Классические задачи Computer Science на языке Python

Дэвид Копец

MANNING

Classic Computer Science Problems in Python

DAVID KOPEC

MANNING
SHELTER ISLAND

Дэвид Копец

Классические задачи Computer Science на языке Python

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2020

ББК 32.973.2-018.1
УДК 004.43
К65

Копец Дэвид

К65 Классические задачи Computer Science на языке Python. — СПб.: Питер, 2020. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1428-3

Многие задачи в области Computer Science, которые на первый взгляд кажутся новыми или уникальными, на самом деле уходят корнями в классические алгоритмы, методы кодирования и принципы разработки. И устоявшиеся техники по-прежнему остаются лучшим способом решения таких задач! Научитесь писать оптимальный код для веб-разработки, обработки данных, машинного обучения и других актуальных сфер применения Python.

Книга даст вам возможность глубже освоить язык Python, проверить себя на испытанных временем задачах, упражнениях и алгоритмах. Вам предстоит решать десятки заданий по программированию: от самых простых (например, найти элементы списка с помощью двоичной сортировки), до сложных (выполнить кластеризацию данных методом k-средних). Прорабатывая примеры, посвященные поиску, кластеризации, графам и пр., вы вспомните то, о чем успели позабыть, и овладеете классическими приемами решения повседневных задач.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1617295980 англ.
ISBN 978-5-4461-1428-3

Краткое содержание

Оглавление

Посвящается моей бабушке Эрминии Антос,
которая всю жизнь училась сама и учила других.

Благодарности

Я благодарю всех сотрудников издательства Manning, которые помогали в создании этой книги: Шерил Вейсман, Дирдре Хайам, Кэти Теннант, Дотти Марсико, Дженет Вейл, Барбару Мирецки, Александра Драгосавлевича, Мэри Пирджис и Марию Тюдор.

Я благодарю специалиста по отбору новых книг Брайана Сойера, который предложил нам переключиться на Python после того, как я закончил книгу о Swift. Спасибо редактору по развитию Дженнифер Стаут за неизменно доброжелательное отношение. Благодарю научного редактора Фрэнсис Буонтемпо — она внимательно изучила каждую главу и на каждом шагу давала подробные полезные комментарии. Я благодарю литературного редактора Энди Кэрролла, чье исключительное внимание к деталям как в книге о Swift, так и в этой позволило отловить несколько допущенных мной ошибок. Спасибо также корректору Хуану Руфесу.

Эту книгу рецензировали Эл Кринкер, Эл Пезевски, Алан Богусевич, Брайан Канада, Крейг Хендерсон, Дэниэл Кенни-Юнг, Эдмонд Сисей, Ева Барановска, Гэри Барнхарт, Джефф Кларк, Джеймс Уотсон, Джеффри Лим, Иенс Кристиан, Бредал Мэдсен, Хуан Хименес, Хуан Руфес, Мэтт Лемке, Майур Патил, Майкл Брайт, Роберто Касадей, Сэм Зейдел, Торстен Вебер, Том Джеффрис и Уилл Лопес. Спасибо всем, кто высказал конструктивную и конкретную критику во время создания этого издания. Ваши отзывы были учтены.

Я благодарю свою семью, друзей и коллег, которые вдохновили меня взяться за написание этой книги сразу же после публикации *Classic Computer Science Problems in Swift*. Я благодарю своих подписчиков в «Твиттере» и в других сетях за то, что поддерживали меня и помогали в работе над книгой — в большом и малом. Огромное спасибо моей жене Ребекке Копец и маме Сильвии Копец, которые всегда поддерживают все мои проекты.

Мы создали эту книгу за довольно короткое время. Основная ее часть была написана летом 2018 года. Я ценю, что издательство Manning согласилось сократить свой обычно гораздо более длительный технологический процесс, чтобы я мог работать в удобном для себя режиме. Я знаю, что это затронуло всю команду, поскольку всего за несколько месяцев мы провели три этапа проверок на разных уровнях с привлечением множества разных людей. Большинство читателей были бы поражены тем, сколько разных видов проверок проходит техническая книга в обычном издательстве и какое количество людей принимает участие в ее рецензировании и пересмотре. Я благодарю всех работников издательства, всех официальных рецензентов и всех, кто участвовал в процессе подготовки издания.

Наконец, и это самое главное, я благодарю своих читателей за покупку книги. Я думаю, что в мире, полном унылых электронных учебников, важно поддерживать выпуск бумажных книг, позволяющих автору зазвучать в полный голос. Онлайн-учебники могут быть отличным ресурсом, но благодаря вашему спросу полноформатные, проверенные и тщательно сверстанные бумажные книги по-прежнему занимают свое место в обучении программированию.

Об этой книге

Торговые марки

В издании упомянуты различные торговые марки. Символ торговой марки при каждом появлении фирменного названия не ставится, так как эти названия используются только в целях издания и в интересах владельца товарного знака, без намерения посягнуть на товарный знак. Слово Python является зарегистрированным товарным знаком Python Software Foundation. Connect Four — торговая марка Hasbro, Inc.

Форум этой книги

Покупка книги включает бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять комментарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите на https://www.manning.com/books/classic-computerscience-problems-in-Python. Узнать больше о других форумах на сайте издательства Manning и познакомиться с правилами вы сможете на странице https://forums.manning.com/forums/about.

Издательство Manning обязуется предоставить своим читателям место встречи, на которой может состояться содержательный диалог между отдельными читателями и между читателями и автором. Однако со стороны автора отсутствуют какие-либо обязательства уделять форуму внимание в каком-то определенном объеме — его присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать автору неожиданные вопросы, чтобы его интерес не угасал! Форум и архивы предыдущих обсуждений будут доступны на сайте издательства, пока книга находится в печати.

Об авторе

Дэвид Копец – старший преподаватель на кафедре компьютерных наук и инноваций в колледже Шамплейн в Берлингтоне, штат Вермонт. Он опытный разработчик программного обеспечения и автор книг *Classic Computer Science Problems in Swift* (Manning, 2018) и *Dart for Absolute Beginners* (Apress, 2014). Дэвид получил степень бакалавра экономики и степень магистра компьютерных наук в Дартмутском колледже. Вы можете связаться с ним в «Твиттере» по имени @davekopec.

Об иллюстрации на обложке

Иллюстрация на обложке называется «Одежда китайского бонзы или священника»[1] и позаимствована из книги «Коллекция платья разных народов, старинного и современного»[2], изданного в Лондоне в 1757–1772 годах. Как указано на титульном листе, это гравюры, выполненные на медных пластинах и раскрашенные гуммиарабиком.

Томаса Джеффериса (1719–1771) называли географом короля Георга III. Он был английским картографом, ведущим поставщиком карт своего времени. Он гравировал и печатал карты для правительственных и других государственных учреждений, выпускал широкий спектр коммерческих карт и атласов, особенно Северной Америки. В ходе работы интересовался особенностями одежды населения тех земель, которые обследовал и нанес на карту. Зарисовки костюмов блестяще представлены в этом издании. В конце XVIII века увлечение далекими землями и путешествия ради удовольствия были относительно новым явлением, и коллекции, подобные этой, были популярны, позволяя как туристам, так и тем, кто путешествует, не вставая с кресла, познакомиться с жителями других стран.

Разнообразие иллюстраций в книгах Джеффериса – яркое свидетельство уникальности и оригинальности народов мира в то время. С тех пор тенденции в одежде сильно изменились, а региональные и национальные различия, которые

---

[1] Habit of a Bonza or Priest in China.

[2] A Collection of the Dresses of Different Nations, Ancient and Modern.

были такими значимыми 200 лет назад, постепенно сошли на нет. В наши дни часто сложно отличить друг от друга жителей разных континентов. Оптимисты могут сказать, что взамен культурному и визуальному многообразию мы получили более насыщенную и интересную личную жизнь (или по крайней мере ее интеллектуальную и техническую стороны).

В то время как большинство компьютерных изданий мало чем отличаются друг от друга, компания Manning выбирает для своих книг обложки, показывающие богатое региональное разнообразие, которое Джефферис воплотил в своих иллюстрациях два столетия назад. Это ода находчивости и инициативности современной компьютерной индустрии.

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.

Введение

Благодарю вас за покупку книги «Классические задачи Computer Science на языке Python». Python — один из самых популярных языков программирования, и программистами на Python становятся специалисты из абсолютно разных областей. У некоторых из них есть специальное техническое образование, другие изучают Python на досуге, третьи используют его в профессиональной среде, но не для разработки программного обеспечения. Задачи, описанные в этой книге, среднего уровня. С их помощью опытные программисты смогут освежить свои знания в области информатики, изучив некоторые расширенные возможности языка. Программисты-самоучки смогут быстрее обучиться программированию, рассматривая решение классических задач на выбранном языке — Python. Эта книга охватывает такое разнообразие методов решения задач, что каждый найдет в ней что-то для себя.

Это издание не является введением в Python. На эту тему есть множество других превосходных книг, выпущенных как в Manning, так и в других издательствах[1]. В данной книге предполагается, что вы уже программист среднего или высокого уровня на Python. Для примеров использовалась версия Python 3.7, но вам не нужно знать все тонкости последней версии языка. Фактически примеры были подобраны так, чтобы они послужили учебным материалом, помогающим читателям достичь высокого уровня мастерства. Однако эта книга не подходит для читателей, совершенно не знакомых с Python.

---

[1] Если вы только начинаете знакомство с Python, то, возможно, предварительно захотите прочитать книгу Наоми Седер The Quick Python Book (Naomi Ceder, 3rd ed. Manning, 2018).

Почему именно Python

Python используется в разных сферах деятельности: в анализе и обработке данных, кинопроизводстве, компьютерном обучении, управлении в сфере информационных технологий и многих других. В сущности, нет области применения программирования, в которой бы он не задействовался (за исключением, может быть, разработки ядер операционных систем). Python любят за гибкость, красивый и лаконичный синтаксис, объектно-ориентированную чистоту и кипучую деятельность сообщества программистов. Активное сообщество важно — это означает, что Python приветствует новичков и в нем предусмотрена обширная система доступных библиотек для разработчиков.

Из-за всего этого Python иногда рассматривается как язык для начинающих, и такая характеристика, вероятно, верна. Большинство людей согласятся с тем, что Python легче изучать, чем, например, C++, и сообщество этого языка действительно более дружелюбно относится к новичкам. В результате многие осваивают Python, потому что он доступен, и начинают писать на нем программы, которые желательно получить довольно быстро. Но эти люди, возможно, никогда не учились информатике (computer science) и не знают о существовании эффективных методов решения задач. Если вы один из тех программистов, которые знают Python, но не знакомы с информатикой, то эта книга для вас.

Другие изучают Python как второй, третий, четвертый или пятый язык после долгой работы в области создания программного обеспечения. Для них встреча с задачами, которые уже приходилось решать на другом языке, поможет ускорить изучение Python. А эта книга может стать хорошим способом освежить знания перед собеседованием или раскрыть отдельные методы решения задач, которые они раньше не предполагали использовать в своей работе. Я бы посоветовал таким читателям просмотреть оглавление, чтобы увидеть, есть ли в этой книге темы, которые их интересуют.

Что такое классическая задача программирования

Есть мнение, что компьютеры в информатике — то же самое, что телескопы в астрономии. Если это так, то, возможно, язык программирования подобен объективу телескопа. В любом случае термин «классические задачи программирования» здесь означает «задачи программирования, которые обычно преподают в курсе информатики для студентов».

Существуют задачи программирования, которые предлагают решить начинающим программистам и которые уже стали достаточно распространенными, чтобы считаться классическими, как в аудитории во время экзамена на степень бакалавра (в области информатики, разработки программного обеспечения и т. п.), так и в рамках учебника среднего уровня по программированию (например, в книге

начального уровня по искусственному интеллекту или алгоритмам). В данном издании вы найдете набор именно таких задач.

Задачи варьируются от тривиальных, которые решаются написанием нескольких строк кода, до сложных, требующих построения систем на протяжении нескольких глав. Одни относятся к области искусственного интеллекта, другие требуют простого здравого смысла. Некоторые из них практичны, другие — причудливы.

Какие задачи представлены в этой книге

Глава 1 познакомит вас с методами решения задач, которые, вероятно, известны большинству читателей. Такие вещи, как рекурсия, запоминание и манипулирование битами, являются важными строительными блоками других методов, которые будут рассмотрены в последующих главах.

За этим плавным вступлением идет глава 2, посвященная задачам поиска. Тема поиска столь обширна, что под ее заголовком, вероятно, можно поместить большинство задач этой книги. В главе 2 представлены наиболее важные алгоритмы поиска, включая бинарный поиск, поиск в глубину, поиск в ширину и A*. Эти алгоритмы будут часто использоваться в остальной части книги.

В главе 3 мы построим структуру для решения широкого круга задач, которые могут быть абстрактно определены переменными ограниченных областей изменения (limited domains). Это такая классика, как задача восьми ферзей, задача о раскрашивании карты Австралии и криптоарифметика SEND + MORE = MONEY.

В главе 4 исследуется мир графовых алгоритмов, применение которых оказывается удивительно широким для непосвященных. В этой главе вы построите графовую структуру данных и с ее помощью решите несколько классических задач оптимизации.

В главе 5 исследуются генетические алгоритмы — менее детерминистская методика, чем большинство описанных в этой книге. Она иногда позволяет решить задачи, с которыми традиционные алгоритмы не способны справиться за разумное время.

Глава 6 посвящена кластеризации *k-средних* и, пожалуй, является самой алгоритмически специфичной главой в этой книге. Данный метод кластеризации прост в реализации, легок для понимания и широко применим.

