующей итерации цикла (то есть снова будет проверен предикат, помогающий узнать — не достигнут ли конец файла).

Если встречено число `0`, выполнение цикла прерывается до увеличения суммы.

Оба ключевых слова также можно использовать внутри циклов `for`.

A.5. БЛОКИ И ОТСТУПЫ

До сих пор в большинстве примеров циклы и ветви условных выражений состояли ровно из одной инструкции, и это делало синтаксис довольно простым. Однако в общем случае каждая ветвь оператора `if-then-else` может включать произвольное количество инструкций. Чтобы избежать двусмысленности, нужно иметь возможность группировать инструкции в *блоки*.

В самом простом определении блок — это последовательность инструкций, выполняемых последовательно сверху вниз.

Как всегда, разные языки программирования по-разному определяют блоки: некоторые используют фигурные скобки для обозначения начала и конца блока (например, C, Java и т. д.), другие — ключевые слова `begin` и `end` (например, Pascal), а третьи — отступы (Python).

Более того, в некоторых языках программирования блоки могут иметь дополнительное значение, особенно если используется *область видимости блока*; в таких языках доступ к локальным переменным, определенным в блоке, возможен только внутри этого же блока кода (включая вложенные блоки).

Для простоты, как уже упоминалось, мы не заморачиваемся объявлением переменных, а просто предполагаем действие *области видимости функции*, и переменные становятся доступны везде внутри функции (как показано ниже). Предполагается также отсутствие поддержки *замыканий и лексическая область видимости* (иногда упоминаемая как статическая область видимости), согласно которой переменные прекращают свое существование по завершении функции.

Наконец, блоки определяются только отступами. Отступы уже были показаны выше, но следующий пример поможет внести дополнительную ясность:

```
for i in {0..n-1} do
  k ← i × 2
  j ← i × 2 + 1
  if i % 2 == 0 then
    A[k] ← 1
    A[j] ← 0
  else
    A[k] ← 0
    A[j] ← 1
A[2×n-1] ← 0
```

Цикл `for` выполняет `n` раз восемь строк, следующих за ним (все, кроме самой последней строки во фрагменте), и каждая ветвь оператора `if` содержит две инструкции (которые можно распознать по дополнительному отступу).

Строка с инструкцией `A[2×n-1] ← 0` не имеет отступа (она находится на том же уровне, что и первая строка), то есть это следующая инструкция, которая будет выполнена после завершения цикла `for`.

А.6. ФУНКЦИИ

Для группировки и повторного использования кода применяются функции. Функция определяет блок кода, в котором локальные переменные находятся в ограниченной области видимости. У функций есть сигнатура, объявляющая имя и параметры: переменные, служащие входными данными для функции. Наконец, функция также возвращает значение, которое фактически является ее выходным значением.

Разбивка кода на функции позволяет писать повторно используемый код, который проще читать и тестировать, потому что (в идеале) каждая функция может (и должна) реализовывать только одну задачу (или одно действие/алгоритм, если хотите).

Возьмем, к примеру, код в подразделе А.4.2, который вычисляет НОД двух чисел. Его можно преобразовать в функцию:

```
function gcd(a, b)
  while a <> b do
    if a > b then
      a ← a − b
    else
      b ← b − a
  return a
```

Обратите внимание, что, кроме прочего, теперь сразу видно, где хранится окончательный результат, но самое замечательное, что вызывающей стороне даже не нужно беспокоиться об этом, потому что функция сама позаботится о возврате правильного значения. Более того, переменные `a` и `b` существуют только внутри функции `gcd`, поэтому все, что происходит в теле этой функции, не повлияет на остальной код.

А.6.1. Перегрузка и аргументы по умолчанию

Параметры функций могут иметь значения по умолчанию:

```
function f(a, b=2)
  return a × b²
```

Например, функцию `f`, второй параметр которой имеет значение по умолчанию, можно вызвать с двумя аргументами, например: `f(5,3)` — или только с одним. В этом случае вызов, например, `f(5)` будет равносилен вызову `f(5,2)`.

Наличие значений для параметров по умолчанию допускает более компактный синтаксис перегрузки функций и методов.

Поскольку для псевдокода предполагается свободная типизация, это единственный вид перегрузки, который необходим в нашем случае и вообще может быть реализован (в то время как в строго типизированных языках, таких как C++ или Java, пришлось бы предусмотреть перегруженные версии функции, принимающие целочисленные аргументы, строковые...).

А.6.2. Кортежи

Иногда бывает нужно, чтобы функции возвращали более одного значения. Чтобы решить эту задачу, в псевдокоде предполагается, что функции могут возвращать кортежи.

Кортеж похож на массив, но имеет свои отличия:

- это список значений фиксированной длины (в то время как массивы могут увеличиваться или уменьшаться);
- его элементы могут иметь любой тип, и кортежи способны хранить элементы сразу нескольких типов.

Кортежи обозначаются круглыми скобками: `(1,2)` — это кортеж длиной 2 (он же *пара*), элементами которого являются числа со значениями `1` и `2`.

Кортежи значений можно присваивать кортежам переменных. Инструкция:

```
(x, y, z) ← (0, -1, 0.5)
```

эквивалентна следующему коду:

```
x ← 0
y ← -1
z ← 0.5
```

Аналогично можно записать инструкцию:

```
(name, age) ← ("Marc", 20)
```

Этот синтаксис может очень пригодиться для реализации функций, возвращающих несколько значений. Предположим, мы написали функцию `min_max`, возвращающую максимальное и минимальное значения из массива; тогда можно предположить, что она возвращает пару значений, и вызывать ее следующим образом:

```
(a_min, a_max) ← min_max(A)
```

A.6.3. Кортежи и деструктурирование объектов

Рекомендуется избегать неименованных кортежей, потому что в отсутствие имен полей трудно судить об их предназначении и приходится ориентироваться только на позиции внутри кортежа. Намного лучше использовать объекты (например, объект с полями `min` и `max` был бы понятнее в примере из предыдущего подраздела).

Кортежи, однако, обеспечивают жизнеспособную альтернативу, когда назначение полей достаточно очевидно. Чтобы объединить сильные стороны объектов и кортежей, используется специальная нотация для присваивания всех или части полей объекта кортежу.

Предположим, есть объект `Employee` с полями `name`, `surname`, `age`, `address` и т. д.

Если представить, что `empl` является экземпляром `Employee`, то можно использовать следующий синтаксис для извлечения любого подмножества полей из `empl` в псевдонимы[1]:

```
(name, surname, age) ← empl
```

В этом примере извлекаются только три поля.

Этот синтаксис особенно удобен в сочетании с циклами `for-each`, потому что позволяет перебирать коллекцию сотрудников и напрямую обращаться к псевдонимам

[1] Псевдоним — это просто другое имя для переменной: его можно реализовать, создав новую переменную, просто ссылающуюся на исходную.

нужных полей, без необходимости писать что-то вроде `empl.name` каждый раз, когда нужно обратиться к полю объекта `empl`. Вот сравнительный пример, поясняющий разницу:

```
for empl in employees do
  user ← empl.surname + empl.age

for (surname, age) in employees do
  user ← surname + age
```

Б Нотация «О большое»

В этом приложении поговорим о модели памяти и известной математической нотации «О большое», чтобы вы могли увидеть, что она не так плоха, как о ней иногда говорят. Не волнуйтесь, для чтения книги вам будет достаточно даже поверхностного понимания.

Б.1. АЛГОРИТМЫ И ПРОИЗВОДИТЕЛЬНОСТЬ

Описание производительности алгоритма — нетривиальное упражнение. Возможно, вы привыкли описывать производительность кода с помощью эталонных тестов, и этот подход проявил себя простым и доступным. Но когда возникает необходимость эффективно описать ее, чтобы результаты можно было представить другим, то приходится решать больше проблем, чем изначально предполагалось.

Например, что вообще подразумевать под производительностью? Обычно этим словом обозначается какая-то мера, но какая?

Первый ответ, который напрашивается сам собой: интервал времени, необходимый для выполнения алгоритма с определенными входными данными. Возможно, даже вы осознаете, что может потребоваться усреднить это время по нескольким наборам входных данных и нескольким прогонам с одними и теми же данными, чтобы учесть влияние внешних факторов. Но все равно это измерение содержит случайный шум, что особенно характерно для современных многоядерных и многозадачных архитектур. Сложно проводить эксперименты в изолированном окружении, и на результаты влияют как сама операционная система, так и фоновые процессы.

Но это еще не самое худшее. На результат в значительной степени влияет аппаратное обеспечение, на котором проводятся эксперименты, поэтому он не будет иметь никакого значения как абсолютное число.

Можете попробовать запустить контрольный тест и сравнить производительность вашего алгоритма с каким-то другим известным алгоритмом. Иногда этот метод помогает получить значимые результаты, но и это не лучший путь, потому что попрежнему придется принимать во внимание слишком много переменных факторов, от аппаратного обеспечения и операционной системы, где производится тестирование, до типа и размера входных данных, используемых в тестах.

Продолжая рассуждения, можно предложить на роль оценки производительности подсчет выполняемых инструкций. Вероятно, это хорошая оценка времени, необходимого для выполнения всего алгоритма, и достаточно обобщаемая, верно?

Верно... но не совсем.

1. Собираетесь ли вы подсчитывать машинные инструкции? Если да, то эти подсчеты будут зависеть от платформы.
2. Если подсчитать только высокоуровневые инструкции, то результат будет сильно зависеть от выбранного языка программирования. Scala или Nim могут быть гораздо более лаконичными, чем Java, Pascal или COBOL.
3. Что можно считать лучшим? Имеет ли значение, если алгоритм выполняет 99 инструкций вместо 101?

Я мог бы продолжать и продолжать, но вы уже наверняка поняли суть. Проблема в том, что мы с вами слишком много внимания уделяем деталям, не имеющим большого значения. Путь к выходу из этого тупика — абстрагирование от этих деталей. Можно, скажем, определить упрощенную вычислительную модель и *модель памяти*. Последняя представляет собой набор основных операций, выполняемых с внутренней *оперативной памятью*.

Б.2. МОДЕЛЬ ПАМЯТИ

В рамках модели памяти делается несколько допущений:

- каждая элементарная операция (арифметические операции, `if` или вызов функции) занимает ровно один шаг (далее просто *шаг*);
- циклы и подпрограммы не считаются элементарными операциями, рассматриваются как состоящие из множества элементарных операций и потребляющие ресурсы пропорционально количеству этих операций;
- каждое обращение к памяти занимает ровно один шаг;
- память считается бесконечной.

Последний пункт может показаться немного нереалистичным, но обратите внимание, что в этой модели не делаются различия между доступом к кэшу, оперативной памяти, жесткому диску или хранилищу в центре обработки данных. Другими словами:

- для большинства реальных приложений есть возможность обеспечить необходимый объем памяти;
- при таком предположении можно абстрагироваться от деталей реализации памяти и сосредоточиться на логике.

В модели памяти производительность алгоритма измеряется количеством шагов, которые он выполняет при заданных входных данных.

Эти упрощения позволяют абстрагироваться от деталей платформы. Для некоторых задач или их вариантов детали могут иметь значение. Например, возможно различать доступ к кэшу и доступ к диску. Но в целом это очень неплохое обобщение.

Теперь, определив используемую модель, обсудим, что будет считаться важным улучшением алгоритма.

Б.3. ПОРЯДОК ВЕЛИЧИНЫ

Другой аспект, который желательно упростить, — способ подсчета инструкций. Можно договориться учитывать только некоторые виды операций, например доступ к памяти. Или, как при анализе алгоритмов сортировки, учитывать только количество перестановок элементов.

Кроме того, небольшие вариации в количестве выполняемых шагов вряд ли имеют значение. Скорее, можно рассуждать об изменениях в разы или на порядки: 2×, 10×, 100× и т. д.

Но, чтобы действительно понять, когда один алгоритм работает лучше, чем другой в данном экземпляре задачи, нужно еще кое-что: выразить количество выполненных шагов как функцию от размера задачи.

Предположим, мы с вами сравниваем два алгоритма сортировки и утверждаем, что «в заданном наборе тестов первый алгоритм выполняет 100 шагов, а второй — 10». Действительно ли это поможет нам предсказать производительность обоих алгоритмов на другом экземпляре задачи?

И наоборот, если мы сможем доказать, что после получения массива с n элементами алгоритму А потребуется выполнить $n \times n$ перестановок, а алгоритму Б — только n, то у нас будет очень хороший способ предсказать, как каждый из них будет работать с данными любого объема.

И здесь в игру вступает нотация «О большое». Но сначала взгляните на рис. Б.1, чтобы получить представление о сравнении времени выполнения двух упомянутых алгоритмов. Эта диаграмма должна помочь вам понять, почему предпочтительнее алгоритм Б.