Глава 7 призвана объяснить, что такое нейронная сеть, и дать читателю представление о том, как выглядит простейшая нейронная сеть. Здесь не ставится цель всесторонне раскрыть эту захватывающую развивающуюся область. В этой главе вы построите нейронную сеть на чисто теоретической основе, не используя внешние библиотеки, чтобы по-настоящему прочувствовать, как работает нейронная сеть.

Глава 8 посвящена состязательному поиску в идеальных информационных играх для двух игроков. Вы изучите алгоритм поиска, известный как минимакс,

который можно применять для разработки искусственного противника, способного хорошо играть в такие игры, как шахматы, шашки и Connect Four.

Глава 9 охватывает интересные и забавные задачи, которые не совсем вписываются в остальные главы этой книги.

Для кого эта книга

Эта книга предназначена для программистов среднего и высокого уровня. Опытные специалисты, которые хотят углубить свое знание Python, найдут здесь задачи, приятно знакомые со времен обучения информатике или программированию. Программисты среднего уровня познакомятся с этими классическими задачами на выбранном языке — Python. Для разработчиков, которые готовятся к собеседованию по программированию, издание, скорее всего, станет ценным подготовительным материалом.

Кроме профессиональных программистов, эту книгу могут счесть полезной студенты, обучающиеся по программам бакалавриата по информатике и интересующиеся Python. Она не претендует на роль строгого введения в структуры данных и алгоритмы. *Это не учебник по структурам данных и алгоритмам.* Вы не найдете на ее страницах доказательств теорем или обильного использования нотаций O большого (big-O). Напротив, эта книга позиционируется как доступное практическое руководство по методам решения задач, которые должны стать конечным продуктом изучения структуры данных, алгоритмов и классов искусственного интеллекта.

Подчеркну еще раз: предполагается, что читателям знакомы синтаксис и семантика Python. Читатель с нулевым опытом программирования едва ли извлечет пользу из этой книги, а программисту с нулевым опытом в Python наверняка будет трудно. Другими словами, «Классические задачи Computer Science на языке Python» — это книга для программистов, работающих на Python, и студентов, изучающих информатику.

Версии Python, хранилище исходного кода и аннотации типов

Исходный код в этой книге написан в соответствии с версией 3.7 языка Python. В нем используются функции, которые стали доступными только в Python 3.7, поэтому часть кода не будет работать на более ранних версиях Python. Вместо того чтобы пытаться заставить примеры работать в более ранней версии, просто загрузите последнюю версию Python, прежде чем начинать работу с книгой.

В этой книге применяется только стандартная библиотека Python (за небольшим исключением — в главе 2 установлен модуль typing_extensions), поэтому весь код этой книги должен работать на любой платформе, где поддерживается Python (macOS, Windows, GNU/Linux и т. п.). Код был протестирован только на CPython

(основной интерпретатор Python, доступный на python.org), хотя, скорее всего, бóльшая его часть будет работать в любой версии другого интерпретатора Python, совместимой с Python 3.7.

В книге не объясняется, как использовать инструменты Python, такие как редакторы, IDE, отладчики и Python REPL. Исходный код книги доступен в Интернете в репозитории GitHub по адресу https://github.com/davecom/ClassicComputerScienceProblemsInPython. Исходный код разделен на папки по главам. Имена исходных файлов вы увидите в тексте глав в заголовках листингов кода. Эти исходные файлы находятся в соответствующих папках хранилища. Для того чтобы запустить выполнение задачи, просто введите `python3 имяфайла.py` или `python имяфайла.py` в зависимости от того, как на вашем компьютере называется интерпретатор Python 3.

Во всех листингах кода в этой книге задействуются аннотации (иногда говорят подсказки) типов Python, также известные как сигнатуры типов. Это относительно новая функция языка Python, и она может смутить программистов Python, которые никогда не встречали ее раньше. Аннотации типов применяются по следующим причинам.

1. Они дают ясное представление о типах переменных, параметрах функций и возвращаемых функциями значений.
2. Вследствие п. 1 они в некотором смысле автоматически документируют код. Вместо того чтобы искать возвращаемый тип функции в комментарии или строке документации, можно просто посмотреть на ее сигнатуру.
3. Они позволяют проверять корректность кода. Одно из популярных средств проверки типов в Python — mypy.

Не всем нравятся аннотации типов, и решение использовать их в книге было, честно говоря, рискованным. Я надеюсь, что они скорее помогут, чем помешают. Писать программу на Python с аннотациями типов немного дольше, но при чтении кода это обеспечивает бóльшую ясность. Интересно отметить, что аннотации типов не влияют на фактическое выполнение кода в интерпретаторе Python. Вы можете удалить их из любого кода в этой книге, и он все равно должен работать. Если вы никогда прежде не встречали аннотации типов и чувствуете, что вам нужно более подробно с ними познакомиться, прежде чем углубиться в книгу, обратитесь к приложению B, в котором содержится ускоренный курс по аннотациям типов.

Никакой графики и пользовательских интерфейсов — только стандартная библиотека

В этой книге нет примеров, которые формировали бы графический вывод или использовали графический пользовательский интерфейс (GUI). Почему? Цель ее состоит в том, чтобы решить поставленные задачи с помощью максимально кратких и удобочитаемых методов. Зачастую вывод графики мешает или делает решения

значительно более сложными, чем нужно, чтобы проиллюстрировать рассматриваемые методику или алгоритм.

Кроме того, если не применять какую-либо платформу с графическим интерфейсом, то весь код, представленный в этой книге, становится в высшей степени переносимым. Он может так же легко работать на дистрибутиве Python, встроенном в Linux, как и на компьютере под управлением Windows. Кроме того, было принято сознательное решение использовать пакеты из стандартной библиотеки Python вместо любых внешних библиотек, как это делается в большинстве книг по углубленному изучению Python. Почему? Потому что цель этой книги состоит в том, чтобы научить читателя методам решения задач на основе базовых принципов, а не способом «установить решение командой pip install». Я надеюсь, что благодаря необходимости решать каждую задачу с нуля вы начнете понимать, что происходит внутри популярных библиотек. Как минимум, работая в этой книге только со стандартной библиотекой, мы получим лучше переносимый и более простой в применении код.

Это не означает, что в некоторых случаях графические решения более наглядны для реализации выбранного алгоритма, чем текстовые решения. Просто они не являются темой этой книги, и их использование лишь усложнило бы ее.

Книги этой серии

Это вторая книга из серии «Классические задачи Computer Science», опубликованной издательством Manning. Первой была *Classic Computer Science Problems in Swift*, вышедшая в 2018 году. В каждой книге этой серии мы стремимся представить видение конкретного языка в свете решения одних и тех же (в основном) задач программирования.

Если вам понравится эта книга и вы решите изучить другой язык, описанный в одной из книг серии, переход от одной к другой может оказаться простым способом улучшить знания этого языка. В настоящее время серия охватывает только Swift и Python. Первые две книги я написал сам, потому что имею значительный опыт работы с обоими языками, но мы уже обсуждаем создание новых книг серии в соавторстве с людьми, которые являются экспертами в других языках. Если вам понравится эта книга, я призываю вас следить за выходом других изданий серии. Для получения дополнительной информации о них посетите сайт https://classicproblems.com/.

1 Простые задачи

Для начала рассмотрим несколько простых задач, которые можно решить с помощью нескольких сравнительно коротких функций. Несмотря на свою простоту, эти задачи все же позволят нам изучить некоторые интересные методы решения. Считайте их хорошей разминкой.

1.1. Ряд Фибоначчи

Ряд Фибоначчи — это такая последовательность чисел, в которой любое число, кроме первого и второго, является суммой двух предыдущих:

```
0, 1, 1, 2, 3, 5, 8, 13, 21...
```

Первым числом в последовательности Фибоначчи является 0, четвертым — 2. Отсюда следует, что для получения значения любого числа *n* в последовательности Фибоначчи можно использовать следующую формулу:

```
fib(n) = fib(n - 1) + fib(n - 2)
```

1.1.1. Первый вариант рекурсии

Приведенная формула для определения числа в последовательности Фибоначчи (рис. 1.1) представляет собой псевдокод, который можно легко перевести в *рекурсивную* функцию Python. (Рекурсивная функция — это функция, которая вызывает

сама себя.) Такой механический перевод станет нашей первой попыткой написать функцию, возвращающую заданное значение последовательности Фибоначчи (листинг 1.1).

Листинг 1.1. fib1.py

```
def fib1(n: int) -> int:
    return fib1(n - 1) + fib1(n - 2)
```

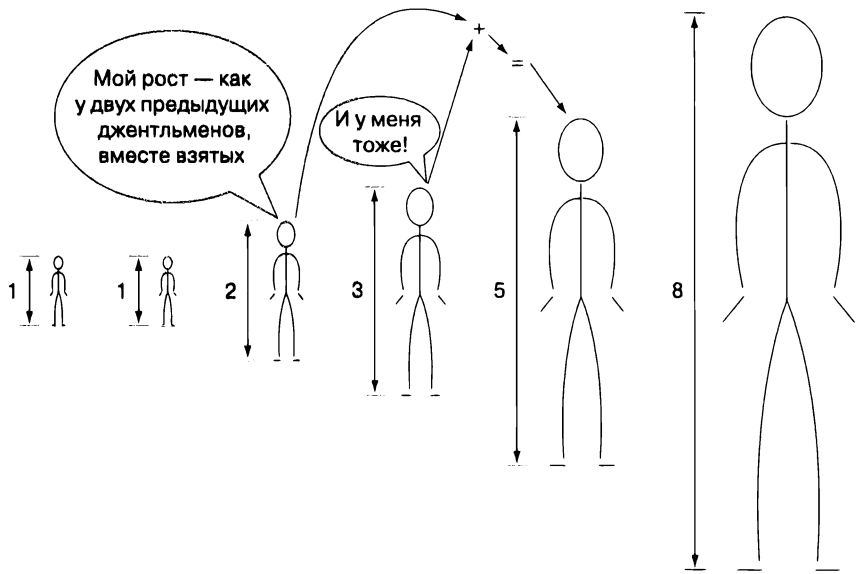


Рис. 1.1. Рост каждого человечка равен сумме роста двух предыдущих

Попробуем запустить эту функцию, вызвав ее и передав ей значение (листинг 1.2).

Листинг 1.2. fib1.py (продолжение)

```
if __name__ == "__main__":
    print(fib1(5))
```

Ой-ой-ой! При попытке запустить `fib1.py` получаем ошибку превышения максимальной глубины рекурсии:

```
RecursionError: maximum recursion depth exceeded
```

Проблема в том, что функция `fib1()` будет работать вечно, не возвращая окончательный результат. Каждый вызов `fib1()` приводит к двум другим вызовам `fib1()`, и так без конца. Такую ситуацию называют *бесконечной рекурсией* (рис. 1.2), она подобна *бесконечному циклу*.

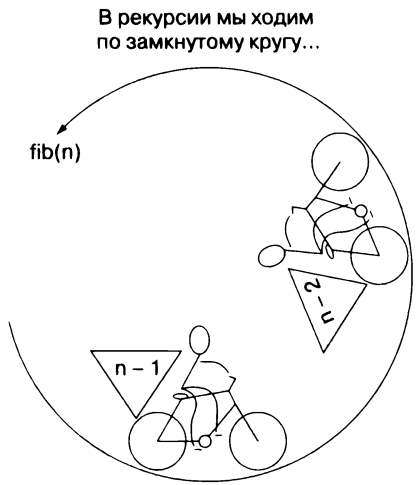


Рис. 1.2. Рекурсивная функция fib(n) вызывает сама себя с аргументами n – 2 и n – 1

1.1.2. Использование базовых случаев

Обратите внимание: до тех пор пока мы не запустили `fib1()`, среда Python никак не сообщает о том, что с функцией что-то не так. Избегать бесконечной рекурсии — обязанность программиста, а не компилятора или интерпретатора. Причина бесконечной рекурсии заключается в том, что мы нигде не указали базовый случай. В рекурсивной функции базовый случай служит точкой остановки.

Естественные базовые случаи для функции Фибоначчи — это два первых специальных значения последовательности — 0 и 1. Ни 0, ни 1 не являются суммой двух предыдущих чисел последовательности. Это два первых специальных значения. Давайте попробуем указать их в качестве базовых случаев (листинг 1.3).

Листинг 1.3. fib2.py

```
def fib2(n: int) -> int:
    if n < 2: # базовый случай
        return n
    return fib2(n - 2) + fib2(n - 1)
    # рекурсивный случай
```

ПРИМЕЧАНИЕ

В отличие от первоначального предложения версия fib2() функции Фибоначчи возвращает 0 для аргумента, равного нулю (fib2(0)), а не единице. В контексте программирования это имеет смысл, поскольку мы привыкли к последовательностям, начинающимся с нулевого элемента.

Функция `fib2()` может быть успешно вызвана и возвращает правильные результаты. Попробуйте вызвать ее для нескольких небольших значений (листинг 1.4).