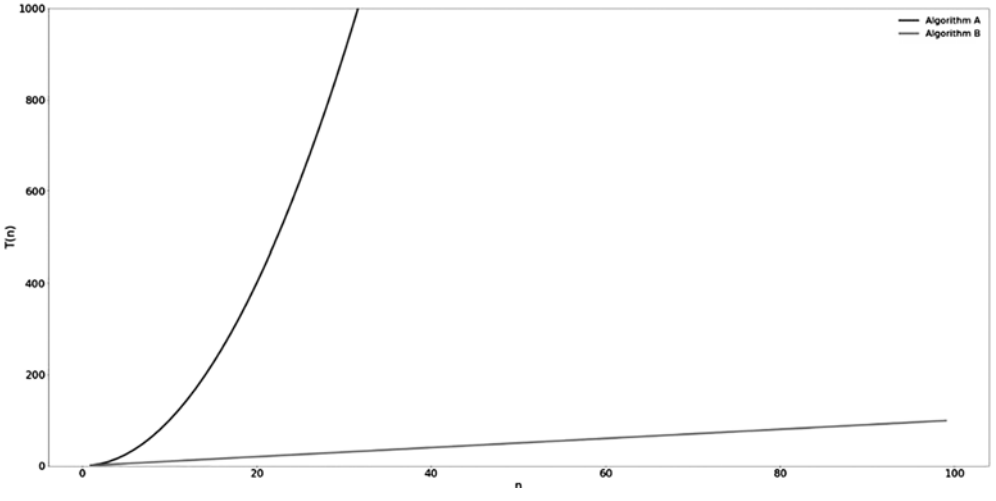


Рис. Б.1. Визуальное сравнение времени выполнения квадратичного (А) и линейного (Б) алгоритмов. Первый растет настолько быстрее, что на диаграмме удалось показать график роста времени выполнения алгоритма А (изогнутая линия) только до[1] n ~= 30, хотя максимальное значение оси Y в 10 раз превышает максимальное значение, достигнутое алгоритмом В

Б.4. НОТАЦИЯ

В обозначении нотация «О большое» O — это математическая форма описания роста определенных величин с ростом размера входных данных. Обычно она записывается в виде заглавной буквы O, заключающей выражение в круглых скобках: что-то вроде O($f(n)$), отсюда и название «О большое». $f(n)$ может быть любой функцией с аргументом n; мы рассматриваем только функции с целыми числами, потому что n обычно описывает размер некоторых входных данных.

Не стану подробно описывать здесь нотацию «О большое»; это выходит за рамки темы книги. Но есть много учебников с углубленным описанием этой темы. Основная идея, которую нужно запомнить: запись O($f(n)$) выражает границу.

[1] Здесь и в тексте книги символы ~= будут означать «примерно равно».

Математически выражение $g(n) = O(f(n))$ означает, что для любого достаточно большого входного значения существует вещественная константа c (значение которой не зависит от n) такая, что $g(n) = c \times f(n)$ для каждого возможного значения n (вероятно, даже большего, чем некоторое значение n_0).

Например, если $f(n) = n$ и $g(n) = 40 + 3 \times n$, то мы можем выбрать $c = 4$ и $n_0 = 40$.

На рис. Б.2 показано, как эти три функции растут относительно друг друга. Хотя $f(n)$ всегда меньше, чем $g(n)$, величина $c \times f(n)$ в какой-то момент становится больше, чем $g(n)$. Чтобы лучше понять поворотный момент, можно вычислить две формулы, подставив фактические значения параметра n. Используем следующее обозначение $f(1) \rightarrow 1$, чтобы утверждать, что $f(1)$ оценивается как 1 (другими словами, результат вызова $f(1)$ равен 1).

Для n меньше 40 значение $g(n)$ будет больше. Для $n = 30$ значения $f(30) \rightarrow 120$ и $g(n) \rightarrow 130$. Вместо этого, как известно, $f(40) \rightarrow 160$ и $g(40) \rightarrow 160$, $f(41) \rightarrow 164$ и $g(41) \rightarrow 163$. Для любого значения больше 41 выполняется условие $40 + 3 \times n \leq 4 \times n$.

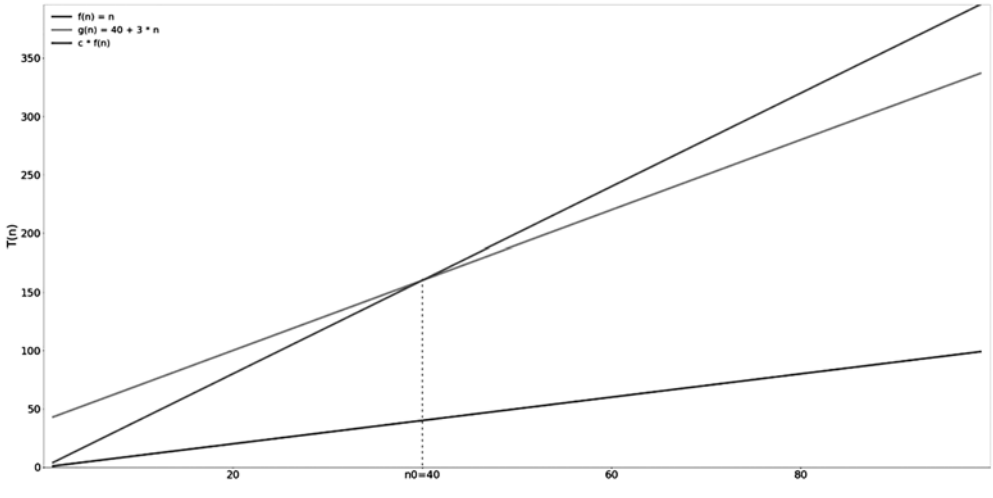


Рис. Б.2. Визуальное сравнение функций: f(n) = n, g(n) = 40 + 3 × n, 4 × f(n). Хотя f(n) всегда меньше, чем g(n), 4 × f(n) становится больше, чем g(n), при достаточно больших значениях n

Но не забывайте, что условие $g(n) \leq c \times f(n)$ должно выполняться для каждого $n \geq n_0$. Мы не можем строго доказать это, построив график или подставив конечное число значений n в формулу (контрпример приводится на рис. Б.3), поэтому придется прибегнуть к некоторым математическим выкладкам; тем не менее построение графиков функций поможет понять, как они растут и движетесь ли вы в правильном направлении.

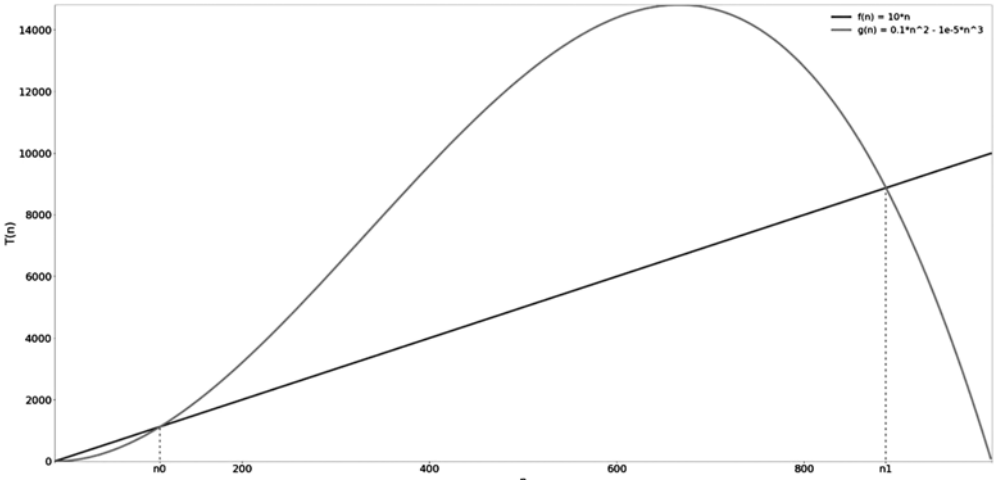


Рис. Б.3. Пример, показывающий, почему следует быть осторожными, делая выводы. Хотя g(n) больше, чем f(n), при n0~=112, это верно не для всех значений $n > n_0$. На самом деле при n_1 ~= 887 имеется еще одно пересечение двух функций, и после этого f(n) снова становится больше, чем g(n)

Вернемся к рассматриваемому примеру. Если мы говорим, что производительность алгоритма A равна $O(n^2)$, то это означает, что $T(n) = O(n^2)$, где $T(n)$ — время работы алгоритма. Или, другими словами, алгоритму A никогда не потребуется больше квадратичного числа шагов.

Определение $O(n)$ имеет некоторые следствия. Приведу их.

- «Для любого достаточно большого объема входных данных». Это важная часть определения. Интересно, как ведут себя функции, когда n становится (очень) большим, и не стоит заботиться о том, выполняются ли неравенства при малых значениях n. Подумайте, например, о функциях $f(x) = e^x$ и $g(x) = e \times x$. $f(x) < g(x)$, когда x меньше 1, но для больших значений $f(x)$ растет намного быстрее.
- Постоянные множители не имеют значения: $O(n) = O(3 \times n) = O(100 \times n)$. Это можно доказать, выбрав правильные значения константы в предыдущем неравенстве.
- Некоторые задачи требуют постоянного времени: подумайте о суммировании первых n целых чисел. Простейшему алгоритму потребуется выполнить $n - 1$ сложений, но если использовать метод Гаусса, то понадобится только одно сложение, одно умножение и одно деление, независимо от результата.

  В таких случаях производительность можно выразить как $O(c) = O(1)$ для любой положительной константы c.

  $O(1)$ обозначает постоянное время выполнения: другими словами, время не зависит от объема входных данных.

- При суммировании выражений с «О большое» О выбирается наибольшее: $O(f(n) + g(n)) = O(f(n))$, если $g(n) = O(f(n))$. То есть если два алгоритма запускаются последовательно, то в общем времени выполнения преобладает самый медленный.
- $O(f(n)) \times g(n))$ нельзя упростить, если ни одна из функций не является константой.

ПРИМЕЧАНИЕ

Часто, обозначая время выполнения алгоритма с помощью нотации «О большое», мы подразумеваем, что граница одновременно является и нижней, и верхней, если явно не указывается обратное. Тем не менее $\Theta(f(n))$ было бы правильным обозначением класса функций, верхняя и нижняя границы которых равны $f(n)$.

Теперь у нас есть все необходимые инструменты для однозначного описания производительности алгоритмов. В оставшихся разделах этого приложения рассмотрим примеры практического применения этой нотации.

Б.5. ПРИМЕРЫ

Если вы впервые видите нотацию «О большое», то совершенно нормально чувствовать недопонимание. Требуется некоторое время и практика, чтобы привыкнуть к ней.

Рассмотрим несколько примеров и попробуем выразить знания, полученные в предыдущем разделе, в математических терминах.

Предположим, у вас есть четыре числа, которые нужно сложить вместе: {1, 3, –1,4, 7}. Сколько операций сложения нам понадобится? Итак, посмотрим:

$1 + 3 + (-1,4) + 7$

По всей видимости, можно обойтись тремя сложениями. А если понадобится сложить десять чисел? Легко убедиться, что для этого потребуется девять сложений. Можно ли обобщить это количество операций для любого объема входных данных? Да, можно, потому что легко доказать, что для вычисления суммы n чисел потребуется $n - 1$ сложений.

То есть резонно утверждать, что суммирование n чисел (асимптотически) требует $O(n)$ операций. Если обозначить через $T(n)$ количество необходимых операций, то можно сказать, что $T(n) = O(n)$.

Рассмотрим два варианта этой задачи:

- нужно просуммировать первые пять элементов списка дважды, а остальные — один раз;
- нужно просуммировать квадраты элементов.

В первом случае для списка {1, 3, –1,4, 7, 4, 1, 2} имеем:

$1 + 3 + (-1,4) + 7 + 4 + 1 + 2 + \underline{1 + 3 + (-1,4) + 7 + 4}$

Здесь можно заметить повторяющиеся элементы (они подчеркнуты в формуле). Итак, для $n = 7$ нам понадобится выполнить 11 операций.

В общем случае понадобится $n + 4$ операции, а $T_1(n) = n + 4$.

Верно ли, что $T_1(n) = O(n)$? Вернемся к определению. Нужно найти две константы, целое число n_0 и действительное число c такие, что:

$$T_1(n) \le c \times n \text{ для каждого } n > n_0$$
$$\Leftrightarrow$$
$$n + 4 \le c \times n \text{ для каждого } n > n_0$$
$$\Leftrightarrow$$
$$c \ge 1 + 4 / n \text{ для каждого } n > n_0$$

Поскольку $4 / n$ уменьшается с ростом n, мы можем выбрать $c = 2$ и $n_0 = 4$ или, скажем, $c = 100$ и $n_0 = 1$, и неравенство будет выполнено.

То же верно для суммирования квадратов: для n чисел потребуется n умножений (или возведений в квадрат) и $n - 1$ сложений, поэтому $T_2(n) = 2n - 1$.

Можно доказать, что $T_2(n) = O(n)$, при $c = 2{,}5$ и $n_0 = 1$.

Теперь мы в силах также доказать, что $T_1(n) = O(n^2)$. Я оставляю поиск подходящих значений c и n_0 вам как самостоятельное упражнение. Однако эта граница не будет строгой: другими словами, существуют другие функции от n, такие что $T_1(n) \le f(n) < O(n^2)$ для достаточно больших n.

Точно так же верно было бы утверждение, что $T_1(n) = O(n^{1000})$. И это, безусловно, было бы правдой, но насколько полезным стало бы знание того, что нашему алгоритму требуется время, чуть меньшее, чем возраст Вселенной, для обработки небольшого объема входных данных?[1]

Как нетрудно догадаться, обычно интерес представляют строгие границы. Хотя это верно и ожидаемо и для наших примеров суммирования легко доказать, что граница $O(n)$ является строгой, тем не менее вы наверняка удивитесь, узнав, что есть алгоритмы, для которых неизвестна строгая граница или по крайней мере ее нельзя доказать.

В качестве последнего примера рассмотрим задачу перечисления всех пар чисел в списке, независимо от их порядка. Например, для списка {1, 2, 3} у нас есть пары (1, 2), (1, 3), (3, 2).

Получается, что количество неупорядоченных пар для n элементов равно $T_3(n) = n \times (n - 1) / 2$:

$$T_3(n) = n \times (n - 1) / 2 \le 1 \times n^2 \text{ для } n > 1.$$

[1] Оно может пригодиться, если понадобится доказать, что алгоритм действительно выполнится за конечное время. Но в реальных приложениях такое знание обычно не имеет практического применения.

То есть можно утверждать, что $T_3(n) = O(n^2)$. На этот раз квадратичная граница также является строгой, как показано на рис. Б.4.