Листинг 1.4. fib2.py (продолжение)

```
if __name__ == "__main__":
    print(fib2(5))
    print(fib2(10))
```

Не пытайтесь вызывать `fib2(50)`. Это никогда не закончится! Почему? Потому что каждый вызов `fib2()` приводит к двум новым вызовам `fib2()` — рекурсивным вызовам `fib2(n-1)` и `fib2(n-2)` (рис. 1.3). Другими словами, дерево вызовов растет в геометрической прогрессии. Например, вызов `fib2(4)` приводит к такой последовательности вызовов:

```
fib2(4) -> fib2(3), fib2(2)
fib2(3) -> fib2(2), fib2(1)
fib2(2) -> fib2(1), fib2(0)
fib2(2) -> fib2(1), fib2(0)
fib2(1) -> 1
fib2(1) -> 1
fib2(1) -> 1
fib2(0) -> 0
fib2(0) -> 0
```

Если посчитать их (и проследить, добавив несколько вызовов `print`), то обнаружится, что для вычисления всего лишь четвертого элемента нужны девять вызовов `fib2()`! Дальше — хуже. Для вычисления пятого элемента требуется 15 вызовов, десятого — 177, двадцатого — 21 891. Мы могли бы написать и что-нибудь получше.

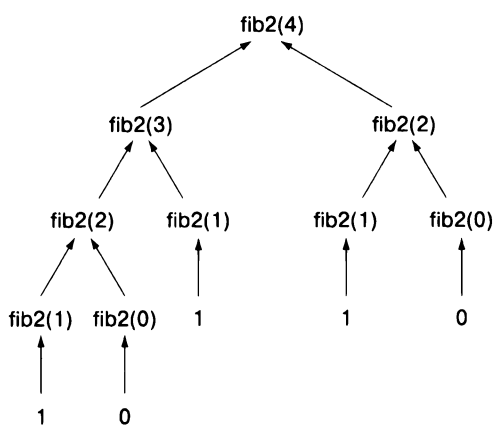


Рис. 1.3. Каждый вызов fib2(), не являющийся базовым случаем, приводит к еще двум вызовам fib2()

1.1.3. Спасение — в мемоизации

Мемоизация — это метод, при котором сохраняются результаты выполненных вычислений, так что, когда они снова понадобятся, их можно найти, вместо того чтобы вычислять во второй (или миллионный) раз (рис. 1.4)[1].

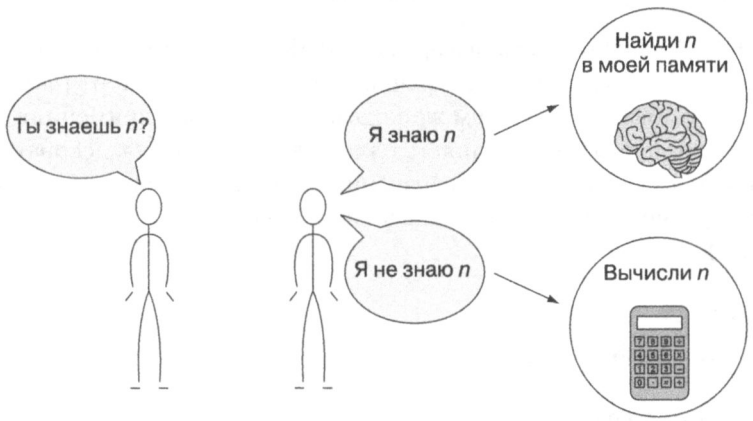


Рис. 1.4. Механизм мемоизации для людей

Создадим новую версию функции Фибоначчи, которая использует словарь Python для целей мемоизации (листинг 1.5).

Листинг 1.5. fib3.py

```
from typing import Dict
memo: Dict[int, int] = {0: 0, 1: 1} # базовые случаи

def fib3(n: int) -> int:
    if n not in memo:
        memo[n] = fib3(n - 1) + fib3(n - 2) # мемоизация
    return memo[n]
```

Теперь можно смело вызывать `fib3(50)` (листинг 1.6).

Листинг 1.6. fib3.py (продолжение)

```
if __name__ == "__main__":
    print(fib3(5))
    print(fib3(50))
```

[1] Термин «мемоизация» придумал Дональд Мичи — известный британский специалист в области информатики. Источник: *Michie D.* Memo functions: a language feature with "rote-learning" properties. — Edinburgh University, Department of Machine Intelligence and Perception, 1967.

Вызов `fib3(20)` даст только 39 вызовов `fib3()`, а не 21 891, как в случае вызова `fib2(20)`. Словарь `memo` изначально заполняется базовыми вариантами 0 и 1, избавляя `fib3()` от излишней сложности, связанной со вторым оператором `if`.

1.1.4. Автоматическая мемоизация

Функцию `fib3()` можно сделать еще проще. В Python есть встроенный декоратор для автоматической мемоизации любой функции. В `fib4()` декоратор `@functools.lru_cache()` использован точно с тем же кодом, который мы применили в `fib2()`. Каждый раз, когда `fib4()` выполняется для нового аргумента, декоратор выполняет кэширование возвращаемого значения. При последующих вызовах `fib4()` для того же аргумента сохраненное значение извлекается из кэша и возвращается (листинг 1.7).

Листинг 1.7. fib4.py

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib4(n: int) -> int: # такое же определение, как в fib2()
    if n < 2: # базовый случай
        return n
    return fib4(n - 2) + fib4(n - 1) # рекурсивный случай

if __name__ == "__main__":
    print(fib4(5))
    print(fib4(50))
```

Обратите внимание: мы можем мгновенно вычислить `fib4(50)`, несмотря на то что тело функции Фибоначчи такое же, как и в `fib2()`. Свойство `@lru_cache maxsize` определяет, сколько последних вызовов функции, которые оно декорирует, должно быть закэшировано. Значение `None` означает отсутствие ограничений.

1.1.5. Будьте проще, Фибоначчи!

Существует еще более быстрый вариант. Мы можем решить задачу Фибоначчи старым добрым итеративным методом (листинг 1.8).

Листинг 1.8. fib5.py

```
def fib5(n: int) -> int:
    if n == 0: return n # специальный случай
    last: int = 0 # начальное значение fib(0)
    next: int = 1 # начальное значение fib(1)
```

```
    for _ in range(1, n):
        last, next = next, last + next
    return next

if __name__ == "__main__":
    print(fib5(5))
    print(fib5(50))
```

ПРЕДУПРЕЖДЕНИЕ

В теле цикла for в fib5() используется распаковка кортежа — возможно, слишком хитроумным способом. Кому-то может показаться, что это сделано для краткости в ущерб удобочитаемости. Другие полагают, что краткость сама по себе улучшает удобство чтения. Суть в том, что переменной last присваивается предыдущее значение next, а next — предыдущее значение last плюс предыдущее значение next. Это позволяет избежать создания временной переменной для хранения старого значения next после изменения last, но перед изменением next. Такое применение распаковки кортежа для определенных переменных широко распространено в Python.

При таком подходе тело цикла `for` будет выполняться максимум $n - 1$ раз. Другими словами, это самая эффективная версия. Сравните 19 проходов тела цикла `for` с 21 891 рекурсивным вызовом `fib2()` для 20-го числа Фибоначчи. В реальных приложениях это может серьезно изменить ситуацию!

В рекурсивных решениях мы приступали к задаче с конца. В этом итеративном решении начинаем с начала. Иногда рекурсия является наиболее интуитивно понятным способом решения задачи. Например, суть функций `fib1()` и `fib2()` — это, в сущности, просто механический перевод исходной формулы Фибоначчи. Однако наивные рекурсивные решения могут значительно ухудшить производительность. Помните, что любую задачу, которая может быть решена рекурсивно, можно решить и итеративным способом.

1.1.6. Генерация чисел Фибоначчи с помощью генератора

До сих пор мы писали функции, которые выводят одно значение из последовательности Фибоначчи. А что, если вместо этого мы хотим вывести всю последовательность вплоть до некоторого значения? С помощью оператора `yield` функцию `fib5()` можно легко преобразовать в генератор Python. В процессе итерирования генератора на каждой итерации будет выводиться значение последовательности Фибоначчи с использованием оператора `yield` (листинг 1.9).

Листинг 1.9. fib6.py

```
from typing import Generator

def fib6(n: int) -> Generator[int, None, None]:
    yield 0 # специальный случай
    if n > 0: yield 1 # специальный случай
    last: int = 0 # начальное значение fib(0)
    next: int = 1 # начальное значение fib(1)
    for _ in range(1, n):
        last, next = next, last + next
        yield next # главный этап генерации

if __name__ == "__main__":
    for i in fib6(50):
        print(i)
```

Если запустить `fib6.py`, то будет напечатано первое 51 число из последовательности Фибоначчи. На каждой итерации цикла `for i in fib6(50):` функция `fib6()` выполняется до оператора `yield`. Если достигнут конец функции и операторов `yield` больше нет, то цикл завершает итерирование.

1.2. Простейшее сжатие

Зачастую важна бывает экономия места — виртуального или реального. Использовать меньше места означает более эффективно работать и экономить деньги. Если вы арендуете квартиру большей площади, чем нужно для ваших семьи и вещей, то можете ужаться до жилья меньшего размера, которое стоит дешевле. Если вы побайтно платите за хранение данных на сервере, то можете сжать данные так, чтобы их хранение обходилось дешевле. *Сжатие* — это процесс получения данных и их кодирования (изменения формы) таким образом, чтобы они занимали меньше места. *Распаковка* предусматривает обратный процесс — возвращение данных в исходную форму.

Если сжатие данных так эффективно для их хранения, то почему оно не применяется для всех данных? Дело в том, что существует компромисс между временем и пространством. На то, чтобы сжать часть данных и распаковать их обратно в исходную форму, требуется время. Поэтому сжатие данных имеет смысл только в ситуациях, когда небольшой размер важнее, чем быстрое выполнение. Возьмем, к примеру, большие файлы, передаваемые через Интернет. Их сжатие имеет смысл, поскольку для передачи файлов потребуется больше времени, чем для их распаковки после получения. Кроме того, время, необходимое для сжатия файлов при их хранении на исходном сервере, необходимо учитывать только один раз.

Сжать данные самыми простыми способами можно тогда, когда вы понимаете, что типы хранилищ данных используют больше битов, чем необходимо для их содержимого. Например, на низком уровне, если целые числа без знака, значения ко-

торых никогда не превысят 65 535, сохраняются в памяти как 64-разрядные целые числа без знака, то они сохраняются неэффективно. Вместо этого их можно хранить как 16-разрядные целые числа без знака. Это уменьшит потребление пространства для фактического хранения чисел на 75 % (16 бит вместо 64). Таким неэффективным способом хранятся миллионы чисел, это может означать до мегабайта впустую потраченного пространства.

В Python из соображений простоты (что, конечно же, является законной целью) разработчик иногда избавлен от размышлений. Не существует типов 64-разрядных целых чисел без знака и 16-разрядных целых чисел без знака. Есть только один тип `int`, который позволяет хранить числа произвольной точности. Узнать, сколько байтов памяти потребляют ваши объекты Python, поможет функция `sys.getsizeof()`. Но из-за вычислительных издержек, свойственных объектной системе Python, в Python 3.7 нет способа создать тип `int`, который занимал бы менее 28 байт (224 бита). Для хранения одного числа типа `int` можно добавлять по одному биту за раз (как мы и сделаем в этом примере), но оно все равно занимает как минимум 28 байтов.

ПРИМЕЧАНИЕ

> Если вы давно не имели дела с двоичными числами, напомню, что бит — это одно значение, равное 1 или 0. Последовательность нулей и единиц позволяет представить число в системе счисления по основанию 2. Для целей этого раздела вам не нужно выполнять какие-либо математические операции в двоичной системе счисления, но вы должны понимать, что число битов, которые способен хранить тип, определяет то, сколько разных значений в нем можно представлять. Например, 1 бит может представлять два значения (0 или 1), 2 бита могут представлять четыре значения (00, 01, 10, 11), 3 бита — восемь значений и т. д.

Если число различных значений, которые способен представлять тип, меньше, чем число значений, которые могут представлять биты, используемые для его хранения, то такой тип может быть сохранен более эффективно. Рассмотрим, к примеру, нуклеотиды, которые образуют ген в ДНК[1]. Каждый нуклеотид может принимать только одно из четырех значений: A, C, G или T. (Подробнее об этом будет рассказано в главе 2.) Однако, если ген хранится как тип `str`, что можно рассматривать как коллекцию символов Unicode, каждый нуклеотид будет представлен символом, для хранения которого обычно требуется 8 бит. В двоичном коде для хранения типа с четырьмя возможными значениями требуется всего 2 бита. Значения 00, 01, 10 и 11 — это четыре различных значения, которые могут быть представлены 2 битами. Если A соответствует 00, C — 01, G — 10, а T — 11, то объем

[1] Этот пример взят из книги: *Седжвик Р., Уэйн К.* Алгоритмы на Java. 4-е изд. — М.: Вильямс, 2013 (*Sedgewick R., Wayne K.* Algorithms. 4th ed. — Addison-Wesley Professional, 2011).

хранилища, необходимого для строки нуклеотидов, может быть уменьшен на 75 % (с 8 до 2 бит на каждый нуклеотид).

Вместо того чтобы хранить нуклеотиды как тип `str`, их можно сохранить как строку битов (рис. 1.5). *Строка битов* — это именно то, чем она кажется: последовательность нулей и единиц произвольной длины. К сожалению, стандартная библиотека Python не содержит готовых конструкций для работы с битовыми строками произвольной длины.

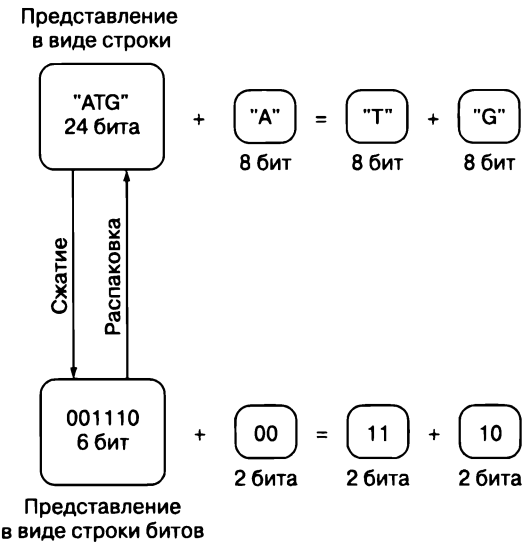


Рис. 1.5. Сжатие типа str, представляющего ген, в строку битов, где выделяется по 2 бита на нуклеотид

Следующий код (листинг 1.10) преобразует строку, состоящую из букв A, C, G и T, в строку битов и обратно. Строка битов хранится в виде числа `int`. Поскольку тип `int` в Python может быть произвольной длины, его можно использовать как строку битов произвольной длины. Чтобы преобразовать ее обратно в `str`, реализуем на Python специальный метод `__str__()`.

Листинг 1.10. trivial_compression.py