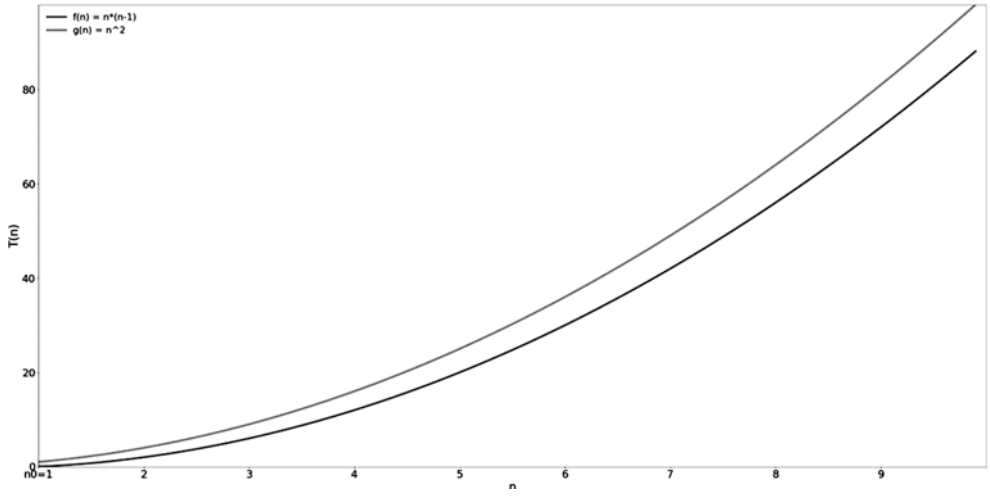


Рис. Б.4. Визуальное сравнение функций f(n) = n × (n – 1) и g(n) = n^2. Последняя всегда больше для любого положительного значения n

B Основные структуры данных

Нельзя построить дом, начав с крыши. Точно так же нельзя получить знания о сложных структурах данных, не освоив основы.

В этом приложении дается обзор основных структур данных и некоторых из наиболее широко используемых алгоритмов.

Сначала приведу самые основные структуры данных: *массивы*, *связанные списки*, *деревья* и *хеш-таблицы*. Я полагаю, что читатели уже знакомы с ними; в конце концов, эти структуры являются строительными блоками всех остальных структур. Тем не менее рассмотрим их еще раз, просто чтобы освежить память.

В последнем разделе кратко сравним эти структуры данных. Для каждой я перечислю ключевые особенности (например, поддерживают ли они порядок, являются ли статическими или динамическими) и обобщу их в таблице. Это поможет в будущем выбирать наиболее подходящие структуры для решения той или иной задачи, с которой мы с вами можем столкнуться.

B.1. ОСНОВНЫЕ СТРУКТУРЫ ДАННЫХ

Структуры данных являются одной из основ программирования, они постепенно разрабатывались с самого появления информатики.

В этом разделе я перечислю самые основные способы организации элементов данных в памяти, чтобы впоследствии элементы можно было извлекать в соответствии

с определенными критериями. Виды критериев, а также способы использования хранилища и выполнения основных операций (добавление, удаление и поиск) определяют характеристики структуры данных.

Приводимые основные структуры данных являются строительными блоками для реализации более мощных и сложных структур данных.

В.2. МАССИВ

Массивы — одна из самых простых и в то же время самых широко используемых структур данных. Как правило, это наборы данных одного типа, они поддерживаются большинством языков программирования. На самом низком уровне массив, выражаясь простым языком, — блок памяти, в котором элементы хранятся последовательно. Многие языки программирования поддерживают только *статические массивы*. Их размеры не могут меняться, и количество хранимых в них элементов должно быть определено при создании (или по крайней мере при инициализации) массива. *Динамические массивы*, напротив, могут увеличиваться при добавлении и уменьшаться при удалении элементов. На рис. В.1 показан пример такого массива. Динамические массивы поддерживаются некоторыми распространенными языками, например, в JavaScript массивы по своей сути динамические.

ПРИМЕЧАНИЕ

Можно доказать возможность реализации динамических массивов таким образом, чтобы в совокупности[1] вставка и удаление происходили так же быстро, как и для статических массивов.

Все элементы массива должны быть одного типа и занимать одинаковый объем памяти[2]. Когда это требование соблюдается, намного проще переходить от одного элемента к следующему, просто добавляя к адресу предыдущего элемента размер элемента массива.

---

[1] В анализе «О большого» под совокупным временем работы подразумевается ожидаемая производительность при достаточно большом количестве операций. Например, если (честную) монету подбросить только один раз, нельзя знать заранее, выпадет она орлом или решкой. Если подбросить ее дважды, то, возможно, один раз выпадет решка и один раз орел, но это не факт. Если повторить эксперимент миллион раз, не проделывая никаких трюков с монетой, то получим орел и решку почти в 50 % случаев. Точно так же для некоторых алгоритмов одна операция может занять больше времени, чем ожидалось, но если выполнить ее на большом объеме данных, то среднее время выполнения будет вполне предсказуемым.

[2] Низкоуровневая реализация массивов может различаться в разных языках программирования, по крайней мере теоретически.

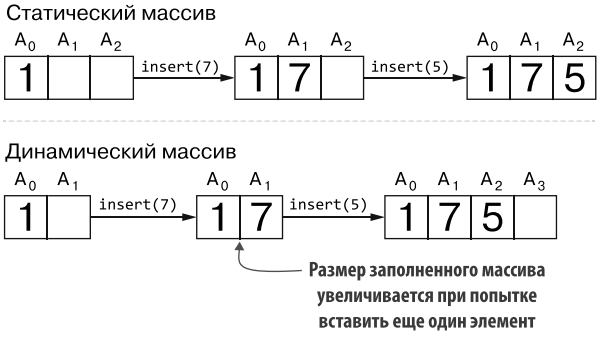


Рис. В.1. Пример вставки в статический *(вверху)* и динамический *(внизу)* массивы. Обратите внимание, что размер статического массива в верхней половине рисунка — постоянный (он всегда содержит три элемента), тогда как динамический массив в нижней половине первоначально имеет размер 2, который затем увеличивается (фактически удваивается) при добавлении элемента, переполняющего массив. В этом примере в статический массив нельзя добавить какой-либо другой элемент, не затерев имеющийся

ПРИМЕЧАНИЕ

Поскольку массивы размещены в одном блоке памяти, они с большей вероятностью будут поддерживать так называемую локальность ссылок. Например, при обходе массива следующие данные для чтения, скорее всего, будут находиться в той же странице памяти. Это может способствовать некоторым видам оптимизации (см. оригинальную статью[1] о принципе локальности).

Ключевое преимущество массивов — постоянное время доступа к любым элементам. Было бы точнее сказать, что к любому элементу массива можно получить доступ за постоянное время, прочитать или записать первый, последний или любой другой элемент, если известна его позиция (рис. В.2).

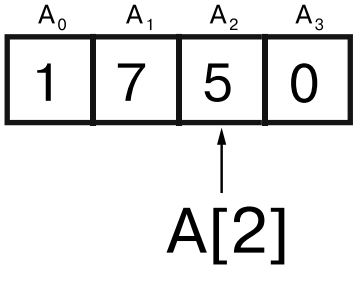


Рис. В.2. Извлечение третьего элемента массива. Если массив хранится в переменной с именем A, то A[2] обозначает третий элемент массива. Это связано с тем, что в компьютерах нумерация элементов массива начинается с 0

Несмотря на то что произвольный доступ является одной из сильных сторон массивов, другие операции с массивами выполняются медленнее. Как уже упоминалось,

[1] *Denning P. J., Schwartz S. C.* Properties of the Working-Set Model // Communications of the ACM, 1972. — Volume 15, Issue 3. March. — P. 191–198.

нельзя изменить размер массива, просто добавив или удалив элемент в конце. Каждый раз, когда в этом возникает необходимость, требуется перераспределить массив, если только он по своей природе не является динамическим. Чтобы амортизировать накладные расходы на изменение размера при большом количестве операций, можно сразу выделить больший кусок памяти и следить за количеством элементов.

Как будет показано в следующем разделе, списки позволяют ускорить вставку и удаление, но они медленнее работают, когда требуется произвольный доступ.

В.3. СВЯЗАННЫЙ СПИСОК

Связанный список хранит последовательность элементов, заключая каждый в объект, называемый узлом.

Как показано на рис. В.3, каждый узел содержит значение и одну или две ссылки на другие узлы.

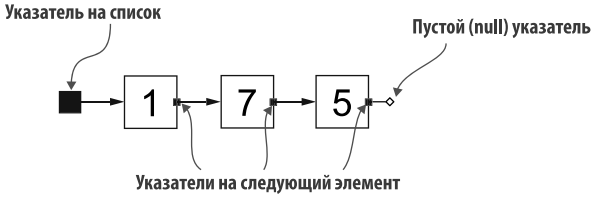


Рис. В.3. Графическое представление связанного списка. Здесь каждый элемент содержит указатель (ссылку) на следующий узел в списке. Ссылка на первый узел хранится в специальном указателе — голове списка. Чтобы просмотреть весь список, нужно проследовать по указателю в голове, а затем по ссылкам от узла к узлу

Значение может быть простого типа, как, например, число, или сложного, как, например, строка или объект. Порядок элементов определяется исключительно ссылками. Узлы не обязательно должны размещаться в непрерывном блоке памяти, поэтому списки по своей природе являются динамическими — они могут увеличиваться или уменьшаться по мере необходимости.

Более формально список можно определить рекурсивно (рис. В.4). Это может быть:

- пустой список;
- узел, содержащий значение и ссылку на связанный список.

На рис. В.5 показана другая важная особенность списков; списки могут быть:

- *односвязными* — каждый узел имеет только ссылку (или указатель) на следующий элемент;
- *двусвязными* — каждый узел хранит две ссылки: на следующий и на предыдущий элементы.

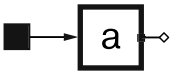
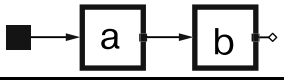
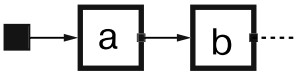
Пустой список	`[]`	
Одноэлементный	`a::[]`	
2 элемента	`a::(b::[])`	
n элементов	`a::(b::(c::...(x::[])))`	

Рис. В.4. Индуктивное определение списка. В среднем столбце показано формальное обозначение списка, а в правом — графическое представление того же списка. В формальном представлении рекурсивная природа более явная, потому что это действительно рекурсивная нотация. Пустой список считается примитивом, строительным блоком, и каждый непустой список рекурсивно определяется как первый элемент, за которым следует его хвост с оставшимися элементами, возможно, пустой список

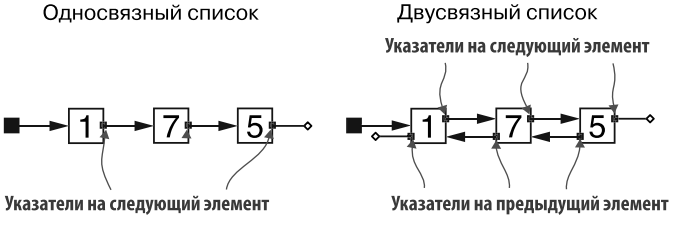


Рис. В.5. Односвязный *(слева)* и двусвязный *(справа)* списки

Как упоминалось выше, выбор между односвязными и двусвязными списками — это компромисс. Первый требует меньше памяти для хранения каждого узла, а второй позволяет использовать более быстрые алгоритмы для удаления элемента, но требует небольших накладных расходов для обновления указателей.

Чтобы иметь доступ к связанному списку, ссылка на его голову (первый узел) должна храниться в переменной. При добавлении или удалении элементов важно не забыть обновить эту ссылку, иначе может случиться так, что эта переменная-указатель будет ссылаться на внутренний узел или даже на недопустимую область памяти.

Вставка в списки может выполняться тремя способами.

- *В начало.* Это самый простой случай, и такая вставка выполняется за постоянное время. Иллюстрация на рис. В.6 показывает, что достаточно обновить ссылку на голову, записав в нее указатель на новый узел, и обновить ссылку в новом узле, записав в нее указатель на прежнюю голову списка (если только список не был пуст).

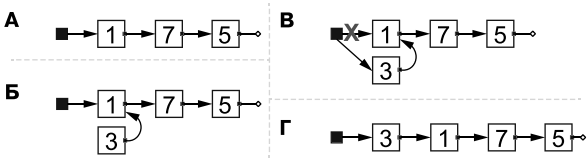


Рис. В.6. Вставка в начало связанного списка. *А*. Первоначальный список. *Б*. Создание нового узла, в котором в ссылку на следующий узел записывается указатель на предыдущую голову. *В*. Корректировка указателя на голову. *Г*. Окончательный результат

- *В конец.* Не самое практичное решение, потому что требует обойти весь список, чтобы найти последний узел; взгляните на рис. В.7, чтобы понять почему. Можно, конечно, предусмотреть сохранение дополнительного указателя на конец списка, но это вызовет и дополнительные накладные расходы, потому что каждый метод, изменяющий список, должен будет проверять необходимость обновления этой ссылки.

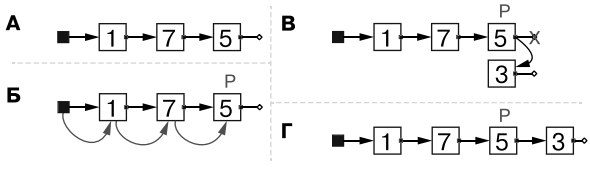


Рис. В.7. Вставка в конец связанного списка. *А*. Первоначальный список. *Б*. Обход связанного списка, чтобы достичь последнего элемента: назовем его P. *В*. Создание нового узла и обновление указателя на следующий элемент в P. *Г*. Окончательный результат

- *В любую другую позицию*. Такая вставка может потребоваться для поддержания определенного порядка следования элементов. Однако эта операция тоже обходится довольно дорого, потому что в худшем случае требует линейного времени, как показано на рис. В.8.

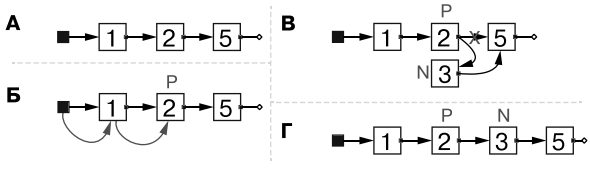


Рис. В.8. Вставка в произвольное место в связанном списке. В этом примере показана вставка в отсортированный связанный список (но сортировка не является обязательным требованием). *А*. Первоначальный список. *Б*. Обход связанного списка в поисках узла, предшествующего позиции вставки нового элемента (так как это односвязный список, в процессе обхода нужно сохранять указатель на предшествующий элемент). Обозначим этот предшествующий узел через P. *В*. Создание нового узла N и обновление указателя на следующий элемент в P. Запись в N.next прежнего значения P.next. *Г*. Окончательный результат

Те же аргументы за/против можно привести в отношении удаления узла.

- *В начале*. Можно выполнить за постоянное время, переписав в ссылку на голову указатель из ссылки на следующий узел из удаляемого элемента (после проверки, что список не пустой). Дополнительно необходимо гарантировать освобождение памяти, занятой удаленным узлом, или ее доступность для сборки мусора.
- *В конце*. Нужно найти предпоследний узел и обновить его указатель на следующий узел. В двусвязном списке достаточно взять последний элемент списка и перейти к предшествующему ему узлу. В односвязном списке в процессе обхода нужно сохранять ссылки на текущий и предшествующий ему узлы. Эта операция выполняется за линейное время.
- *В любой другой позиции*. Здесь верны те же соображения, что и для вставки, плюс необходимо позаботиться о сохранении ссылки на предпоследний элемент в списке.

Связанные списки — это рекурсивные структуры данных, что вытекает из их рекурсивного определения и означает возможность решения проблем со списками по индукции. Вы можете попробовать предоставить решение для базового случая (пустой список), а также варианты объединения некоторых действий с головой и хвостом (который представляет собой список меньшего размера). Например, получить алгоритм поиска максимального числа в списке можно так:

- если список пуст, вернуть `null`;
- если в списке есть хотя бы один элемент, взять голову списка (назовем ее x) и применить алгоритм к списку с оставшимися $N - 1$ элементами, получив значение y.

  Тогда в результате получится:

  - x, если y равен `null` или $x \geq y$;
  - иначе y.

B.4. ДЕРЕВО

Деревья — еще один широко используемый абстрактный тип данных, обеспечивающий иерархическую структуру, подобную спискам, но в разветвленной форме. Их действительно можно рассматривать как обобщение *связанных списков*: каждый узел все так же имеет одного предшественника, называемого *родителем*, но может иметь более одного преемника, которых называют *потомками*. Каждый *потомок* сам является поддеревом (пустым или включающим корень с его поддеревьями).

На рис. B.9 показано обобщенное дерево, но формально дерево можно определить как:

- пустое дерево:
- узел с одной или несколькими ссылками на дочерние элементы и (не обязательно) ссылкой на родителя.

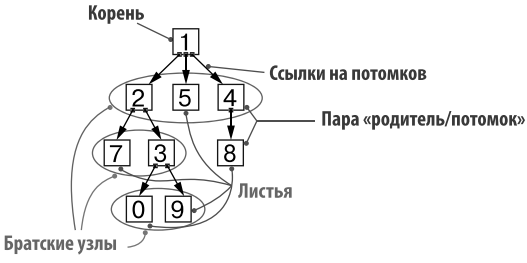


Рис. B.9. Пример дерева. Стрелки сверху вниз представляют связи между узлами и их потомками; эти стрелки всегда направлены от родителя к потомку. У каждого узла есть один и только один родитель. Узлы, имеющие одного и того же родителя, называются братскими. Между братскими узлами нет явной связи, поэтому доступ к братским узлам возможен только через их общего родителя. Узлы без потомков называются листьями

Каждый узел дерева, подобно узлам списка, содержит значение, но сверх того содержит также список ссылок на другие узлы — его потомков. Существует ограничение: каждый узел в дереве может быть потомком только одного другого узла, за исключением корня, у которого нет родителя (поэтому в дереве нет ни одного узла, указывающего на его корень). По этой причине ссылку на корень дерева нужно хранить отдельно (в переменной), как и в случае с односвязным списком, голова которого не связана с другими узлами. Более того, деревья определяют «вертикальную» иерархию, отношения «родитель — потомок», в то время как между одноуровневыми узлами или между узлами в разных поддеревьях не может быть никаких отношений.

Прежде чем двигаться дальше, уточним терминологию, используемую при работе с деревьями.

- Если узел `x` имеет два дочерних узла `y` и `z`, то `x` является *родителем* этих узлов. Соответственно, верно следующее: `parent(y) == parent(z) == x`. На рис. B.9 узел с номером `1` является родителем узлов с номерами `4`, `5` и `2`.
- В наборе узлов `x`, `y` и `z`, определенных здесь, `y` и `z` называются *братскими* узлами.
- *Корень* — это единственный узел дерева, не имеющий родителя.
- *Предком* узла `x` является любой узел на пути от корня дерева до `x`. Другими словами, либо `parent(x)`, либо `parent(parent(x))` и т. д. На рис. B.9 узел с номером `2` является предком для узлов `7`, `3`, `0` и `9`.
- *Лист* — это любой узел, не имеющий потомков. Иначе это можно выразить так: все дочерние элементы узла являются пустыми поддеревьями.
- Фундаментальной характеристикой дерева является его *высота* — длина наибольшего пути от корня до листа. Высота дерева на рис. B.9 равна `3`, потому что самый длинный путь `1` → `2` → `3` → `0` (или, что то же самое, `1` → `2` → `3` → `9`) имеет три связи «родитель — потомок».

В следующем подразделе мы поближе познакомимся с двоичными деревьями поиска.

B.4.1. Двоичные деревья поиска

Двоичное (бинарное) дерево — это дерево, в котором количество дочерних элементов любого узла не превышает 2, то есть каждый узел может иметь 0, 1 или 2 потомка.

Двоичное дерево поиска (Binary Search Tree, BST) — это двоичное дерево, в котором каждый узел имеет связанный с ним ключ и удовлетворяет двум условиям: если `key(x)` — это ключ, связанный с узлом `x`, то:

- `key(x) > key(y)` для каждого узла `y` в левом поддереве `x`;
- `key(x) < key(z)` для каждого узла `z` в правом поддереве `x`.

Ключ узла также может быть его значением, но в общем случае узлы BST могут хранить и ключ, и значение, и любые дополнительные данные независимо друг от друга.

Вот еще несколько определений:

- *сбалансированное дерево* — это дерево, в котором высота левого и правого поддеревьев каждого узла отличается не более чем на 1 *и* оба поддерева, левое и правое, сбалансированы;
- *полное дерево* — это дерево, имеющее высоту H и у которого каждый листовой узел находится либо на уровне H, либо на уровне $H - 1$;
- *совершенное сбалансированное дерево* — это сбалансированное дерево, в котором все внутренние узлы имеют левое и правое поддеревья одинаковой высоты;
- *идеально сбалансированное дерево* также является *полным*.

ОПРЕДЕЛЕНИЯ

Существует два определения *сбалансированности* дерева: *сбалансированность по высоте*, которое используется в определениях выше, и *сбалансированность по весу*. Это две независимые друг от друга характеристики дерева, потому что ни одна из них не подразумевает другую. Обе могут привести к одинаковым результатам, но обычно используется первая, и мы просто будем подразумевать ее в тексте книги, говоря о сбалансированных деревьях.

Двоичные деревья поиска являются рекурсивными структурами. Каждое двоичное дерево поиска, по сути, может быть:

- пустым деревом;
- узлом с ключом и левым и правым поддеревьями.

Эта рекурсивная природа позволяет использовать понятные рекурсивные алгоритмы для всех основных операций с BST.

Двоичные деревья поиска предлагают компромисс между гибкостью и производительностью вставки в связанный список и эффективностью поиска в упорядоченном массиве. Все основные операции (`insert`, `delete`, `search`, `minimum` и `maximum`, `successor` и `predecessor`) требуют проверки количества узлов, пропорционального высоте дерева.

Следовательно, чем меньше высота дерева, тем выше производительность этих операций.

КАКОВА НАИМЕНЬШАЯ ВОЗМОЖНАЯ ВЫСОТА ДЕРЕВА?

Для двоичного дерева с n узлами наименьшая возможная высота равна $\log(n)$.

Рассмотрим двоичное дерево. Очевидно, что корень может быть только один. У него может быть не более 2 потомков, поэтому может быть не более 2 узлов на уровне 1. У каждого из них может быть 2 потомка, поэтому может быть не более $4 = 2^2$ узлов на уровне 2. Сколько узлов может быть на уровне 3? Как вы, наверное, догадались, $2^3 = 8$. Спускаемся вниз по дереву: с каждым уровнем высота увеличивается на единицу, а количество возможных узлов на текущем уровне удваивается.

Итак, на высоте h может быть 2^h узлов. Но также известно, что общее количество узлов полного дерева высотой h равно $2^0 + 2^1 + 2^2 + 2^3 + ... + 2^h = 2^{h+1} - 1$.

Чтобы высота была как можно меньше, нужно заполнить все уровни (кроме, возможно, последнего), потому что в противном случае можно просто перемещать узлы с последнего уровня вверх по дереву, пока не заполнятся вакансии на верхних этажах.

Следовательно, для n узлов, $2^{h+1} - 1 \leq n$, и, взяв логарифм обеих сторон, получаем $h \geq \log(n + 1) - 1$.

Двоичные деревья поиска по своей природе несбалансированны: напротив, деревья, созданные из одного и того же набора элементов, могут сильно различаться формой и высотой, в зависимости от последовательности вставки этих элементов (рис. В.10).

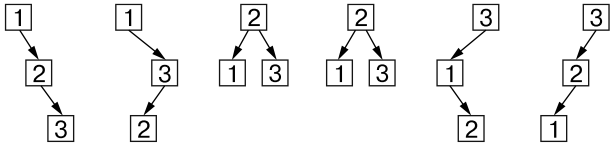


Рис. В.10. Все возможные формы двоичного дерева поиска из трех элементов. Форма зависит от порядка вставки элементов. Обратите внимание, что две последовательности вставки элементов, [2, 1, 3] и [2, 3, 1] создают деревья идентичной формы

В среднем после большого количества операций вставки вероятность получить перекошенное дерево довольно низка. Однако стоит отметить, что простейший алгоритм удаления узлов склонен приводить к получению перекошенного дерева. Для решения этой проблемы было предложено множество обходных путей, но на сегодняшний день нет доказательств, что они всегда дают лучший результат, чем простая версия.

Впрочем, есть несколько решений, позволяющих поддерживать сбалансированность двоичного дерева поиска без снижения производительности при вставке или удалении. Это, например:

- деревья поиска 2–3;
- красно-черные деревья;
- B-деревья;
- AVL-деревья.

B.5. ХЕШ-ТАБЛИЦА

Хеширование является, пожалуй, наиболее распространенным способом представления таблиц символов. Однако, решая задачу связывания ключей со значениями, мы неизбежно столкнемся с несколькими проблемами.

B.5.1. Хранение пар «ключ — значение»

Предполагается, что ключи и значения могут выбираться из разных областей. Нужно также принять решение о допустимости появления дубликатов ключей. Для простоты будем рассматривать только наборы уникальных ключей: статическую группу ключей всегда можно сделать уникальной.

Самый простой случай — когда роль ключей играют неотрицательные целые числа. Теоретически можно использовать массив и хранить в k-м элементе массива значение, связанное с ключом k. Это, однако, работает, только когда диапазон возможных ключей ограничен. Если, например, разрешить использовать в качестве ключей любые 32-битные целые числа без знака, то для хранения значений понадобится массив более чем с 4 миллиардами элементов, размер которого может оказаться больше объема ОЗУ даже в самом современном компьютере. Хуже того, если в роли ключей рассматривать «длинные целые числа», занимающие 8 байт в памяти, то соответствующий массив будет иметь 18 миллиардов миллиардов элементов. Нет, это не опечатка: речь идет о миллиардах миллиардов.

Хуже всего то, что при использовании массивов, даже если известно, что в них будет храниться только несколько тысяч целочисленных ключей, все равно понадобится массив с 2^{32} элементами, если в роли ключа будет позволено использовать любое 32-битное целочисленное значение.

Мы мало что можем сделать для улучшения ситуации, когда нужно хранить много элементов, но, когда известно, что храниться будет только ограниченное количество элементов (скажем, несколько сотен или тысяч), это совсем другая история. Даже если ключи могут произвольно выбираться из обширного множества (например, из множества всех целых чисел, которые могут быть представлены 32 битами, их количество чуть больше 4 миллиардов), но храниться будут лишь некоторые из них, можно использовать более экономное решение. Это решение — хеширование.

В.5.2. Хеширование

Хеширование обеспечивает компромисс между массивами с индексированием по ключу и несортированными массивами в сочетании с последовательным поиском. Первое решение обеспечивает поиск за постоянное время, но требует пространства, пропорционального набору возможных ключей. Последнее требует линейного времени для поиска, но пространство пропорционально количеству фактических ключей.

Используя хеш-таблицы, можно ограничить размер массива, скажем, M элементами. Однако, как будет показано ниже, в массиве можно хранить больше M элементов, в зависимости от способа разрешения коллизий. В хеш-таблицах каждый ключ преобразуется в индекс от 0 до $M - 1$ с помощью хеш-функции.

Стоит отметить, что введение такого преобразования ослабляет ограничение на использование только неотрицательных целых чисел в качестве ключей. При желании можно «сериализовать» любой объект в строку и преобразовать любую строку в целое число по модулю M. В оставшейся части обсуждения для краткости будем предполагать, что ключи являются целыми числами.