```
class CompressedGene:
    def __init__(self, gene: str) -> None:
        self._compress(gene)
```

В классе `CompressedGene` предоставляется строка символов типа `str`, описывающих нуклеотиды в гене. Внутри класса хранится последовательность нуклеотидов в виде строки битов. Основное назначение метода `__init__()` состоит в инициализации конструкции строки битов соответствующими данными. Метод `__init__()`

вызывает `_compress()`, который выполняет всю грязную работу по фактическому преобразованию предоставленной строки нуклеотидов `str` в строку битов.

Обратите внимание на то, что имя `_compress()` начинается с подчеркивания. В Python нет концепции настоящих закрытых методов или переменных. (Все переменные и методы могут быть доступны через рефлексию, строгого соблюдения приватности здесь нет.) По соглашению знак подчеркивания в начале имени указывает на то, что акторы за пределами класса не должны полагаться на реализацию этого метода. (Впоследствии это может быть изменено, и такие методы должны рассматриваться как приватные.)

СОВЕТ

Если имя метода или переменной в классе начинается с двух символов подчеркивания, то Python будет искажать это имя, изменяя его имя реализации с помощью «соли», что не позволит другим классам легко его обнаруживать. В этой книге используется одинарное подчеркивание для обозначения приватных переменных и методов, но вы можете применять двойное, если действительно хотите подчеркнуть, что элемент является приватным. Подробнее об именовании в Python читайте в разделе «Описательные стили именования» в PEP 8: http://mng.bz/NA52.

Теперь посмотрим, как на самом деле можно выполнить сжатие (листинг 1.11).

Листинг 1.11. trivial_compression.py (продолжение)

```
def _compress(self, gene: str) -> None:
    self.bit_string: int = 1 # начальная метка
    for nucleotide in gene.upper():
        self.bit_string <<= 2 # сдвиг влево на 2 бита
        if nucleotide == "A": # поменять 2 последних бита на 00
            self.bit_string |= 0b00
        elif nucleotide == "C": # поменять 2 последних бита на 01
            self.bit_string |= 0b01
        elif nucleotide == "G": # поменять 2 последних бита на 10
            self.bit_string |= 0b10
        elif nucleotide == "T": # поменять 2 последних бита на 11
            self.bit_string |= 0b11
        else:
            raise ValueError("Invalid Nucleotide:{}".format(nucleotide))
```

Метод `_compress()` последовательно просматривает каждый символ в строке нуклеотидов типа `str`. Встретив символ A, он добавляет в строку битов 00, встретив C — 01 и т. д. Помните, что каждый нуклеотид занимает 2 бита. Поэтому перед добавлением очередного нуклеотида мы сдвигаем битовую строку на 2 бита влево (`self.bit_string <<= 2`).

Каждый нуклеотид добавляется с помощью операции ИЛИ (`|`). После сдвига влево с правой стороны строки битов добавляются два нуля. Если в побитовых

операциях ИЛИ, таких как `self.bit_string | = 0b10`, операндами являются ноль и любое другое значение, то это значение заменяет 0. Другими словами, мы постоянно добавляем 2 новых бита в правую часть строки битов. Два добавляемых бита определяются типом нуклеотида.

Наконец, мы реализуем декомпрессию и специальный метод `__str__()`, который ее использует (листинг 1.12).

Листинг 1.12. trivial_compression.py (продолжение)

```
def decompress(self) -> str:
    gene: str = ""
    for i in range(0, self.bit_string.bit_length() - 1, 2):
    # -1 чтобы исключить метку
        bits: int = self.bit_string >> i & 0b11
        # получить только 2 значимых бита
        if bits == 0b00: # A
            gene += "A"
        elif bits == 0b01: # C
            gene += "C"
        elif bits == 0b10: # G
            gene += "G"
        elif bits == 0b11: # T
            gene += "T"
        else:
            raise ValueError("Invalid bits:{}".format(bits))
    return gene[::-1] # [::-1] обращение строки посредством обратных срезов

def __str__(self) -> str: # представление строки в виде красивого вывода
    return self.decompress()
```

Функция `decompress()` считывает из строки битов по 2 бита и использует их, чтобы определить, какой символ добавить в конец представления гена типа `str`. Поскольку биты считываются в обратном направлении, по сравнению с последовательностью, в которой они были сжаты (справа налево, а не слева направо), то представление `str` в итоге обращается (с помощью нотации срезов для обращения `[::-1]`). Наконец, обратите внимание на то, как удобный метод `bit_length()` типа `int` помог нам при разработке `decompress()`. Проверим, как это работает (листинг 1.13).

Листинг 1.13. trivial_compression.py (продолжение)

```
if __name__ == "__main__":
    from sys import getsizeof
    original: str =

"TAGGGATTAACCGTTATATATATATAGCCATGGATCGATTATATAGGGATTAACCGTTATATATATATAGC
CATGGATCGATTATA" * 100
    print("original is {} bytes".format(getsizeof(original)))
    compressed: CompressedGene = CompressedGene(original) # сжатие
```

```
print("compressed is {} bytes".format(getsizeof(compressed.bit_string)))
print(compressed) # распаковка
print("original and decompressed are the same: {}".format(original ==
    compressed.decompress()))
```

Используя метод `sys.getsizeof()`, можно указать при выводе, действительно ли мы сэкономили почти 75 % места в памяти при сохранении гена с помощью этой схемы сжатия (листинг 1.14).

Листинг 1.14. trivial_compression.py (вывод)

```
original is 8649 bytes
compressed is 2320 bytes
TAGGGATTAACC...
original and decompressed are the same: True
```

ПРИМЕЧАНИЕ

В классе CompressedGene мы широко задействовали операторы if для выбора разных вариантов методов сжатия и распаковки. Это довольно типично для Python, поскольку в нем нет оператора switch. Вы также увидите, что для выбора из множества вариантов Python часто основательно опирается на словари вместо интенсивного использования операторов if. Представьте себе, например, словарь, в котором мы могли бы находить соответствующую комбинацию битов для каждого нуклеотида. Иногда это выглядит более читабельным, но может снизить производительность. Несмотря на то что с технической точки зрения сложность поиска по словарю равна O(1), затраты на выполнение функции хеширования иногда приводят к тому, что словарь оказывается менее производительным, чем серия операторов if. Так ли это в конкретном случае, зависит от того, какие операторы if определенной программы должны выполняться для принятия решения. Возможно, имеет смысл выполнить тесты производительности для обоих методов в случае, если вам нужно сделать выбор между операторами if и поиском по словарю в критически важном месте кода.

1.3. Невскрываемое шифрование

Одноразовый шифр — это способ шифрования фрагмента данных путем его комбинации с бессмысленными случайными фиктивными данными таким образом, что оригинал не может быть восстановлен без доступа как к результату шифрования, так и к фиктивным данным. По сути, это шифратор с парой ключей. Один ключ — это результат шифрования, а второй — случайные фиктивные данные. Сам по себе каждый из ключей бесполезен, только их комбинация позволяет разблокировать исходные данные. При правильном выполнении одноразовый шифр представляет собой форму невскрываемого (unbreakable) шифрования. Процесс шифрования показан на рис. 1.6.

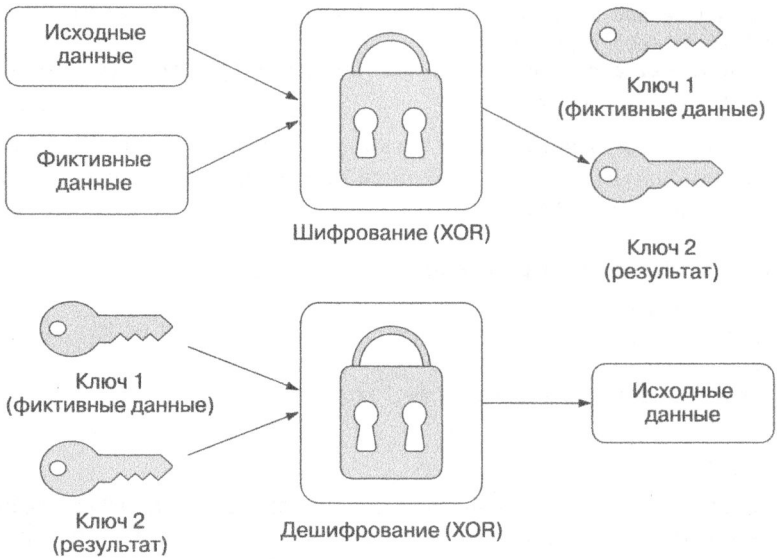


Рис. 1.6. Одноразовый шифр приводит к созданию двух ключей, которые можно разделить, а затем объединить для восстановления исходных данных

1.3.1. Получение данных в заданной последовательности

В этом примере мы зашифруем строку `str`, используя одноразовый шифр. Один из способов представления типа `str` в Python 3 — это последовательность байтов UTF-8 (UTF-8 — это кодировка символов Unicode). Тип `str` может быть преобразован в последовательность байтов UTF-8, представленную как тип `bytes`, с помощью метода `encode()`. Аналогично последовательность байтов UTF-8 может быть преобразована обратно в `str` с помощью метода `decode()` для типа `bytes`.

Существует три критерия, которым должны соответствовать фиктивные данные, используемые в операции одноразового шифрования, чтобы полученный результат невозможно было взломать. Фиктивные данные должны быть той же длины, что и исходные, быть действительно случайными и полностью секретными. Первый и третий критерии диктует здравый смысл. Если фиктивные данные повторяются, потому что слишком короткие, может наблюдаться закономерность. Если один из ключей не является действительно секретным (например, повторно применяется в другом месте или частично раскрыт), то у злоумышленника будет подсказка. Второй же критерий ставит вопрос: можем ли мы генерировать действительно случайные данные? Для большинства компьютеров ответ — нет.

В этом примере задействуем функцию генерации псевдослучайных данных `token_bytes()` из модуля `secrets` (впервые включен в стандартную библиотеку

в Python 3.6). Наши данные не будут действительно случайными в том смысле, что внутри пакета `secrets` все еще применяется генератор псевдослучайных чисел, но для наших целей он будет достаточно близок к случайному. Сгенерируем случайный ключ для использования в качестве фиктивных данных (листинг 1.15).

Листинг 1.15. unbreakable_encryption.py

```
from secrets import token_bytes
from typing import Tuple

def random_key(length: int) -> int:
    # генерировать length случайных байтов
    tb: bytes = token_bytes(length)
    # преобразовать эти байты в битовую строку и вернуть ее
    return int.from_bytes(tb, "big")
```

Эта функция создает `int` длиной `length`, заполненный случайными байтами. Для преобразования байтов в `int` используется метод `int.from_bytes()`. Как преобразовать несколько байтов в одно целое число? Ответ находится в разделе 1.2. Прочитав его, вы узнали, что тип `int` может иметь произвольный размер, и увидели, как его можно задействовать в качестве универсальной строки битов. Здесь тип `int` используется таким же образом. Например, метод `from_bytes()` принимает 7 байт (7 байт · 8 бит = 56 бит) и преобразует их в 56-битное целое число. Почему это полезно? Побитовые операции могут выполняться легче и производительнее для одного `int` (читается как «длинная строка битов»), чем для множества отдельных байтов, представленных в виде последовательности. И мы собираемся применить побитовую операцию XOR.

1.3.2. Шифрование и дешифрование

Как объединить фиктивные данные с исходными данными, которые мы хотим зашифровать? В этом поможет операция *XOR* — логическая побитовая (работает на уровне битов) операция, которая возвращает `true`, если один из ее операндов равен `True`, и `False`, если оба оператора равны `true` или ни один из них не равен `true`. Как вы уже догадались, аббревиатура XOR означает exclusive or (исключающее ИЛИ).

В Python оператор XOR обозначается как `^`. В контексте битов двоичных чисел XOR возвращает 1 для `0 ^ 1` и `1 ^ 0`, но 0 для `0 ^ 0` и `1 ^ 1`. Если объединить два числа, выполнив для каждого их бита XOR, то полезным свойством этой операции будет то, что при объединении результата с одним из операндов получим второй операнд:

```
A ^ B = C
C ^ B = A
C ^ A = B
```

Такое представление ключа лежит в основе шифрования с использованием одноразового шифра. Чтобы сформировать результат, мы просто выполняем XOR для значения `int`, представляющего байты исходной строки `str`, и случайно сгенерированного значения `int` той же битовой длины, полученного с помощью `random_key()`. Полученная пара ключей будет фиктивными данными и результатом шифрования (листинг 1.16).

Листинг 1.16. unbreakable_encryption.py (продолжение)