Точная хеш-функция, которую нужно использовать, зависит от типа ключей и коррелирует с размером массива. Вот наиболее известные примеры:

- *метод деления* — для целочисленного ключа k его хеш $h(k)$ определяется как:

  $$h(k) = k \,\%\, M,$$

  где $\%$ представляет оператор деления по модулю (остаток от деления).

  Для этого метода размер M таблицы должен быть простым числом, не слишком близким к степени двойки.

  Например, если $M = 13$, то $h(0) = 0$, $h(1) = 1$, $h(2) = 2$... $h(13) = 0$, $h(14) = 1$ и т. д.;

- *метод умножения*:

  $$h(k) = \lfloor M(kA \,\%\, 1) \rfloor,$$

  где $0 < A < 1$ — действительная константа, а $(k \times A \,\%\, 1)$ — дробная часть произведения $k \times A$. В этом случае величина M обычно выбирается равной степени двойки, но величина A должна выбираться с учетом величины M.

  Например, пусть $M = 16$ и $A = 0{,}25$, тогда:

```
k = 0 => h(k) = 0
k = 1 => k × A = 0.25, k×A%1 = 0.25, h(k) = 4
k = 2 => k × A = 0.5, k×A%1 = 0.5, h(k) = 8
k = 3 => k × A = 0.75, k×A%1 = 0.75, h(k) = 12
k = 4 => k × A = 1, k×A%1 = 0, h(k) = 0
k = 5 => k × A = 1.25, k×A%1 = 0.25, h(k) = 4
...
```

  и т. д.

  Как видите, величина 0,25 — не лучший выбор для A, потому что $h(k)$ будет принимать только пять различных значений. Однако этот пример отлично иллюстрирует и сам метод, и то, почему важно с особым тщанием подходить к выбору его параметров.

Существуют и более продвинутые методы улучшения качества хеш-функции, все больше и больше приближающие ее к равномерному распределению.

B.5.3. Разрешение коллизий в хешировании

Независимо от качества или однородности хеш-функции, количество ключей `m` может продолжать расти и наконец превысить размер *n* таблицы. В этот момент срабатывает *принцип Дирихле* и стройный порядок вещей начинает ломаться.

ОПРЕДЕЛЕНИЕ

Принцип Дирихле гласит, что если количество возможных значений ключа превышает количество доступных слотов, то в какой-то момент у вас обязательно появятся два разных ключа, отображаемых в один слот. Что случится, если попытаться добавить в таблицу оба ключа? Очевидно, возникнет конфликт, коллизия, и понадобится способ разрешения этих коллизий, позволяющий различать два разных ключа и отыскивать их при поиске.

Существует два основных способа разрешения конфликтов сопоставления разных ключей в один слот.

- *Объединение в цепочку* — каждый элемент в массиве хранит ссылку на другую структуру данных, содержащую все ключи, отображенные в этот элемент (рис. B.11). Вторичной структурой данных может быть связанный список (обычно), дерево или даже другая хеш-таблица (как при идеальном хешировании, методе, который обеспечивает наилучшую возможную производительность хеширования на статическом наборе ключей, известном заранее). В этом случае нет ограничений на количество элементов, которые могут храниться в хеш-таблице, но производительность ухудшается по мере добавления все большего количества элементов, потому что некоторые списки становятся все длиннее и длиннее и требуют все большего числа шагов, чтобы найти элемент.

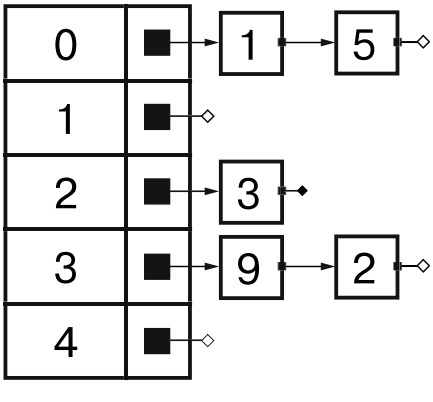


Рис. B.11. Пример хеш-таблицы, использующей прием объединения в цепочку и связанный список для разрешения конфликтов

- *Открытая адресация* — элементы хранятся непосредственно в массиве, а в случае конфликтов генерируется (детерминированно) другое значение хеш-функции для следующей позиции, которую нужно попробовать. На рис. В.12 показан пример хеш-таблицы, использующей открытую адресацию для разрешения конфликтов.

  Хеш-функции для открытой адресации выглядят так:

$$h(k, i) = (h'(k) + f(i, k)) \% M,$$

  где i — количество уже проверенных позиций, а f — функция количества попыток и, возможно, ключа. Открытая адресация позволяет экономить память благодаря отсутствию вторичных структур данных, но имеет проблемы, которые редко делают ее лучшим выбором. Во-первых, усложняется удаление элементов, а поскольку размер хеш-таблицы ограничен и выбирается при создании, то это означает, что такие таблицы будут быстро заполняться и их придется перераспределять. Хуже того, элементы имеют тенденцию группироваться в кластеры, и, когда таблица заполнится, потребуется предпринять много попыток, чтобы найти свободный слот, — высока вероятность, что потребуется линейное число попыток, когда таблица заполнена почти наполовину.

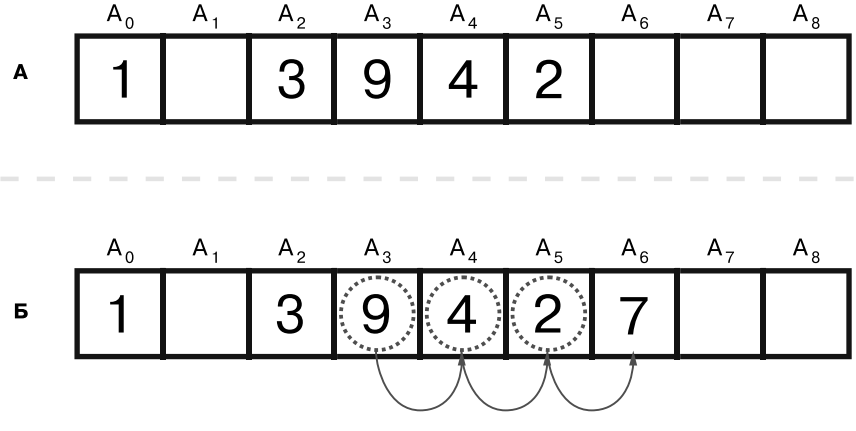


Рис. В.12. Хеш-таблица, использующая открытую адресацию для разрешения конфликтов. Назначение элементов зависит от порядка вставки. Предположим, что f(i) = i и имеют место те же коллизии, что показаны на рис. В.11. *А*. Мы можем заключить, что ключ 9 был вставлен перед 2 (иначе, поскольку предполагается, что 9 и 2 будут отображены в один и тот же слот, как на рис. В.11, ключ 2 был бы сохранен в A3). Кроме того, ключ 4 тоже был вставлен перед 2, потому что, когда алгоритм обнаружит, что слот A3 не пустой, следующим он попробует слот A4 (поскольку f(i) = i, выполняется линейный просмотр позиций после коллизии). *Б*. Попытка добавить 7, при этом предполагается, что h(7) = 3. Из-за особенностей открытой адресации и определения функции f последовательно опробуются слоты 3, 4 и 5, которые уже заняты. Затем наконец обнаруживается свободный слот A6 и в него добавляется новый ключ

ПРОБЛЕМЫ С МАСШТАБИРОВАНИЕМ ХЕШ-ТАБЛИЦ

Даже при использовании приема объединения в цепочку количество слотов в таблице обычно остается неизменным, потому что изменение размера таблицы потребует изменения хеш-функции и, следовательно, целевых слотов потенциально для всех уже сохраненных элементов. Это, в свою очередь, заставит удалить каждый элемент из старой таблицы, а затем вставить их в новую. Стоит отметить, что такого рода ситуации часто возникают в распределенных кэшах (таких как Cassandra или Memcached), когда необходимо добавить новый узел, и — если в архитектуре веб-сайта не используются надлежащие обходные пути — это может замедлить или даже обрушить весь сайт.

В главе 7 описывается один из способов смягчения этой проблемы — последовательное хеширование.

B.5.4. Производительность

Как уже упоминалось, прием объединения в цепочку — часто самый предпочтительный способ разрешения коллизий, потому что дает несколько преимуществ с точки зрения времени выполнения и распределения памяти. Для хеш-таблицы, использующей прием объединения в цепочку и связанные списки, когда таблица имеет размер m и содержит n элементов, все операции требуют в среднем $O(n / m)$ времени.

ПРИМЕЧАНИЕ

В большинстве случаев можно считать, что операции с хеш-таблицами выполняются за время O(1), однако следует помнить, что в худшем случае, когда все элементы отображаются в один и тот же слот (то есть в одну и ту же цепочку) внутри таблицы, время составляет O(n). В таких случаях время, необходимое для удаления или поиска элемента, равно O(n). Это, однако, очень маловероятно, по крайней мере, когда хеш-функция спроектирована правильно.

Однако, если набор возможных ключей статичен и известен заранее, то можно использовать *идеальное хеширование* и для всех операций получить время O(1) в худшем случае.

B.6. СРАВНИТЕЛЬНЫЙ АНАЛИЗ ОСНОВНЫХ СТРУКТУР ДАННЫХ

Теперь, описав все основные структуры данных, попробую обобщить их характеристики, свойства и производительность в табл. B.1.

Свойства, которые мы рассмотрим:

- *упорядоченность* — поддержка детерминированного порядка элементов. Это может быть естественный порядок элементов или порядок вставки;

- *уникальность* — запрет дублирования элементов/ключей;
- *ассоциативность* — возможность индексирования элементов по ключу;
- *динамичность* — необходимость заранее определить, сможет ли контейнер изменять размер при вставке/удалении или всегда будет иметь максимальный размер;
- *локальность* — ссылочная локальность, то есть хранение всех элементов в одном непрерывном блоке памяти.

Таблица B.1. Сравнение основных структур данных

Структура	Упорядоченность	Уникальность	Ассоциативность	Динамичность	Локальность
Массив	Да	Нет	Нет	Нет*	Да
Односвязный список	Да	Нет	Нет	Да	Нет
Двусвязный список	Да	Нет	Нет	Да	Нет
Сбалансированное дерево (например, BST)	Да	Нет	Нет	Да	Нет
Куча (см. главу 2)	Нет**	Да	«Ключ — приоритет»	Нет	Да
Хеш-таблица	Нет	Да	«Ключ — значение»	Да***	Нет

* В большинстве языков программирования массивы изначально являются статическими, но ценой небольшой потери производительности из статических массивов можно получить динамические.

** Кучи лишь частично определяют порядок ключей. Они позволяют сортировать ключи по приоритету, но не сохраняют информацию о порядке вставки.

*** Хеш-таблицы имеют динамический размер, когда конфликты разрешаются с использованием метода объединения в цепочки.

В рамках сравнения мы также должны учитывать относительную производительность. Но что на самом деле означает производительность для всей структуры данных, если мы обсудили только время работы их отдельных методов?

Обычно производительность структуры данных и тем более ее реализации — это компромисс между производительностью отдельных методов. Это справедливо для всех структур данных: не существует идеальной структуры данных с оптимальной производительностью всех поддерживаемых операций. Например, массивы обеспечивают быстрый произвольный доступ на основе позиции, но требуют дополнительного времени для изменения своей формы и еще больше времени[1], когда нужно

[1] Поиск и прямой доступ в несортированных массивах выполняется ужасно медленно. Отсортированные массивы способны выполнить те же операции за время $O(\log(n))$.

найти элемент по значению. Списки позволяют быстро вставлять новые элементы, но поиск и доступ по позиции в списках выполняются медленно. Хеш-таблицы предлагают быстрый поиск по ключу, но поиск последующего элемента или максимума и минимума в них выполняется очень медленно.

На практике выбор наилучшей структуры данных должен основываться на тщательном анализе решаемой задачи и производительности структуры данных, но самое важное не подобрать самую оптимальную, а избежать использования плохой структуры данных (которая может стать узким местом) просто ради того, чтобы получить небольшую экономию в среднем (и более простом в реализации) выборе.

Г

Контейнеры в роли очередей с приоритетами

Одним из самых больших подмножеств структур данных, если не самым большим (на сегодняшний день), является набор контейнеров. Контейнер — это набор объектов, поддерживающий операции по их добавлению, удалению и извлечению. Основные различия между разными типами контейнеров обусловлены:

1) порядком извлечения элементов из контейнера;
2) требованиями к уникальности элементов;
3) требованиями к ассоциативности, то есть договоренностью, должен контейнер хранить простые элементы или пары «ключ — значение»;
4) возможностями поиска элементов и набором операций, которые можно выполнять с хранимыми данными;
5) производительностью.

Пункты 1–4 определяют абстракцию контейнера — его поведение, или, говоря техническим языком, его API. Однако даже при одинаковых API реализации могут различаться.

Но пока сосредоточимся на абстрактных структурах данных, а конкретнее — на структурах данных с поддержкой понятия приоритета. На высоком уровне мы имеем черный ящик, *контейнер*, хранящий значения и удаляющий и возвращающий определенный элемент каждый раз, когда мы его просим об этом.

Это описание является достаточно общим, охватывает практически все типы контейнеров: суть сводится к последовательному определению приоритета элементов. Выражаясь языком математики, приоритет — это однозначное соответствие между элементами и числами, причем, как правило, более низкие значения означают более высокий приоритет.

Некоторые из этих способов задания приоритета настолько распространены, что классифицируются как самостоятельные структуры данных. Рассмотрим самые распространенные из них.

Г.1. МУЛЬТИМНОЖЕСТВО

Мультимножество (bag), как показано на рис. Г.1, — это множество, поддерживающее только операции `add` и `iterate`. Мультимножество не поддерживает операцию удаления элементов. После проверки наличия элементов в контейнере клиенты могут выполнять итерации по этим элементам; однако фактический порядок не определен, и клиенты не должны строить никаких предположений в отношении его.

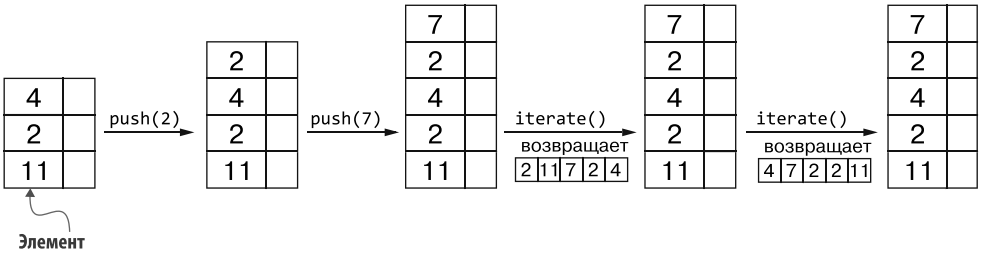


Рис. Г.1. Операции с мультимножеством. Вставка добавляет элементы в контейнер, но не задает индекс для них. Единственная другая доступная операция — перебор элементов; она не изменяет содержимое контейнера и не гарантирует порядок обхода элементов, это может быть порядок вставки или совершенно случайный порядок, отличающийся для каждого нового цикла итераций

Мультимножество может пригодиться, когда требуется собрать предметы и обработать их как единое целое, например собрать выборку, а затем вычислить по ней некоторую статистику, такую как среднее значение или стандартное отклонение — порядок в этом случае не имеет значения.

Вот как можно описать мультимножество в терминах *очередей с приоритетами* (Priority Queues, PQ): это очередь, не имеющая операции удаления элемента (метода `top`), но поддерживающая *просмотр* элементов по одному, причем приоритет каждого элемента задается случайным числом из равномерного распределения. Кроме того, приоритеты меняются в каждой итерации.

Г.2. СТЕК

Стек — это коллекция, возвращающая свои элементы в соответствии с политикой *LIFO* (last in first out — «последним пришел, первым ушел»), то есть в порядке, обратном порядку их добавления. Это делает стеки очень полезными, когда нужно обратить последовательность или сначала получить доступ к недавно добавленным элементам.

Стек можно представить (рис. Г.2) как специализированную очередь с приоритетами, в которой приоритеты элементов определяются временем их вставки (самый последний вставленный элемент имеет наивысший приоритет).

Первое важное отличие стека от мультимножества состоит в том, что для стека порядок имеет значение.

Второе отличие: стек позволяет не только выполнять обход элементов, но и удалять их из контейнера. В частности, стек — это структура данных с ограниченным доступом, потому что позволяет добавлять или удалять элементы только на вершине стека.

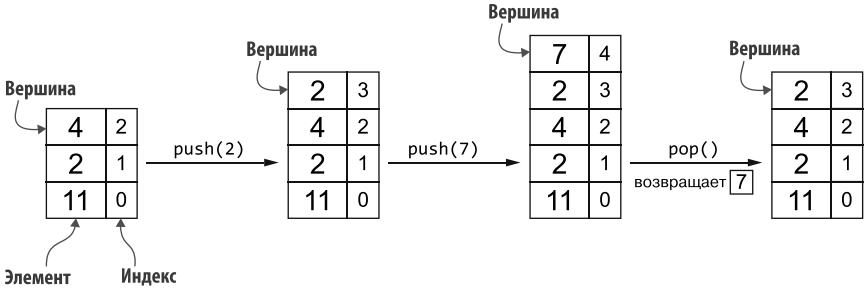


Рис. Г.2. Операции со стеком. Вставка (push) добавляет элементы на вершину стека. Удаление (pop) удаляет и возвращает верхний элемент. Стеки (как и очереди) также обычно поддерживают операцию просмотра (peek), возвращающую верхний элемент, не удаляя его из контейнера

Для объяснения работы стека обычно приводят такие примеры из реальной жизни, как стопка грязных тарелок (тарелки можно забирать только с вершины стопки, поэтому сначала вы моете те, что были добавлены последними). Однако именно в информатике стек нашел особенно важное применение:

- *для управления памятью*: позволяет запускать подпрограммы (стек вызовов);
- *в реализации шаблона «Хранитель» (Memento)*: позволяет отменять действия в текстовом редакторе или просматривать историю посещения страниц в браузере вперед и назад;
- *для поддержки рекурсии* и некоторых важных алгоритмов;
- в мире JavaScript можно услышать о *путешествиях во времени* в React.

И этот список отнюдь не исчерпывающий.

Г.3. ОЧЕРЕДЬ

Очередь — это коллекция, основанная на политике *FIFO* (first in first out — «первым пришел, первым ушел»), то есть очередь возвращает элементы в том же порядке, в каком они были добавлены в нее. Очередь тоже является структурой данных с ограниченным доступом, но, в отличие от стеков, позволяет добавлять элементы в начало очереди, а удалять с конца (это также означает, что очереди имеют чуть более сложную реализацию, чем стеки).

На рис. Г.3 показан пример очереди.

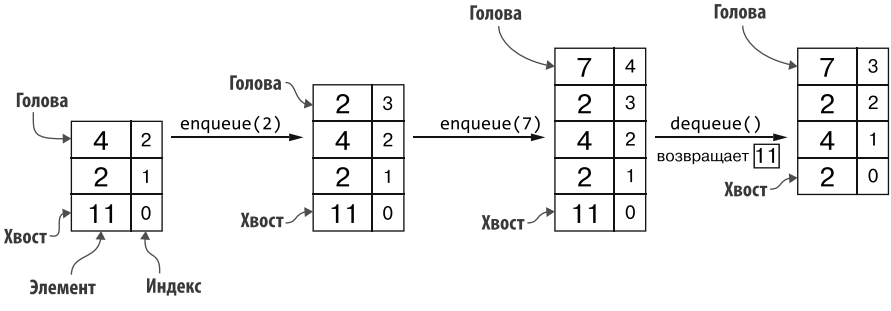


Рис. Г.3. Операции с очередями. Вставка (enqueue) добавляет элементы в начало очереди. Удаление (dequeue) удаляет элемент с конца очереди и возвращает его

Очередь, как нетрудно представить, является частным случаем очереди с приоритетами, где приоритетом элемента является его возраст: чем старше элемент, тем выше его приоритет.

Название этой структуры данных весьма показательно. И действительно, в реальном мире мы сталкиваемся с этой структурой каждый раз, когда становимся в справедливую очередь: тот, кто ждал дольше всех, обслуживается первым. Как и стеки, очереди имеют фундаментальное значение для информатики и используются во многих алгоритмах.

Г.4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ КОНТЕЙНЕРОВ

Теперь пришло время свести в единую систему обсуждаемые абстрактные структуры с учетом их нескольких основных характеристик: поддержки упорядочения ключей, требования к уникальности элементов и ассоциативности. В табл. Г.1 перечислены все контейнеры, представленные в предыдущих подразделах, а также словари (хеш-таблицы, обсуждавшиеся в приложении В).

Таблица Г.1. Сравнительные характеристики контейнеров

Структура	Упорядоченность	Уникальность	Ассоциативность
Мультимножество	Нет	Нет	Нет
Стек	Да	Нет	Нет
Очередь	Да	Нет	Нет
Очередь с приоритетами	Да	Нет*	Нет
Словарь	Нет	Да	«Ключ — значение»

* Как упоминается в подразделе 2.6.6, ослабление ограничения уникальности ключей требует дополнительных усилий для обновления приоритетов.

Д Рекурсия

Весьма вероятно, что вы как читатель этой книги уже знакомы с циклами. Как обсуждалось в приложении А, циклы `for`, `while` и `do-while` (и некоторые другие, в зависимости от языка) могут служить примерами итераций.

Циклы дают простую возможность организовать многократное выполнение одних и тех же действий с рядом элементов. Обычно циклы идут рука об руку с определенными структурами данных, например с контейнерами, такими как списки и массивы.

Циклы, как правило, хорошо работают, когда структура данных имеет линейную форму, однако иногда простого цикла недостаточно для решения задачи. Ведь, например, деревья и графы несколько сложнее списков. Конечно, существуют обходные решения, позволяющие использовать циклы для итераций и по этим структурам, но часто проще и удобнее использовать другой подход.

Повсюду в этой книге для описания алгоритмов независимым от языка способом используется псевдокод. Однако рекурсии иллюстрировать лучше все-таки с использованием кода на реальном языке программирования. Поэтому для описания идей в этом приложении применен код на JavaScript. Выбор JavaScript обусловлен такими двумя характеристиками этого языка, как:

- полноценная поддержка замыканий для чтения/записи;
- полноценная поддержка функций.

Эти аспекты позволяют проиллюстрировать интересный прием, называемый *мемоизацией* (memoization), который можно использовать в некоторых контекстах для повышения производительности рекурсивных алгоритмов.

Те же результаты можно получить аналогичным образом в большинстве функциональных языков. В объектно-ориентированных языках есть свои обходные пути: например, в Java мемоизацию можно реализовать с использованием статических полей класса.

Д.1. ПРОСТАЯ РЕКУРСИЯ

Простейший случай рекурсии — вызов функцией самой себя в заданной точке в потоке выполнения. Наиболее часто для иллюстрации рекурсии приводят в качестве примера вычисление факториала числа и чисел Фибоначчи.

В листинге Д.1 показана возможная реализация на JavaScript функции, вычисляющей числа Фибоначчи.

Листинг Д.1. Числа Фибоначчи, реализация на JavaScript

```
function fibonacci(n) {
  if (n < 0) {
    throw new Error('n can not be negative');
  } else if (n === 0 || n === 1) {
    return 1; // Базовый случай
  } else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Д.1.1. Ловушки

Рекурсия не лишена рисков. Применяя рекурсию, прежде всего нужно гарантировать наличие базового случая, достижимого всегда. Если в примере в листинге Д.1 забыть проверить неотрицательность аргумента *n*, то вызов `fibonacci(-1)` переберет все отрицательные целые числа, поддерживаемые в JavaScript, прежде чем будет достигнут базовый случай *n* === 0. Но, вероятнее всего, в этом случае код сгенерирует ошибку переполнения стека вызовов задолго до того, как приблизится к базовому случаю. (Конечно, даже при использовании цикла нужно правильно определить условия остановки, иначе можно получить бесконечный цикл.)

Даже эта дополнительная проверка не может гарантировать полной безопасности. В слабо типизированных языках, таких как JavaScript, механизм приведения типов попытается преобразовать в числа аргументы, переданные этой функции.

Попробуйте, например, вызвать `fibonacci('a')` или `fibonacci(true)`. Представленная выше реализация не проверяет тип `n`, соответственно, операция `'a' - 1` вернет `NaN`[1], так же как и операция `NaN - 1`. Поскольку `NaN` не меньше 0, не равно 0 или 1, то в результате образуется бесконечная (теоретически) последовательность рекурсивных вызовов, которая снова приведет к ошибке.

[1] NaN — специальное значение в JavaScript, означающее Not a Number («не число»), которое возвращается, когда возникает проблема при разборе числа интерпретатором или при выполнении числовой операции.

Другим недостатком является пустая трата ресурсов. Если внимательно посмотреть на код в листинге Д.1, то можно понять, почему это не лучший вариант использования рекурсии. Например, попробуем отследить рекурсивные вызовы для первоначального вызова `fibonacci(4)`:

Листинг Д.1. Числа Фибоначчи, JavaScript

```
fibonacci(4)
  fibonacci(3)
    fibonacci(2)
      fibonacci(1)
      fibonacci(0)
    fibonacci(1)
  fibonacci(2)
    fibonacci(1)
    fibonacci(0)
```

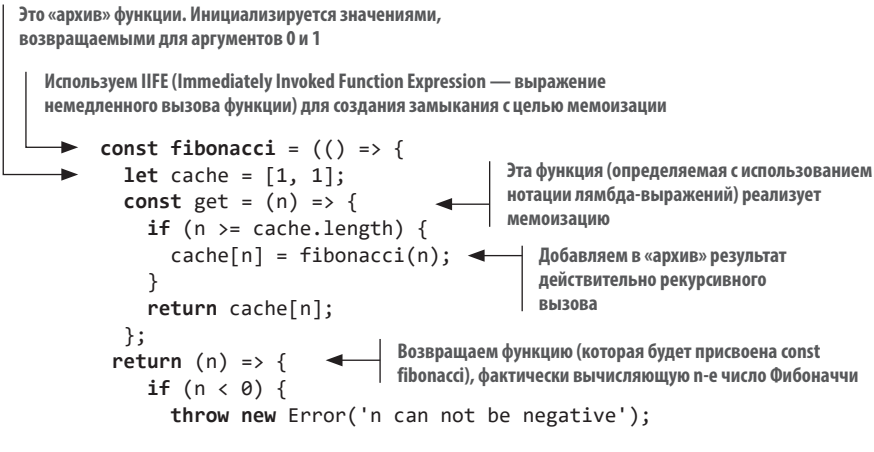
Отступы здесь добавлены чисто стилистически, чтобы помочь исследовать иерархию вызовов.

Если то же самое проделать для вызова `fibonacci(5)`, ситуация будет выглядеть еще более очевидной: одно и то же значение вычисляется несколько раз (с накладными расходами на вызовы функций).

Многократное вычисление подзадач обычно предполагает наличие другого, более эффективного решения. И действительно, в подобных случаях *динамическое программирование* часто может дать лучшую альтернативу.

В JavaScript аналогичный подход к отслеживанию результатов рекурсивных вызовов получил название «мемоизация». В этом случае перед выполнением рекурсивного вызова проверяется наличие искомого результата в кэше, как это делает реализация в листинге Д.2.

Листинг Д.2. Мемоизация функции вычисления чисел Фибоначчи