```
def encrypt(original: str) -> Tuple[int, int]:
    original_bytes: bytes = original.encode()
    dummy: int = random_key(len(original_bytes))
    original_key: int = int.from_bytes(original_bytes, "big")
    encrypted: int = original_key ^ dummy # XOR
    return dummy, encrypted
```

ПРИМЕЧАНИЕ

Функция int.from_bytes() получает два аргумента. Первый, bytes — это строка, которую нужно преобразовать в int. Второй — порядок следования этих байтов (big). Порядок следования означает последовательность байтов, используемую для хранения данных. Какой байт идет первым, старший или младший? В нашем случае, если применять один и тот же порядок при шифровании и дешифровке, это не имеет значения, потому что мы фактически манипулируем данными только на уровне отдельных битов. В других ситуациях, когда вы не контролируете обе стороны процесса шифрования, порядок может иметь значение, поэтому будьте осторожны!

Расшифровка — это просто рекомбинация пары ключей, сгенерированной с помощью `encrypt()`. Она достигается посредством повторного выполнения операции XOR побитово для двух ключей. Окончательный результат должен быть преобразован обратно в тип `str`. Для этого сначала `int` преобразуется в `bytes` с помощью `int.to_bytes()`. Данный метод требует, чтобы число байтов было преобразовано из `int`. Чтобы получить это число, нужно разделить длину в битах на 8 (количество битов в байте). А затем метод `bytes decode()` возвращает `str` (листинг 1.17).

Листинг 1.17. unbreakable_encryption.py (продолжение)

```
def decrypt(key1: int, key2: int) -> str:
    decrypted: int = key1 ^ key2 # XOR
    temp: bytes = decrypted.to_bytes((decrypted.bit_length()+ 7) // 8, "big")
    return temp.decode()
```

Здесь также необходимо прибавить 7 к длине расшифрованных данных, прежде чем использовать целочисленное деление (`//`) на 8, чтобы гарантировать округление и избежать ошибки смещения на единицу. Если наше шифрование с одноразовыми ключами работает, то мы должны шифровать и дешифровать одну и ту же строку Unicode без проблем (листинг 1.18).

Листинг 1.18. unbreakable_encryption.py (продолжение)

```
if __name__ == "__main__":
    key1, key2 = encrypt("One Time Pad!")
    result: str = decrypt(key1, key2)
    print(result)
```

Если в консоль выводится фраза "One Time Pad!", значит, все заработало.

1.4. Вычисление числа π

Число π, равное 3,14159... имеет большое значение в математике и может быть получено с использованием множества формул. Одна из самых простых — формула Лейбница. Согласно этой формуле следующий бесконечный ряд сходится к числу π:

```
π = 4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11...
```

Вы могли заметить, что числитель в этом бесконечном ряду всегда равен 4, знаменатель увеличивается на 2, а операциями над элементами ряда являются по очереди сложение и вычитание.

Мы можем моделировать ряды простым способом, переводя части формулы в переменные внутри функции. Числитель может быть константой 4, а знаменатель — переменной, которая начинается с 1 и каждый раз увеличивается на 2. Операция может быть представлена как −1 или 1 в зависимости от того, является она сложением или вычитанием. Наконец, переменная `pi` в листинге 1.19 используется для накопления суммы элементов ряда по мере выполнения цикла `for`.

Листинг 1.19. calculating_pi.py

```
def calculate_pi(n_terms: int) -> float:
    numerator: float = 4.0
    denominator: float = 1.0
    operation: float = 1.0
    pi: float = 0.0
    for _ in range(n_terms):
        pi += operation * (numerator / denominator)
        denominator += 2.0
        operation *= -1.0
    return pi

if __name__ == "__main__":
    print(calculate_pi(1000000))
```

СОВЕТ

На большинстве платформ значения float в Python являются 64-битными числами с плавающей точкой (то, что в C соответствует типу double).

Эта функция является примером того, как быстрое преобразование между формулой и программным кодом может быть простым и эффективным при моделировании или имитации интересной концепции. Механическое преобразование — полезный инструмент, но необходимо помнить, что это не всегда самое эффективное решение. Безусловно, формула Лейбница для числа π может быть реализована посредством более эффективного или компактного кода.

ПРИМЕЧАНИЕ

Чем больше элементов бесконечного ряда вычисляется (чем больше значение n_terms при вызове метода calc_pi()), тем точнее будет окончательное значение числа π.

1.5. Ханойские башни

Есть три высоких вертикальных столбика (здесь и далее — башни). Мы будем их обозначать A, B и C. Диски с отверстиями в центре нанизаны на башню A. Самый широкий диск — будем называть его диском 1 — находится внизу. Остальные диски, расположенные над ним, обозначены возрастающими цифрами и постепенно сужаются кверху. Например, если бы у нас было три диска, то самый широкий из них, тот, что снизу, имел бы номер 1. Следующий по ширине диск, под номером 2, располагался бы над диском 1. Наконец, самый узкий диск, под номером 3, лежал бы на диске 2.

Наша цель — переместить все диски с башни A на башню C, учитывая следующие ограничения.

- Диски можно перемещать только по одному.
- Единственный доступный для перемещения диск — тот, что расположен наверху любой башни.
- Более широкий диск никогда не может располагаться поверх более узкого.

Схематически задача показана на рис. 1.7.

1.5.1. Моделирование башен

Стек — это структура данных, смоделированная по принципу «последним пришел — первым вышел» (last-in-first-out, LIFO). То, что попадает в стек последним, становится первым, что оттуда извлекается. Две основные операции стека — это `push` (поместить) и `pop` (извлечь). Операция `push` помещает новый элемент в стек, а `pop` удаляет из стека и возвращает последний вставленный элемент. Можно легко смоделировать стек в Python, используя список в качестве резервного хранилища (листинг 1.20).

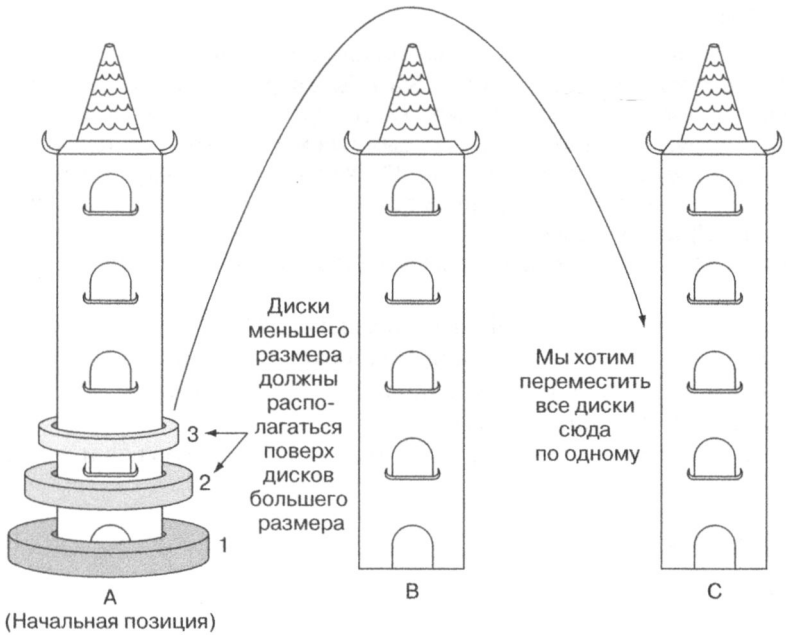


Рис. 1.7. Задача состоит в том, чтобы переместить три диска по одному с башни A на башню C. Диск большего размера никогда не может располагаться поверх диска меньшего размера

Листинг 1.20. hanoi.py

```
from typing import TypeVar, Generic, List
T = TypeVar('T')

class Stack(Generic[T]):

    def __init__(self) -> None:
        self._container: List[T] = []

    def push(self, item: T) -> None:
        self._container.append(item)

    def pop(self) -> T:
        return self._container.pop()

    def __repr__(self) -> str:
        return repr(self._container)
```

ПРИМЕЧАНИЕ

В представленном классе Stack реализован метод __repr__(), позволяющий легко исследовать содержимое башни. __repr__() — это то, что будет выводиться при применении к стеку функции print().

ПРИМЕЧАНИЕ

Как говорилось во введении, в книге используются аннотации типов. Импорт Generic из модуля ввода позволяет Stack быть параметрическим классом для определенного типа в аннотациях типов. Произвольный тип T определен в T = TypeVar ('T'). T может быть любым типом. Когда впоследствии аннотация типа будет задействоваться для Stack при решении задачи о ханойских башнях, в подсказке будет стоять Stack[int], то есть вместо T будет применен тип int. Другими словами, здесь стек — это стек целых чисел. Если вы испытываете затруднения с аннотациями типов, загляните в приложение B.

Стеки идеально подходят для задачи о ханойских башнях. Для того чтобы переместить диск на башню, мы можем использовать операцию `push`. Для того чтобы переместить диск с одной башни на другую, мы можем вытолкнуть его с первой (`pop`) и поместить на вторую (`push`).

Определим башни как объекты `Stack` и заполним первую из них дисками (листинг 1.21).

Листинг 1.21. hanoi.py (продолжение)

```
num_discs: int = 3
tower_a: Stack[int] = Stack()
tower_b: Stack[int] = Stack()
tower_c: Stack[int] = Stack()
for i in range(1, num_discs + 1):
    tower_a.push(i)
```

1.5.2. Решение задачи о ханойских башнях

Как можно решить задачу о ханойских башнях? Предположим, что мы пытаемся переместить только один диск. Тогда мы бы знали, как это сделать, верно? Фактически перемещение одного диска — базовый случай для рекурсивного решения данной задачи. Перемещение нескольких дисков является рекурсивным случаем. Ключевой момент в том, что у нас, по сути, есть два сценария, которые необходимо закодировать: перемещение одного диска (базовый случай) и перемещение нескольких дисков (рекурсивный случай).

Чтобы понять рекурсивный случай, рассмотрим конкретный пример. Предположим, у нас есть три диска — верхний, средний и нижний, расположенные на башне A, и мы хотим переместить их на башню C. (Впоследствии это поможет схематически описать задачу.) Сначала мы могли бы переместить верхний диск на башню C. Затем — переместить средний диск на башню B, а после этого — верхний диск с башни C на башню B. Теперь у нас есть нижний диск, все еще расположенный на башне A, и два верхних диска на башне B. По существу, мы уже успешно переместили два диска с одной башни (A) на другую (B). Перемещение нижнего диска с A на C является базовым случаем (перемещение одного диска). Теперь можем переместить два верхних диска с B на C посредством той же про-

цедуры, что и с A на B. Перемещаем верхний диск на A, средний — на C и, наконец, верхний — с A на C.

СОВЕТ

> В учебных классах информатики нередко встречаются маленькие модели этих башен, построенные из штырей и пластиковых дисков. Вы можете изготовить собственную модель с помощью трех карандашей и трех листов бумаги. Возможно, это поможет вам наглядно представить решение.

В примере с тремя дисками был простой базовый случай перемещения одного диска и рекурсивный случай перемещения остальных дисков (в данном случае двух) с применением временной третьей башни. Мы можем разбить рекурсивный случай на следующие этапы.

1. Переместить верхние $n - 1$ дисков с башни A на башню B (временная), используя C в качестве промежуточной башни.
2. Переместить нижний диск с A на C.
3. Переместить $n - 1$ дисков с башни B на башню C, башня A — промежуточная.

Удивительно, но этот рекурсивный алгоритм работает не только для трех, а для любого количества дисков. Закодируем его как функцию `hanoi()`, которая отвечает за перемещение дисков с одной башни на другую, используя третью временную башню (листинг 1.22).

Листинг 1.22. hanoi.py (продолжение)

```
def hanoi(begin: Stack[int], end: Stack[int], temp: Stack[int], n: int) ->
     None:
    if n == 1:
        end.push(begin.pop())
    else:
        hanoi(begin, temp, end, n - 1)
        hanoi(begin, end, temp, 1)
        hanoi(temp, end, begin, n - 1)
```

После вызова `hanoi()` нужно проверить башни A, B и C, чтобы убедиться, что диски были успешно перемещены (листинг 1.23).

Листинг 1.23. hanoi.py (продолжение)

```
if __name__ == "__main__":
    hanoi(tower_a, tower_c, tower_b, num_discs)
    print(tower_a)
    print(tower_b)
    print(tower_c)
```

Вы обнаружите, что диски действительно были перемещены. При кодировании решения задачи о ханойских башнях не обязательно понимать каждый

шаг, необходимый для перемещения нескольких дисков с башни A на башню C. Мы пришли к пониманию общего рекурсивного алгоритма перемещения любого количества дисков и систематизировали его, позволяя компьютеру сделать все остальное. В этом и заключается сила формулирования рекурсивных решений задач: мы зачастую можем представлять себе решения абстрактно, не тратя силы на мысленное представление каждого отдельного действия.

Кстати, функция `hanoi()` будет выполняться экспоненциальное число раз в зависимости от количества дисков, что делает решение задачи даже для 64 дисков непригодным. Вы можете попробовать выполнить ее с другим числом дисков, изменяя переменную `num_discs`. По мере увеличения количества дисков число шагов выполнения задачи о ханойских башнях растет экспоненциально, подробнее об этом можно прочитать во множестве источников. Если интересно больше узнать о математике, стоящей за рекурсивным решением этой задачи, — см. разъяснение Карла Берча в статье «О ханойских башнях», http://mng.bz/c1i2.24.

1.6. Реальные приложения

Различные методы, представленные в этой главе (рекурсия, мемоизация, сжатие и манипулирование на битовом уровне), настолько широко распространены в разработке современного программного обеспечения, что без них невозможно представить мир вычислений. Несмотря на то что задачи могут быть решены и без них, зачастую более логично или целесообразно решать их с использованием этих методов.

В частности, рекурсия лежит в основе не только многих алгоритмов, но даже целых языков программирования. В некоторых функциональных языках программирования, таких как Scheme и Haskell, рекурсия заменяет циклы, применяемые в императивных языках. Однако следует помнить, что все, что может быть достигнуто с помощью рекурсивного метода, может быть выполнено также итерационным способом.

Мемоизация была успешно использована для ускорения работы синтаксических анализаторов — программ, которые интерпретируют языки. Это полезно во всех задачах, где результат недавнего вычисления, скорее всего, будет запрошен снова. Еще одна область действия мемоизации — среды выполнения языков программирования. Некоторые такие среды выполнения, например для версии Prolog, автоматически сохраняют результаты вызовов функций *(автомемоизация)*, поэтому функцию не приходится выполнять в следующий раз при таком же вызове. Это похоже на работу декоратора `@lru_cache()` в `fib6()`.

Сжатие сделало мир, подключенный к Интернету с его ограниченной пропускной способностью, более терпимым. Метод битовых строк, рассмотренный в разделе 1.2, применим для простых типов данных в реальном мире, имеющих ограниченное число возможных значений, для которых даже 1 байт является избыточным. Однако большинство алгоритмов сжатия работают путем поиска шаблонов

или структур в наборе данных, которые позволяют устранить повторяющуюся информацию. Они значительно сложнее, чем описано в разделе 1.2.

Одноразовые шифры не подходят для общих случаев шифрования. Они требуют, чтобы и шифратор, и дешифратор имели один из ключей (фиктивные данные в нашем примере) для восстановления исходных данных, что слишком громоздко и в большинстве схем шифрования не позволяет достичь цели — сохранить ключи в секрете. Но, возможно, вам будет интересно узнать, что название «одноразовый шифр» придумали шпионы, которые во время холодной войны использовали настоящие бумажные блокноты с записанными в них фиктивными данными для создания шифрованных сообщений.

Эти методы являются строительными блоками программ, на них основаны другие алгоритмы. В следующих главах вы увидите, как широко они применяются.

1.7. Упражнения

1. Напишите еще одну функцию, которая вычисляет *n*-й элемент последовательности Фибоначчи, используя метод вашей собственной разработки. Напишите модульные тесты, которые оценивали бы правильность этой функции и ее производительность, по сравнению с другими версиями, представленными в этой главе.
2. Вы видели, как можно применять простой тип `int` в Python для представления строки битов. Напишите эргономичную обертку для `int`, которую можно было бы использовать как последовательность битов (сделайте ее итеративной и реализуйте `__getitem__()`). Переопределите `CompressedGene` с помощью этой обертки.
3. Напишите программу решения задачи о ханойских башнях, которая работала бы для любого количества башен.
4. Задействуйте одноразовый шифр для шифрования и дешифровки изображений.

2 Задачи поиска

Поиск — это такой широкий термин, что всю книгу можно было бы назвать «Классические задачи поиска на Python». Эта глава посвящена основным алгоритмам поиска, которые должен знать каждый программист. Несмотря на декларативное название, она не претендует на полноту.

2.1. Поиск ДНК

В компьютерных программах гены обычно представляются в виде последовательности символов A, C, G и T, где каждая буква означает *нуклеотид*, а комбинация трех нуклеотидов называется *кодоном* (рис. 2.1). Кодон кодирует конкретную аминокислоту, которая вместе с другими аминокислотами может образовывать белок. Классическая задача в программах биоинформатики — найти в гене определенный кодон.

2.1.1. Хранение ДНК

Мы можем представить нуклеотид в виде простого `IntEnum` с четырьмя вариантами значений (листинг 2.1).

Листинг 2.1. dna_search.py

```
from enum import IntEnum
from typing import Tuple, List

Nucleotide: IntEnum = IntEnum('Nucleotide', ('A', 'C', 'G', 'T'))
```

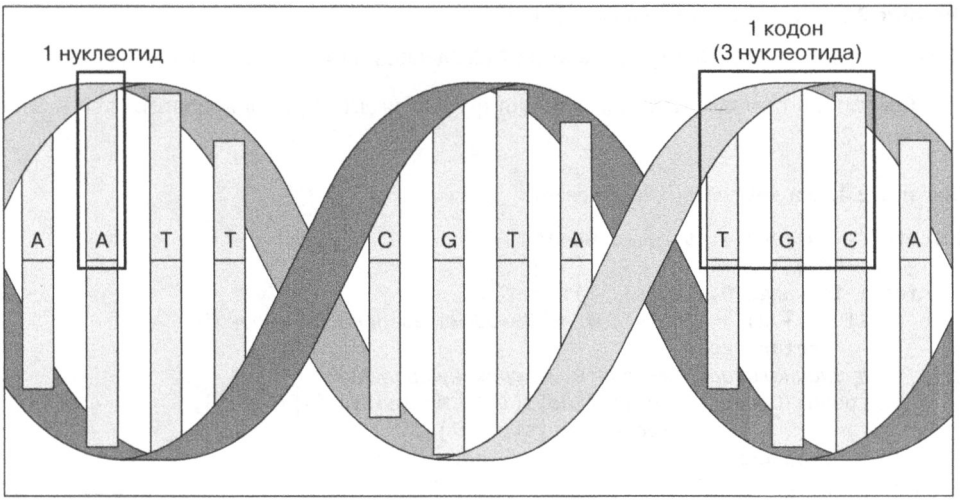


Рис. 2.1. Нуклеотид представлен буквой A, C, G или T. Кодон состоит из трех нуклеотидов, а ген — из нескольких кодонов

`Nucleotide` относится к типу `IntEnum`, а не просто `Enum`, поскольку `IntEnum` «бесплатно» обеспечивает операторы сравнения (<, >= и т. п.). Наличие этих операторов в типе данных необходимо для алгоритмов поиска, которые мы собираемся реализовать, чтобы иметь возможность сравнивать значения. Импорт `Tuple` и `List` из пакета `typing` нужен для того, чтобы помочь с аннотациями типа.

Кодоны могут быть определены как кортеж из трех нуклеотидов, а ген — как список кодонов (листинг 2.2).

Листинг 2.2. dna_search.py (продолжение)

```
Codon = Tuple[Nucleotide, Nucleotide, Nucleotide]
# псевдоним типа для кодонов
Gene = List[Codon] # псевдоним типа для генов
```

ПРИМЕЧАНИЕ

Позже нам придется сравнивать объекты Codon друг с другом, но не нужно определять собственный класс с оператором <, явно реализованным для Codon. Это связано с тем, что в Python есть встроенная поддержка сравнения кортежей, состоящих из типов, которые также являются сравнимыми.

Как правило, в Интернете гены представлены в формате файлов, которые содержат одну гигантскую строку, содержащую все нуклеотиды в той последовательности, в которой они располагаются в гене. Мы определим такую строку для предполагаемого гена и назовем ее `gene_str` (листинг 2.3).

Листинг 2.3. dna_search.py (продолжение)

```
gene_str: str = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT"
```

Нам также понадобится служебная функция для преобразования `str` в `Gene` (листинг 2.4).

Листинг 2.4. dna_search.py (продолжение)

```
def string_to_gene(s: str) -> Gene:
    gene: Gene = []
    for i in range(0, len(s), 3):
        if (i + 2) >= len(s): # не выходить за пределы строки!
            return gene
        # инициализировать кодон из трех нуклеотидов
        codon: Codon = (Nucleotide[s[i]], Nucleotide[s[i + 1]],
                        Nucleotide[s[i + 2]])
        gene.append(codon) # добавить кодон в ген
    return gene
```

Функция `string_to_gene()` постоянно проходит строку и преобразует каждые три символа в объекты `Codon`, которые добавляет в конец нового объекта `Gene`. Если эта функция обнаруживает отсутствие двух нуклеотидов после текущего места в исследуемой строке `s` (см. оператор `if` в цикле), значит, достигнут конец неполного гена, и эти последние один или два нуклеотида пропускаются.

Функцию `string_to_gene()` можно использовать для преобразования `str` `gene_str` в `Gene` (листинг 2.5).

Листинг 2.5. dna_search.py (продолжение)

```
my_gene: Gene = string_to_gene(gene_str)
```

2.1.2. Линейный поиск

Одна из основных операций, которую мы можем захотеть выполнить с геном, — это поиск определенного кодона. Цель состоит в том, чтобы просто выяснить, существует ли в гене такой кодон.

Линейный поиск перебирает все элементы в пространстве поиска в порядке исходной структуры данных до тех пор, пока не будет найден искомый объект или не достигнут конец структуры данных. По сути, линейный поиск — это самый простой, естественный и очевидный способ поиска чего-либо. В худшем случае линейный поиск потребует проверки каждого элемента в структуре данных, поэтому он имеет сложность $O(n)$, где n — количество элементов в структуре (рис. 2.2).

Определить функцию, которая выполняет линейный поиск, очень легко. Она просто должна перебрать все элементы структуры данных и проверить каждый из них на эквивалентность искомому элементу. В листинге 2.6 такая функция определяется для `Gene` и `Codon`, а затем применяется к `my_gene` и объектам `Codon` с именами `acg` и `gat`.

Рис. 2.2. В худшем случае при линейном поиске нужно последовательно просмотреть каждый элемент массива

Листинг 2.6. dna_search.py (продолжение)

```
def linear_contains(gene: Gene, key_codon: Codon) -> bool:
    for codon in gene:
        if codon == key_codon:
            return True
    return False

acg: Codon = (Nucleotide.A, Nucleotide.C, Nucleotide.G)
gat: Codon = (Nucleotide.G, Nucleotide.A, Nucleotide.T)
print(linear_contains(my_gene, acg)) # True
print(linear_contains(my_gene, gat)) # False
```

ПРИМЕЧАНИЕ

Эта функция имеет демонстрационное назначение. Для всех встроенных типов последовательностей Python (list, tuple, range) реализован метод __contains__(), который позволяет выполнять поиск определенного элемента с помощью оператора in. Фактически оператор in можно использовать для любого типа, для которого реализован метод __contains__(). Например, мы могли бы найти acg в my_gene и вывести результат, написав строку print(acg in my_gene).

2.1.3. Бинарный поиск

Существует более быстрый способ поиска, чем просмотр всех элементов. Но при этом требуется заранее знать кое-что о последовательности структуры данных. Если известно, что структура отсортирована и мы можем мгновенно получить доступ к любому ее элементу по его индексу, то можно выполнить бинарный поиск. Если исходить из этого критерия, то отсортированный список Python — идеальный кандидат для бинарного поиска.

При бинарном поиске средний элемент в отсортированном диапазоне элементов проверяется и сравнивается с искомым элементом. По результатам сравнения диапазон поиска уменьшается наполовину, после чего процесс повторяется. Рассмотрим его на конкретном примере.

Предположим, что у нас есть список отсортированных по алфавиту слов («заяц», «зебра», «кенгуру», «кошка», «крыса», «лама», «собака») и мы ищем слово «крыса».

1. Мы определили, что средним элементом в списке из семи слов является «кошка».
2. Теперь можем определить, что по алфавиту «крыса» идет после «кошки», поэтому она должна находиться (это неточно) в той половине списка, которая следует после «кошки». (Если бы мы нашли «крысу» на этом шаге, то могли бы вернуть местонахождение этого слова; если бы обнаружили, что искомое слово стояло перед средним словом, которое мы проверяли, то могли бы быть уверены, что оно находится в половине списка до «кошки».)
3. Мы могли бы повторить пункты 1 и 2 для той половины списка, в которой, как уже знаем, вероятно, находится слово «крыса». По сути, эта половина становится новым исходным списком. Указанные пункты выполняются повторно до тех пор, пока «крыса» не будет найдена или же диапазон, в котором выполняется поиск, больше не будет содержать элементов для поиска, то есть слова «крыса» не будет существовать в заданном списке слов.

Процедура бинарного поиска проиллюстрирована на рис. 2.3. Обратите внимание на то, что, в отличие от линейного поиска, здесь не подразумевается проверка каждого элемента.

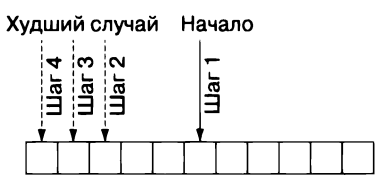


Рис. 2.3. В худшем случае при бинарном поиске придется просматривать только lg(n) элементов списка

При бинарном поиске пространство поиска постоянно сокращается вдвое, поэтому в худшем случае время выполнения поиска составляет $O(\lg n)$. Однако здесь есть своеобразная ловушка. В отличие от линейного поиска для бинарного требуется отсортированная структура данных, а сортировка требует времени. На самом деле сортировка занимает время $O(n \lg n)$ даже при использовании лучших алгоритмов. Если мы собираемся запустить поиск только один раз, а исходная структура данных не отсортирована, возможно, имеет смысл просто выполнить линейный поиск. Но если поиск должен выполняться много раз, то стоит потратить время на сортировку, чтобы каждый отдельный поиск занял значительно меньше времени.

Написание функции бинарного поиска для гена и кодона мало отличается от написания функции для любого другого типа данных, поскольку объекты типа `Codon` можно сравнивать между собой, а тип `Gene` — это просто список (листинг 2.7).

Листинг 2.7. dna_search.py (продолжение)

```
def binary_contains(gene: Gene, key_codon: Codon) -> bool:
    low: int = 0
    high: int = len(gene) - 1
    while low <= high: # пока еще есть место для поиска
        mid: int = (low + high) // 2
        if gene[mid] < key_codon:
            low = mid + 1
        elif gene[mid] > key_codon:
            high = mid - 1
        else:
            return True
    return False
```

Рассмотрим выполнение этой функции построчно:

```
low: int = 0
high: int = len(gene) - 1
```

Начнем поиск с диапазона, который охватывает весь список (ген):

```
while low <= high:
```

Мы продолжаем поиск до тех пор, пока существует диапазон для поиска. Когда `low` станет больше, чем `high`, это будет означать, что в списке не осталось фрагментов для просмотра:

```
mid: int = (low + high) // 2
```

Вычисляем среднее значение `mid`, используя целочисленное деление и простую формулу вычисления среднего, знакомую еще по начальной школе:

```
if gene[mid] < key_codon:
    low = mid + 1
```

Если искомый элемент находится после среднего элемента текущего диапазона, то изменяем диапазон, который будем рассматривать на следующей итерации цикла, перемещая `low` на следующий после текущего среднего элемента. Именно на этом шаге мы вдвое сокращаем диапазон поиска для следующей итерации:

```
elif gene[mid] > key_codon:
    high = mid - 1
```

Если искомый элемент меньше, чем средний, мы точно так же делим диапазон пополам и выбираем другое направление:

```
else:
    return True
```

Если рассматриваемый элемент не меньше и не больше, чем средний, это означает, что мы его нашли! И конечно же, если в цикле закончились итерации, то возвращаем `False` (здесь это повторять не будем), указывая, что элемент не был найден.

Мы можем попробовать запустить функцию с тем же геном и кодоном (листинг 2.8), но необходимо помнить, что сначала нужно отсортировать данные.

Листинг 2.8. dna_search.py (продолжение)

```
my_sorted_gene: Gene = sorted(my_gene)
print(binary_contains(my_sorted_gene, acg)) # True
print(binary_contains(my_sorted_gene, gat)) # False
```

СОВЕТ

Для построения эффективной процедуры бинарного поиска можно воспользоваться модулем bisect из стандартной библиотеки Python: https://docs.python.org/3/library/bisect.html.

2.1.4. Параметризованный пример

Функции `linear_contains()` и `binary_contains()` можно обобщить для работы практически с любой последовательностью Python. Следующие обобщенные версии почти идентичны версиям, которые вы видели ранее, изменились только некоторые имена и аннотации типов.

ПРИМЕЧАНИЕ

В листинге 2.9 задействовано много импортированных типов. В этой главе мы еще раз воспользуемся файлом generic_search.py для многих дополнительных универсальных алгоритмов поиска, что избавит нас от необходимости импорта.

ПРИМЕЧАНИЕ

Прежде чем продолжить работу с книгой, вам нужно установить модуль typing_extensions с помощью команды pip install typing_extensions или pip3 install typing_extensions в зависимости от того, как настроен интерпретатор Python. Этот модуль понадобится для доступа к типу Protocol, который войдет в состав стандартной библиотеки в следующей версии Python (согласно PEP 544). Так что в следующей версии Python импортировать модуль typing_extensions не понадобится и вместо строки from typing_extensions import Protocol можно будет использовать from typing import Protocol.

Листинг 2.9. generic_search.py

```
from __future__ import annotations
from typing import TypeVar, Iterable, Sequence, Generic, List, Callable, Set,
    Deque, Dict, Any, Optional
from typing_extensions import Protocol
```

```
from heapq import heappush, heappop

T = TypeVar('T')

def linear_contains(iterable: Iterable[T], key: T) -> bool:
    for item in iterable:
        if item == key:
            return True
    return False

C = TypeVar("C", bound="Comparable")

class Comparable(Protocol):
    def __eq__(self, other: Any) -> bool:
        ...

    def __lt__(self: C, other: C) -> bool:
        ...

    def __gt__(self: C, other: C) -> bool:
        return (not self < other) and self != other

    def __le__(self: C, other: C) -> bool:
        return self < other or self == other

    def __ge__(self: C, other: C) -> bool:
        return not self < other

    def binary_contains(sequence: Sequence[C], key: C) -> bool:
        low: int = 0
        high: int = len(sequence) - 1
        while low <= high: # пока еще есть место для поиска
            mid: int = (low + high) // 2
            if sequence[mid] < key:
                low = mid + 1
            elif sequence[mid] > key:
                high = mid - 1
            else:
                return True
        return False

if __name__ == "__main__":
    print(linear_contains([1, 5, 15, 15, 15, 15, 20], 5)) # True
    print(binary_contains(["a", "d", "e", "f", "z"], "f")) # True
    print(binary_contains(["john", "mark", "ronald", "sarah"], "sheila"))
    # False
```

Теперь вы можете попробовать выполнить поиск для других типов данных. Эти функции можно использовать практически для любой коллекции Python.

Именно в этом заключается главная эффективность параметризованного кода. Единственный неудачный элемент в этом примере — запутанные аннотации типов Python в форме класса `Comparable`. Тип `Comparable` — это тип, который реализует операторы сравнения (<, >= и т. д.). В будущих версиях Python должен появиться более лаконичный способ создания аннотации типа для типов, которые реализуют эти общие операторы.

2.2. Прохождение лабиринта

Поиск пути через лабиринт аналогичен многим распространенным задачам поиска в информатике. Почему бы буквально не найти путь через лабиринт, чтобы проиллюстрировать алгоритмы поиска в ширину, поиска в глубину и алгоритмы A*?

Наш лабиринт представляет собой двумерную сеть ячеек — объектов `Cell` (листинг 2.10). `Cell` — это перечисление значений `str`, где `" "` означает пустое пространство, а `"X"` — занятое. В иллюстративных целях при выводе лабиринта на печать существуют и другие варианты заполнения ячеек.

Листинг 2.10. maze.py

```
from enum import Enum
from typing import List, NamedTuple, Callable, Optional
import random
from math import sqrt
from generic_search import dfs, bfs, node_to_path, astar, Node

class Cell(str, Enum):
    EMPTY = " "
    BLOCKED = "X"
    START = "S"
    GOAL = "G"
    PATH = "*"
```

Мы снова чересчур много импортируем. Обратите внимание на то, что последний импорт (из `generic_search`) состоит из символов, которые мы еще не определили. Он включен сюда для удобства, но вы можете закомментировать его, пока он не потребуется.

Нам понадобится способ сослаться на конкретную точку в лабиринте. Это будет просто `NamedTuple` со свойствами, представляющими строку и столбец данной ячейки в сетке (листинг 2.11).

Листинг 2.11. maze.py (продолжение)

```
class MazeLocation(NamedTuple):
    row: int
    column: int
```

2.2.1. Создание случайного лабиринта

Класс `Maze` будет сам следить за сеткой (списком списков), описывающей состояние лабиринта. В нем также будут переменные экземпляра, хранящие количество строк и столбцов, начальное и конечное местоположение. Сетка лабиринта будет случайным образом заполнена заблокированными ячейками.

Сгенерированный лабиринт должен быть достаточно разреженным, чтобы почти всегда существовал путь от заданного начального до конечного местоположения. (В конце концов, это нужно для тестирования наших алгоритмов.) Мы позволим функции, вызывающей новый лабиринт, точно задавать степень разреженности, но определим значение по умолчанию, составляющее 20 % заблокированных ячеек. Если случайное число превысит порог, заданный параметром `sparseness`, просто заменим пустое пространство стеной. Если так поступать для каждой возможной точки лабиринта, то статистически разреженность лабиринта в целом будет приближаться к заданному параметру `sparseness` (листинг 2.12).

Листинг 2.12. maze.py (продолжение)

```
class Maze:
    def __init__(self, rows: int = 10, columns: int = 10, sparseness: float =
        0.2, start: MazeLocation = MazeLocation(0, 0), goal: MazeLocation =
        MazeLocation(9, 9)) -> None:
            # инициализация базовых переменных экземпляра
            self._rows: int = rows
            self._columns: int = columns
            self.start: MazeLocation = start
            self.goal: MazeLocation = goal
            # заполнение сетки пустыми ячейками
            self._grid: List[List[Cell]] =
              [[Cell.EMPTY for c in range(columns)]
        for r in range(rows)]
            # заполнение сетки заблокированными ячейками
            self._randomly_fill(rows, columns, sparseness)
            # заполнение начальной и конечной позиций в лабиринте
            self._grid[start.row][start.column] = Cell.START
            self._grid[goal.row][goal.column] = Cell.GOAL

    def _randomly_fill(self, rows: int, columns: int, sparseness: float):
        for row in range(rows):
            for column in range(columns):
                if random.uniform(0, 1.0) < sparseness:
                    self._grid[row][column] = Cell.BLOCKED
```

Теперь, когда у нас есть лабиринт, нужно, чтобы он кратко выводился в консоль. Мы хотим, чтобы элементы лабиринта располагались близко друг к другу и все было похоже на настоящий лабиринт (листинг 2.13).

Листинг 2.13. maze.py (продолжение)

```
# вывести красиво отформатированную версию лабиринта для печати
def __str__(self) -> str:
    output: str = ""
    for row in self._grid:
        output += "".join([c.value for c in row]) + "\n"
    return output
```

Теперь протестируем эти функции лабиринта:

```
maze: Maze = Maze()
print(maze)
```

2.2.2. Мелкие детали лабиринта

Позже нам пригодится функция, которая проверяет, достигли ли мы цели в процессе поиска. Другими словами, мы хотим проверить, является ли определенная `MazeLocation`, которой достиг поиск, нашей целью. Можем добавить в `Maze` следующий метод (листинг 2.14).

Листинг 2.14. maze.py (продолжение)

```
def goal_test(self, ml: MazeLocation) -> bool:
    return ml == self.goal
```

Как передвигаться в лабиринте? Допустим, из заданной ячейки лабиринта мы можем двигаться по горизонтали и вертикали по одной ячейке за ход. Используя эти критерии, функция `successors()` способна находить возможные следующие местоположения из заданной ячейки `Maze`. Однако функция `successors()` будет отличаться для каждой `Maze`, поскольку любой лабиринт имеет собственный размер и набор стен. Поэтому мы определим ее как метод `Maze` (листинг 2.15).

Листинг 2.15. maze.py (продолжение)

```
def successors(self, ml: MazeLocation) -> List[MazeLocation]:
    locations: List[MazeLocation] = []
    if ml.row + 1 < self._rows and self._grid[ml.row + 1][ml.column]
      != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row + 1, ml.column))
    if ml.row - 1 >= 0 and self._grid[ml.row - 1][ml.column] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row - 1, ml.column))
    if ml.column + 1 < self._columns and self._grid[ml.row][ml.column + 1]
      != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column + 1))
    if ml.column - 1 >= 0 and self._grid[ml.row][ml.column - 1]
      != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column - 1))
    return locations
```

Метод `successors()` просто проверяет верхнюю, нижнюю, правую и левую смежные ячейки по отношению к `MazeLocation` в `Maze`, чтобы увидеть, есть ли там пустые места, в которые можно попасть из этой ячейки. Также это позволяет избежать проверки ячеек за пределами лабиринта. Все возможные обнаруженные `MazeLocation` помещаются в список, который в итоге возвращается вызывающей функции.

2.2.3. Поиск в глубину

Поиск в глубину (depth-first search, DFS) — это именно то, чего можно ожидать, судя по названию: поиск, который заходит настолько глубоко, насколько возможно, прежде чем вернуться к последней точке принятия решения в случае, если процесс зайдет в тупик. Мы реализуем параметризованный поиск в глубину, который позволяет решить задачу прохода по лабиринту. Этот поиск можно использовать для решения и других задач. Развитие поиска по лабиринту в глубину проиллюстрировано на рис. 2.4.

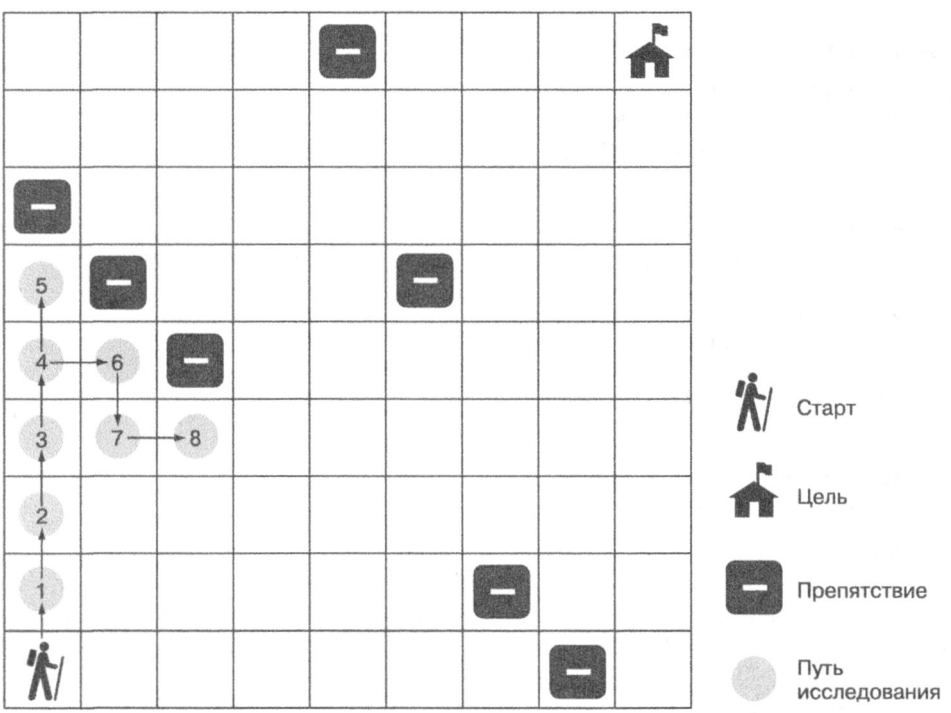


Рис. 2.4. Поиск в глубину проходит по непрерывному пути вглубь, пока не дойдет до препятствия и не будет вынужден вернуться к последней точке принятия решения

Стеки

Алгоритм поиска в глубину опирается на структуру данных, известную как *стек*. (Если вы читали о стеках в главе 1, можете спокойно пропустить этот раздел.) Стек — это структура данных, которая работает по принципу «последним пришел — первым вышел» (last-in-first-out, LIFO). Стек можно представить как стопку документов. Последний документ, помещенный сверху стопки, будет первым, который оттуда извлекут. Обычно стек реализуется на основе более примитивной структуры данных, такой как список. Мы будем реализовывать стек на основе типа `list` Python.

Обычно стек имеет как минимум две операции:

- `push()` — помещает элемент в вершину стека;
- `pop()` — удаляет элемент из вершины стека и возвращает его.

Мы реализуем обе эти операции, а также создадим свойство `empty`, чтобы проверить, остались ли в стеке еще элементы. Разместим код стека в файле `generic_search.py`, с которым уже работали в этой главе (листинг 2.16). Все, что нужно было импортировать, мы уже импортировали.

Листинг 2.16. generic_search.py (продолжение)