```
    } else if (n > 1) {
      return get(n - 1) + get(n - 2);
    }
    return 1;
  };
})();
```

Вычисляем f(n), получив значения для n – 1 и n – 2

Соответствует случаю n === 1 || n === 0

Выполняем IIFE, чтобы результат вызова функции был присвоен константе в первой строке

Оба способа использования рекурсии, простой и с мемоизацией, могут привести к проблемам с нехваткой памяти (подробнее об этом рассказывается в следующих нескольких разделах).

Д.1.2. Оправданная рекурсия

Предыдущий пример показывает, что использование рекурсии не улучшило ситуацию по сравнению с итеративным алгоритмом. На самом деле в некоторых случаях рекурсия является правильным выбором потому, что природа задачи или ее определение являются рекурсивными. Обычно это больше вопрос ясности и чистоты кода, чем производительности. Однако далее будет показано, как *хвостовая рекурсия* сочетает в себе и то и другое (по крайней мере в современных языках программирования).

А пока взглянем на элегантную реализацию обхода бинарного дерева в прямом порядке (листинг Д.3).

Листинг Д.3. Обход узлов двоичного дерева в прямом порядке

```
function preorder(node) {
  if (node === undefined || node === null) {
    console.log("leaf");
  } else {
    console.log(node.value);
    preorder(node.left);
    preorder(node.right);
  }
}
```

Рекурсивный обход левого поддерева

Рекурсивный обход правого поддерева

ОБХОД ДЕРЕВА

Обход в прямом порядке позволяет перечислить ключи, хранящиеся в дереве. Алгоритм создает список, начиная с ключа, хранящегося в корне, а затем рекурсивно обходит дочерние поддеревья. Дойдя до любого узла, он сначала добавляет в список ключ текущего узла, а затем выполняет обход дочерних элементов (в случае двоичного дерева — начиная с левого элемента, а затем, после полного обхода этого поддерева, переходит к правому элементу).

Другими распространенными способами обхода являются симметричный порядок (левое поддерево → текущий узел → правое поддерево) и обратный порядок (левое поддерево → правое поддерево → текущий узел).

Конечно, помимо вывода с узлами, можно выполнять любые другие действия. И да, обход можно также реализовать с использованием циклов, как показано в листинге Д.4.

Листинг Д.4. Обход узлов двоичного дерева в прямом порядке, итеративная версия

```
function preorderIterative(node) {
  if (node === undefined || node === null) {
    return;
  }
  let nodeStack = [node];
  while (nodeStack.length > 0) {
    node = nodeStack.pop();
    if (node === undefined) {
      console.log("leaf");
    } else {
      console.log(node.value);
      nodeStack.push(node.right);
      nodeStack.push(node.left);
    }
  }
}
```

Здесь явно используется стек для моделирования поведения рекурсивных вызовов (см. следующий раздел)

Вталкивание узла в стек эквивалентно рекурсивному вызову

Эти два метода эквивалентны, но посмотрите, насколько элегантнее выглядит первый. Попробуйте написать обе версии для обхода двоичного дерева в обратном порядке, и вы поймете, что реализовать правильно итеративную версию гораздо сложнее.

Д.2. ХВОСТОВАЯ РЕКУРСИЯ

Каждый раз, когда вызывается функция, в стеке программы создается новая запись, так называемый *кадр стека*. Кадр стека имеет несколько полей, необходимых программе или виртуальной машине для выполнения вызываемой функции, а затем, по ее завершении, для возврата в точку вызова.

Вот список (неполный) этих полей:

- *счетчик инструкций* (указатель на следующую инструкцию в вызывающей функции, которой должно быть передано управление после завершения вызываемой функции);
- все аргументы, переданные вызываемой функции;
- все локальные переменные;
- зарезервированное место для значения, возвращаемого функцией.

Рекурсивные вызовы не являются исключением, разве что первый вызов рекурсивной функции может не вернуться в вызывающую, пока не будет достигнут базовый случай. Это означает, что цепочка вызовов может быть очень длинной.

Возьмем, к примеру, функцию вычисления факториала, показанную в листинге Д.5.

Листинг Д.5. Функция вычисления факториала на JavaScript

```
function factorial(n) {
  if (n < 0) {
    throw new Error('n can not be negative');
  } else if (n === 0) {
    return 1;                          // Базовый случай
  }
  let f = factorial(n - 1);            // Рекурсивный вызов factorial. Здесь явно используется
                                       // переменная для вычисления результата ниже
  return n * f;
}
```

Вызов `factorial` с положительным целым числом *n* потребует создания *n* кадров стека. На рис. Д.1 показан примерный вид кадра стека для вызова `factorial(3)`, где можно видеть, что этот вызов создает три новых кадра стека. Поскольку область памяти для стека имеет ограниченный размер (причем не очень большой по сравнению с кучей), рекурсивное выполнение может привести к катастрофе. Попробуйте вызвать этот метод с достаточно большим значением *n*, и вы вызовете ошибку сегментации.

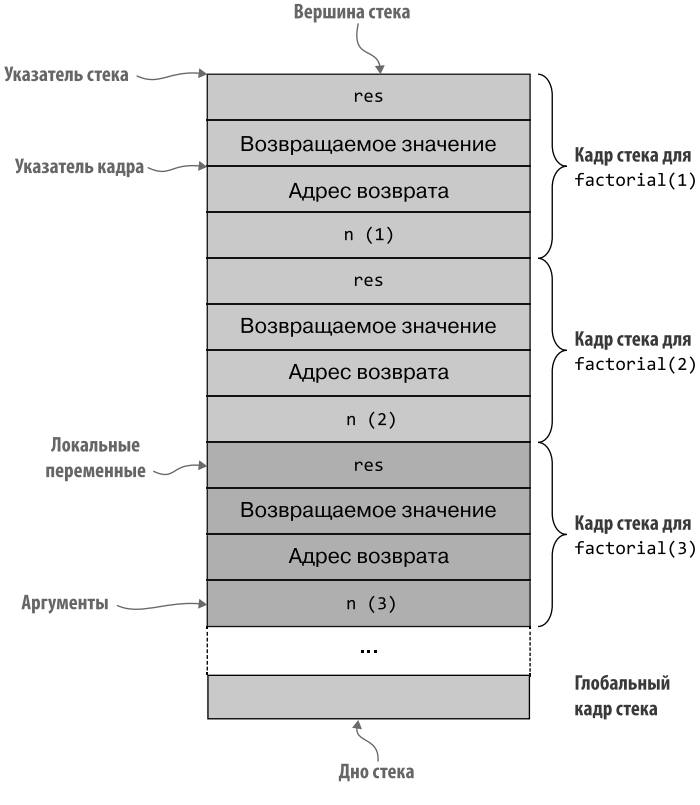


Рис. Д.1. Пример кадра стека для вызова factorial(3) — первой версии функции factorial

Ошибка сегментации (segmentation fault), также известная как *нарушение прав доступа* (access violation), представляет собой состояние отказа, вызванное оборудованием с защитой памяти, уведомляющее операционную систему о том, что какое-то программное обеспечение попыталось получить доступ к области памяти, не предназначенной для него.

К счастью для нас, современные компиляторы способны оптимизировать некоторые виды рекурсивных вызовов. *Хвостовой вызов* — это вызов функции, который выполняется как последняя операция в функции. Рекурсия считается *хвостовой*, если рекурсивный вызов (вызовы) находится в *хвостовой позиции*. Более полную информацию о хвостовых вызовах и рекурсии можно найти в отличной статье в блоге 2ality[1].

Большинство компиляторов оптимизируют хвостовые вызовы и предотвращают создание новых кадров стека. Более того, компиляторы могут заменить хвостовую рекурсию итеративной версией. Например, в JavaScript поддержка оптимизации хвостовых вызовов была введена в ES2015.

Однако, чтобы гарантировать применение этой оптимизации к нашей функции `factorial`, ее нужно немного преобразовать и тем самым сделать рекурсивный вызов последней операцией перед возвратом.

В некоторых функциях сделать это довольно сложно. Например, взгляните на функцию `factorial` в листинге Д.6. Она была реорганизована, и теперь кажется, что рекурсивный вызов выполняется последним, но на самом деле это не так! Фактически последняя выполняемая операция — это умножение *n* на результат рекурсивного вызова. В этом случае оптимизация хвостового вызова применяться не будет.

Листинг Д.6. Функция вычисления факториала (реорганизованная)

```
function factorial(n) {
  if (n < 0) {
    throw new Error('n can not be negative');
  } else if (n === 0) {
    return 1; <-- Базовый случай
  }

  return n * factorial(n-1); <-- Рекурсивный вызов не является последней инструкцией в этой функции; последняя выполняемая операция — умножение
}
```

Похоже, ситуация тупиковая, но не стоит отчаиваться, еще не все потеряно. В листинге Д.7 показано, как можно переписать даже нашу функцию `factorial()`, чтобы она выполняла хвостовую рекурсию. Стоит лишь добавить дополнительный аргумент, накопитель, запоминающий результат уже выполненных умножений, что позволит выполнить эту операцию перед рекурсивным вызовом.

Для оптимизации хвостового вызова компилятор просто превратит его в цикл. Фактически хвостовую рекурсию всегда можно записать в виде простого цикла. Например, в листинге Д.8 показано использование цикла `while` для вычисления факториала.

[1] http://mng.bz/2ejm.

Листинг Д.7. Функция вычисления факториала (с хвостовой рекурсией)

```
function factorial(n, acc=1) {
  if (n < 0) {
    throw new Error('n can not be negative');
  } else if (n === 0) {
    return acc;
  }
  return factorial(n - 1, n * acc);
}
```

Теперь метод принимает два аргумента: число n и аккумулятор со значением, по умолчанию равным 1 (так что вызов factorial(n) эквивалентен вызову factorial(n,1))

Базовый случай: возвращаем аккумулятор с произведением, накопленным предыдущими вызовами

Теперь рекурсивный вызов является последней инструкцией в этой функции, так как умножение производится при оценке аргументов перед рекурсивным вызовом

Листинг Д.8. Функция вычисления факториала (итеративная версия)

```
function factorial(n) {
  let acc = 1;
  if (n < 0) {
    throw new Error('n can not be negative');
  }

  while (n > 1) {
    acc *= n--;
  }
  return acc;
}
```

Инициализируем аккумулятор (базовый случай)

Моделируем хвостовую рекурсию с помощью цикла

Д.3. ВЗАИМНАЯ РЕКУРСИЯ

Функция может вызывать себя не только напрямую, но и через другую функцию. Если две или более функции вызывают себя в цикле, то такие функции называются *взаимно рекурсивными*.

Взаимную хвостовую рекурсию тоже можно оптимизировать, подобно обычному хвостовому вызову, но большинство компиляторов ограничиваются оптимизацией простой хвостовой рекурсии.

Код в листинге Д.9 поможет получить представление о том, как работает взаимная рекурсия.

Листинг Д.9. Пример взаимно рекурсивных функций

```
function f(n) {
  return n + g(n - 1);
}
function g(n) {
  if (n < 0) {
    return 1;
  } else if (n === 0) {
    return 2;
  } else {
    return n * f(n/3);
  }
}
```

Функция f() вызывает g()

А функция g() вызывает f()

Здесь определена пара методов, `f()` и `g()`, причем `f()` не вызывает сама себя напрямую, и точно так же `g()` не вызывает саму себя.

Но взглянем на стек вызовов для `f(7)`:

- `f(7)`;
- `g(6)`;
- `f(2)`;
- `g(1)`;
- `f(0.3333)`;
- `g(-0.6666)`.

Единственный вызов `f()` сгенерировал цепочку вызовов, в которой две функции поочередно вызывают друг друга. Излишне говорить, что взаимную рекурсию еще сложнее отслеживать и оптимизировать.

Задачи классификации и рандомизированные алгоритмы

Чтобы разобраться в анализе производительности таких структур данных, как декартовы деревья (см. главу 3) и фильтры Блума (см. главу 4), вы должны сделать шаг назад и сначала познакомиться с классом алгоритмов, известных далеко не всем разработчикам.

Многим, когда их спрашивают об алгоритмах, сразу приходят в голову мысли о детерминированных алгоритмах. Этот подкласс алгоритмов легко спутать с группой в целом: наш здравый смысл подсказывает, что алгоритм описывает последовательность действий, которые при получении определенного ввода выполняют одни и те же шаги для получения четко определенного результата.

Это действительно распространенный сценарий. Однако алгоритм можно описать как последовательность четко определенных шагов, которые приведут к детерминированному результату, но займут непредсказуемое (хотя и конечное) время. Алгоритмы, имеющие такое поведение, называются *алгоритмами Лас-Вегаса*.

Еще менее понятными кажутся алгоритмы, которые могут давать разные, в том числе непредсказуемые, результаты для одних и тех же входных данных или могут не завершиться за конечное время. Для этого класса алгоритмов тоже есть название: они называются *алгоритмами Монте-Карло*.

Исследователи давно спорят о том, следует считать последние алгоритмами или, скорее, эвристиками, но я склоняюсь к мнению, что это все-таки алгоритмы, потому что последовательность шагов в алгоритмах Монте-Карло детерминирована и четко определена. В этой книге представлено несколько алгоритмов Монте-Карло, соответственно, у вас есть возможность составить о них свое обоснованное мнение.

Но прежде, чем углубляться в эти классы алгоритмов, необходимо определить особенно полезный вид задач — *задачи принятия решений*.

Е.1. ЗАДАЧИ ПРИНЯТИЯ РЕШЕНИЙ

Говоря об алгоритмах, возвращающих ответ «да» или «нет», мы попадаем в класс задач, называемых *двоичными (бинарными) задачами принятия решений*.

Алгоритмы двоичной классификации присваивают данным одну из двух меток. Два значения на самом деле могут быть чем угодно, но концептуально это эквивалентно присваиванию логической метки, поэтому в книге используется именно это обозначение.

Исследование только двоичной классификации не является ограничением, потому что мультиклассовые классификаторы можно создать путем объединения нескольких двоичных классификаторов.

Один из фундаментальных результатов в информатике и, в частности, в области вычислимости и сложности состоит в том, что любую задачу оптимизации можно выразить как задачу принятия решения и наоборот.

Например, отыскивая кратчайший путь в графе, можно определить эквивалентную задачу принятия решения, установив пороговое значение T и проверив, существует ли путь с длиной не более T. Решив эту задачу для различных значений T и выбирая эти значения с помощью двоичного поиска, можно найти решение задачи оптимизации, используя алгоритм задачи принятия решения[1].

Е.2. АЛГОРИТМЫ ЛАС-ВЕГАСА

Класс так называемых алгоритмов Лас-Вегаса включает все рандомизированные алгоритмы, которые обладают перечисленными ниже качествами.

1. Всегда дают на выходе правильное решение (или возвращают отказ).
2. Могут потреблять конечное, но непредсказуемое (случайное) количество ресурсов. Важно отметить, что количество необходимых ресурсов нельзя предсказать по входным данным.

---

[1] Если гарантируется, что порог будет целым или рациональным числом, то также возможно уверенно прогнозировать, что решение будет в том же вычислительном классе, что и решение задачи принятия решений, поэтому если, например, есть полиномиальное решение задачи принятия решений, то решение задачи оптимизации тоже потребует полиномиального времени.

Наиболее ярким примером алгоритма Лас-Вегаса может служить рандомизированная быстрая сортировка[1]: она всегда дает правильное решение, но на каждом шаге рекурсии опорная точка выбирается случайным образом, а время выполнения колеблется от $O(n \log n)$ до $O(n^2)$, в зависимости от сделанного выбора.

Е.3. АЛГОРИТМЫ МОНТЕ-КАРЛО

Алгоритмы классифицируются как методы Монте-Карло, если иногда результат их работы может оказаться неправильным. Вероятность неправильного ответа обычно является компромиссом с используемыми ресурсами, и практическим алгоритмам удается поддерживать эту вероятность на низком уровне при использовании разумного объема вычислительной мощности и памяти.

В задачах принятия решений, когда ответом алгоритма Монте-Карло является одно из значений, `true` или `false`, имеем три возможные ситуации:

- алгоритм всегда дает верное значение, если возвращает `false` (так называемый алгоритм *с ложным смещением*);
- алгоритм всегда дает верное значение, если возвращает `true` (так называемый алгоритм с *истинным смещением*);
- алгоритм может вернуть неправильный ответ в обоих случаях.

Алгоритмы Монте-Карло детерминированы по количеству необходимых ресурсов и часто используются как аналоги алгоритмов Лас-Вегаса.

Предположим, у нас есть алгоритм A, который всегда возвращает верное решение, но имеет недетерминированное потребление ресурсов. Если количество ресурсов ограничено, то можно запускать A до тех пор, пока он не выдаст решение или не исчерпает выделенные ресурсы. Например, можно остановить рандомизированную быструю сортировку после $n \log(n)$ перестановок.

Используя такой подход, мы жертвуем точностью (гарантией правильного результата) ради уверенности, что ответ (близкий к оптимальному) будет получен в течение определенного времени (или потребит памяти не больше определенного объема).

Стоит также отметить, что для некоторых рандомизированных алгоритмов нельзя с уверенностью предполагать, остановятся ли они в конце концов и найдут ли решение (хотя обычно это не так). Примером такого алгоритма может служить рандомизированная версия обезьяньей сортировки (Bogosort).

---

[1] *Бхаргава А.* Грокаем алгоритмы. — СПб.: Питер, 2017. — С. 75.

E.4. МЕТРИКИ КЛАССИФИКАЦИИ

При анализе структур данных, таких как фильтры Блума или декартовы деревья, изучение требований их методов к времени и памяти имеет первостепенное значение, но этого недостаточно. Помимо скорости работы и объема потребляемой памяти, подобные алгоритмы Монте-Карло имеют еще одну важную характеристику: качество работы.

Чтобы оценить качество работы, нужно определить *метрики* — функции, измеряющие расстояние между приближенным и оптимальным решениями.

В случае с алгоритмами классификации их качество является мерой, насколько правильно определяется класс каждого входного значения. Таким образом, для двоичной (то есть `true`/`false`) классификации важно, насколько точно алгоритм возвращает `true` для входных точек.

E.4.1. Достоверность

Один из способов измерения качества алгоритма классификации — определение частоты правильных прогнозов. Предположим, что N_P — число точек, действительно принадлежащих классу `true`:

- P_P — число точек, которые алгоритм отнес к классу `true`;
- T_P — число так называемых *истинно положительных* точек, которые алгоритм правильно отнес к классу `true`.

Точно так же пусть N_N — число точек, принадлежащих классу `false`:

- P_N — число точек, которые алгоритм отнес к классу `false`;
- T_N — число так называемых *истинно отрицательных* точек, которые алгоритм правильно отнес к классу `false`.

В этом случае достоверность можно определить так:

$$\text{достоверность} = \frac{T_P + T_N}{N_P + N_N} = \frac{T_P + T_N}{N}.$$

Если достоверность равна 1, это означает, что алгоритм всегда возвращает верный результат.

К сожалению, за исключением этого идеального случая, достоверность не всегда является хорошей мерой качества алгоритмов. Рассмотрим пограничный случай, когда 99 % точек в базе данных фактически принадлежат классу `true` и три классификатора работают так:

- классификатор правильно классифицирует 100 % точек из класса `false` и 98,98 % точек из класса `true`;
- классификатор правильно классифицирует 0,5 % точек из класса `false` и 99,49 % точек из класса `true`;

- классификатор, классифицирующий любые точки как принадлежащие классу `true`.

Удивительно, но последний имеет более высокую достоверность, чем два других, даже притом что неправильно классифицирует все точки из категории `false`.

СОВЕТ

Если хотите убедиться в правоте моих слов, подставьте эти числа в формулу достоверности.

Если использовать эту метрику в машинном обучении, когда обучающая выборка искажена подобным образом[1], то получится ужасная модель или, точнее, модель с плохой способностью к обобщению.

Е.4.2. Точность и полнота

Чтобы улучшить простую достоверность, нужно отслеживать информацию о каждой категории отдельно. С этой целью можно определить две новые метрики:

- *точность* (также называемая *положительной прогностической ценностью*) — определяется как отношение количества правильно предсказанных точек из класса `true` *(истинно положительных)* к общему количеству точек, для которых алгоритм предсказал категорию `true`:

$$\text{точность} = \frac{T_P}{P_P};$$

- *полноту* (также известную как *чувствительность*) — определяется как отношение истинно положительных результатов к общему числу фактически положительных результатов:

$$\text{полнота} = \frac{T_P}{N_P}.$$

Можно дать альтернативное определение точности и полноты, введя количество *ложноположительных* предсказаний F_p, то есть точек, принадлежащих к классу `false`, которые неправильно были отнесены к классу `true`, и количество *ложноотрицательных* предсказаний F_n:

$$\text{точность} = \frac{T_P}{T_P + F_P};$$

$$\text{полпота} = \frac{T_P}{T_P + F_N}.$$

[1] Например, когда один из классов имеет небольшое количество образцов в наборе данных или для него трудно получить данные: такое часто случается в медицине с редкими заболеваниями.

Достоверность оценивает только, насколько хорошо алгоритм справляется с прогнозами, а точность/полнота взвешивают количество успехов с количеством ошибок. На самом деле точность и полнота не являются независимыми, и одну можно определить через другую, поэтому невозможно до бесконечности улучшать обе метрики. В общем случае улучшение полноты влечет ухудшение точности и наоборот.

Для классификаторов машинного обучения можно построить кривую точности/полноты (рис. E.1), а с помощью параметров настраивать модель во время обучения так, чтобы найти компромисс между этими двумя характеристиками.

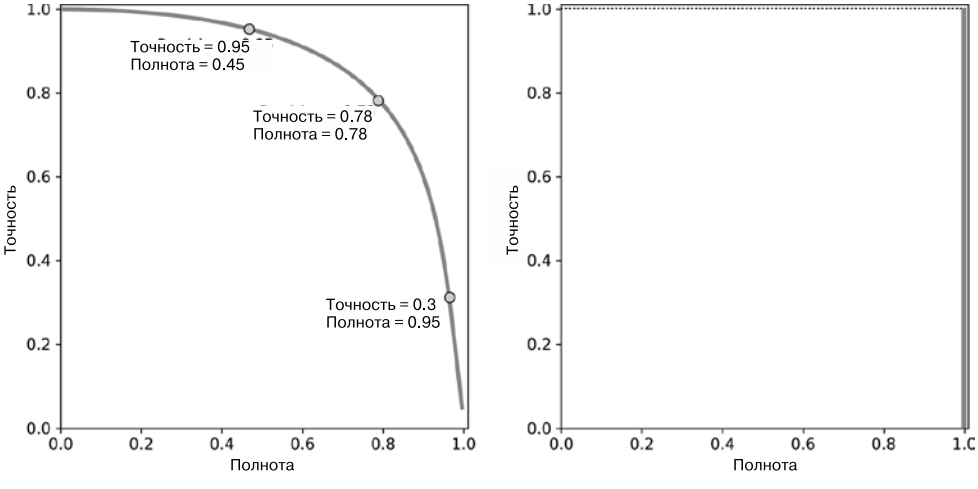


Рис. E.1. Примеры кривой точности/полноты. Кривая *слева* — обобщенная кривая для гипотетического алгоритма. Она построена путем перебора параметров алгоритма и записи полученных значений точности и полноты. Выполнив множество попыток и записав результаты, можно получить кривую, похожую на эту. В общем случае набор параметров модели, улучшающий точность, будет несколько ухудшать полноту и наоборот. *Справа* показана вырожденная кривая точности/полноты для алгоритма с ложным смещением, такого как фильтры Блума; его полнота всегда равна 1 для любой точности. Алгоритмы с истинным смещением имеют симметричную кривую точности/полноты

Итак, в случае двоичной классификации идеальная (100 %) точность означает, что каждая точка, для которой предсказывается принадлежность к классу `true`, действительно принадлежит этому классу. Точно так же полнота составляет 100 %, если алгоритм никогда не дает ложноотрицательных результатов.

В качестве упражнения попробуйте вычислить точность и полноту для примеров из предыдущих разделов и посмотрите, насколько больше информации они дают о качестве модели, чем одна лишь оценка достоверности.

Если в качестве примера взять фильтры Блума (см. главу 4), то в разделе 4.8 я рассказываю, что он всегда правильно классифицирует точки, принадлежащие к категории `false`, поэтому фильтры Блума имеют полноту 100 %.

Их точность, к сожалению, не столь совершенна. В разделе 4.10 я подробно объясняю, как ее вычислить. Однако в этом случае, как и для всех алгоритмов с ложным смещением, компромисс между полнотой и точностью невозможен и повысить точность можно только за счет увеличения используемых ресурсов.

Е.4.3. Другие метрики и подведение итогов

Есть, конечно, и другие метрики, которые могут пригодиться при разработке классификаторов. Одной из наиболее полезных является F-мера[1], объединяющая точность и полноту в одной формуле. Однако обсуждение этих альтернативных метрик выходит за рамки обсуждаемых в книге тем.

Если продолжить параллель с фильтрами Блума, то в отношении метрик, представленных в этом контексте, можно сказать следующее:

- *достоверность (accuracy)* отвечает на вопрос «Какой процент вызовов `contains` возвращает правильный результат?»;
- *точность (precision)* отвечает на вопрос «Какой процент вызовов `contains` возвращает `true` правильно?»;
- *полнота (recall)* отвечает на вопрос «Среди всех вызовов `contains` для элементов, фактически содержащихся в фильтре, какой процент из них вернул значение `true`?» (как мы знаем, полнота фильтров Блума всегда равна 100 %).

[1] https://en.wikipedia.org/wiki/F1_score.

Марчелло Ла Рокка

Продвинутые алгоритмы и структуры данных

Перевела с английского Л. Киселева

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Куликова*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Т. Никифорова, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 21.11.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 68,370. Тираж 700. Заказ 0000.

Джей Венгроу

ПРИКЛАДНЫЕ СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ. ПРОКАЧИВАЕМ НАВЫКИ

Структуры данных и алгоритмы — это не абстрактные концепции, а турбина, способная превратить ваш софт в болид «Формулы-1». Научитесь использовать нотацию «О большое», выбирайте наиболее подходящие структуры данных, такие как хеш-таблицы, деревья и графы, чтобы повысить эффективность и быстродействие кода, что критически важно для современных мобильных и веб-приложений.

Книга полна реальных прикладных примеров на популярных языках программирования (Python, JavaScript и Ruby), которые помогут освоить структуры данных и алгоритмы и начать применять их в повседневной работе. Вы даже найдете слово, которое может существенно ускорить ваш код. Практикуйте новые навыки, выполняя упражнения и изучая подробные решения, которые приводятся в книге.

Начните использовать эти методы уже сейчас, чтобы сделать свой код более производительным и масштабируемым.

КУПИТЬ

Джордж Хайнеман

АЛГОРИТМЫ. С ПРИМЕРАМИ НА PYTHON

Когда нужно, чтобы программа работала быстро и занимала поменьше памяти, профессионального программиста выручают знание алгоритмов и практика их применения. Эта книга — как раз про практику. Ее автор, Джордж Хайнеман, предлагает краткое, но четкое и последовательное описание основных алгоритмов, которые можно эффективно использовать в большинстве языков программирования. О том, какими методами решаются различные вычислительные задачи, стоит знать и разработчикам, и тестировщикам, и интеграторам.