```
class Stack(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container # не равно True для пустого контейнера

    def push(self, item: T) -> None:
        self._container.append(item)

    def pop(self) -> T:
        return self._container.pop()

    def __repr__(self) -> str:
        return repr(self._container)
```

Обратите внимание на то, что реализовать стек с использованием типа `list` в Python означает просто всегда добавлять и удалять элементы только с его правого конца. Метод `pop()` для `list` будет реализован неудачно, если в списке не осталось элементов, поэтому `pop()` потерпит неудачу в стеке, если он пуст.

Алгоритм DFS

Понадобится учесть еще один момент, прежде чем можно будет приступить к реализации DFS. Требуется класс `Node`, с помощью которого мы станем отслеживать переход из одного состояния в другое (или из одного места лабиринта в другое)

во время поиска. Node можно представить как обертку вокруг состояния. В случае прохождения лабиринта эти состояния имеют тип MazeLocation. Будем считать Node состоянием, унаследованным от parent. Кроме того, для класса Node мы определим свойства cost и heuristic и реализуем метод __lt__(), чтобы использовать его позже в алгоритме A* (листинг 2.17).

Листинг 2.17. generic_search.py (продолжение)

```
class Node(Generic[T]):
    def __init__(self, state: T, parent: Optional[Node], cost: float = 0.0,
        heuristic: float = 0.0) -> None:
            self.state: T = state
            self.parent: Optional[Node] = parent
            self.cost: float = cost
            self.heuristic: float = heuristic

def __lt__(self, other: Node) -> bool:
    return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

СОВЕТ

Тип Optional указывает, что переменная может ссылаться на значение параметризованного типа или на None.

СОВЕТ

Строка from __future__ import annotations в верхней части файла позволяет объектам Node ссылаться на самих себя в аннотациях типов своих методов. Без нее нам пришлось бы заключить аннотацию типа в кавычки в виде строки (например, 'Node'). В будущих версиях Python импорт annotations будет не нужен. Подробнее об этом читайте в PEP 563 Postponed Evaluation of Annotations, http://mng.bz/pgzR.

В процессе поиска в глубину нужно отслеживать две структуры данных: стек рассматриваемых состояний (мест), который мы назовем frontier, и набор уже просмотренных состояний — explored. До тех пор пока в frontier остаются состояния, DFS будет продолжать проверять, являются ли они целью поиска (найдя искомое состояние, DFS остановит поиск и возвратит его), и добавлять их наследников в frontier. Также алгоритм будет помечать все просмотренные состояния как explored, чтобы поиск не ходил по кругу и уже изученные состояния не становились наследниками. Если frontier окажется пустым, это будет означать, что объектов для поиска не осталось (листинг 2.18).

Листинг 2.18. generic_search.py (продолжение)

```
def dfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
    List[T]]) -> Optional[Node[T]]:
    # frontier – то, что нам нужно проверить
    frontier: Stack[Node[T]] = Stack()
```

```
    frontier.push(Node(initial, None))
    # explored – то, где мы уже были
    explored: Set[T] = {initial}

    # продолжаем, пока есть что просматривать
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
    current_state: T = current_node.state
    # если мы нашли искомое, заканчиваем
    if goal_test(current_state):
        return current_node
    # проверяем, куда можно двинуться дальше и что мы еще не исследовали
    for child in successors(current_state):
        if child in explored: # пропустить состояния, которые уже исследовали
            continue
        explored.add(child)
        frontier.push(Node(child, current_node))
    return None # все проверили, пути к целевой точке не нашли
```

Если `dfs()` завершается успешно, то возвращается `Node`, в котором инкапсулировано искомое состояние. Для того чтобы восстановить путь от начала до целевой ячейки, нужно двигаться в обратном направлении от этого `Node` к его предкам, используя свойство `parent` (листинг 2.19).

Листинг 2.19. generic_search.py (продолжение)

```
def node_to_path(node: Node[T]) -> List[T]:
    path: List[T] = [node.state]
    # двигаемся назад, от конца к началу
    while node.parent is not None:
        node = node.parent
        path.append(node.state)
    path.reverse()
    return path
```

В целях отображения полезно будет разметить лабиринт с указанием успешного пути, начального и конечного положений. Также хорошо иметь возможность удалить путь, чтобы можно было применить разные алгоритмы поиска для одного и того же лабиринта. Для этого в файле `maze.py` нужно добавить в класс `Maze` следующие два метода (листинг 2.20).

Листинг 2.20. maze.py (продолжение)

```
def mark(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.PATH
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL

def clear(self, path: List[MazeLocation]):
```

```
        for maze_location in path:
            self._grid[maze_location.row][maze_location.column] = Cell.EMPTY
        self._grid[self.start.row][self.start.column] = Cell.START
        self._grid[self.goal.row][self.goal.column] = Cell.GOAL
```

Это было долгое путешествие, но оно подходит к концу — мы готовы пройти по лабиринту (листинг 2.21).

Листинг 2.21. maze.py (продолжение)

```
if __name__ == "__main__":
    # Тестирование DFS
    m: Maze = Maze()
    print(m)
    solution1: Optional[Node[MazeLocation]] = dfs(m.start, m.goal_test,
      m.successors)
    if solution1 is None:
        print("No solution found using depth-first search!")
    else:
        path1: List[MazeLocation] = node_to_path(solution1)
        m.mark(path1)
        print(m)
        m.clear(path1)
```

Успешное решение будет выглядеть примерно так:

```
S****X X
 X  *****
       X*
 XX******X
  X*
  X**X
 X *****
       *
    X *X
      *G
```

Звездочки обозначают путь от начальной до конечной точки, который нашла функция поиска в глубину. Помните: поскольку все лабиринты генерируются случайным образом, не для каждого из них существует решение.

2.2.4. Поиск в ширину

Вы могли заметить, что пути прохода по лабиринту, найденные с помощью поиска в глубину, кажутся неестественными. Обычно это не самые короткие пути. Поиск в ширину (breadth-first search, BFS) всегда находит кратчайший путь, систематически просматривая на каждой итерации поиска ближайший по отношению к исходному состоянию слой узлов. Для одних задач поиск в глубину позволяет найти решение быстрее, чем поиск в ширину, а для других — наоборот. Поэтому

выбор алгоритма поиска иногда является компромиссом между возможностью быстрого поиска решения и гарантированной возможностью найти кратчайший путь к цели, если таковой существует. На рис. 2.5 проиллюстрирован поиск пути по лабиринту в ширину.

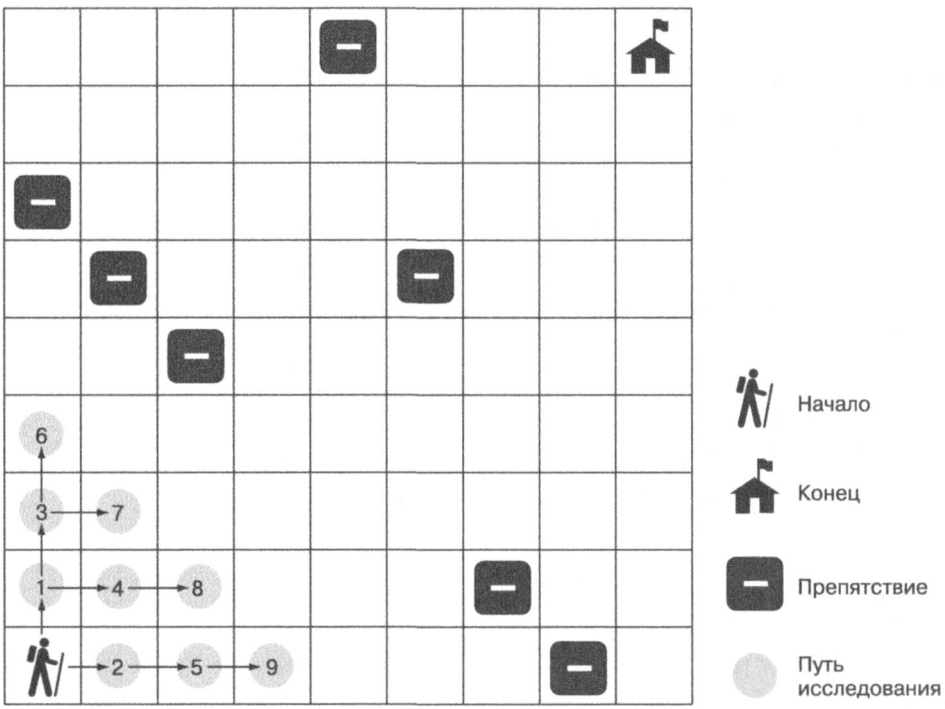


Рис. 2.5. При поиске в ширину сначала просматриваются ближайшие к начальной позиции элементы

Чтобы понять, почему поиск в глубину иногда возвращает результат быстрее, чем поиск в ширину, представьте себе, что вы ищете метку на определенном слое луковицы. Тот, кто использует стратегию поиска в глубину, вонзит нож в центр луковицы и исследует случайно вырезанные куски. Если отмеченный слой окажется рядом с вырезанным куском, есть вероятность, что он будет найден быстрее, чем с помощью стратегии поиска в ширину, при которой нужно кропотливо очищать луковицу, снимая по одному слою за раз.

Чтобы получить более полное представление о том, почему при поиске в ширину всегда определяется кратчайший путь, если только он существует, попробуйте найти железнодорожный маршрут между Бостоном и Нью-Йорком с наименьшим количеством остановок. Если вы будете продолжать двигаться в выбранном на-

правлении и возвращаться назад, когда попали в тупик (как при поиске в глубину), то можете сначала найти маршрут до Сиэтла, прежде чем попасть обратно в Нью-Йорк. Однако при поиске в ширину вы сначала проверите все станции, находящиеся в одной остановке от Бостона. Затем — все станции, находящиеся в двух остановках от Бостона. Затем — в трех остановках от Бостона. Так будет продолжаться до тех пор, пока вы не доберетесь до Нью-Йорка. Поэтому, найдя Нью-Йорк, поймете, что определили маршрут с наименьшим количеством станций, поскольку уже проверили все станции, находящиеся на меньшем расстоянии от Бостона, и ни одна из них не была Нью-Йорком.

Очереди

Для реализации алгоритма BFS нужна структура данных, известная как *очередь*. Если стек — это структура LIFO, то очередь — структура FIFO (first-in-first-out, «первым вошел — первым вышел»). Очередь встречается нам и в обычной жизни — например, цепочка людей, ожидающих у входа в туалет. Кто раньше занял очередь, тот первым туда идет. Очередь как минимум имеет те же методы `push()` и `pop()`, что и стек. В сущности, реализация `Queue` на основе имеющегося в Python типа `deque` практически идентична реализации `Stack`, единственными изменениями являются удаление элементов с левого, а не с правого конца `_container` и замена `list` на `deque` (я использую здесь слово «левый» для обозначения начала резервного хранилища). Элементы, расположенные с левого края, — это самые старые элементы, все еще находящиеся в `deque` (с точки зрения времени попадания в хранилище), поэтому они являются первыми извлекаемыми элементами (листинг 2.22).

Листинг 2.22. generic_search.py (продолжение)

```
class Queue(Generic[T]):
    def __init__(self) -> None:
        self._container: Deque[T] = Deque()

    @property
    def empty(self) -> bool:
        return not self._container # не равно True для пустого контейнера

    def push(self, item: T) -> None:
        self._container.append(item)

    def pop(self) -> T:
        return self._container.popleft() # FIFO

    def __repr__(self) -> str:
        return repr(self._container)
```

СОВЕТ

Почему в реализации Queue в качестве резервного хранилища применяется deque, тогда как в реализации Stack использован list? Это связано с тем, откуда извлекаются данные в процессе операции pop. В стеке мы помещаем элементы с правого конца и извлекаем их тоже справа. В очереди мы помещаем элементы справа, а извлекаем слева. Структура данных list в Python имеет эффективный метод pop для извлечения справа, но не слева. Структура данных deque позволяет эффективно извлекать данные с любой стороны. Так что в deque есть встроенный метод popleft(), у которого нет эквивалента в list. Конечно, можно было бы найти другие способы применения list в качестве резервного хранилища для очереди, но они были бы менее эффективными. Извлечение данных слева в deque — это операция сложности O(1), тогда как для list это операция сложности O(n). В случае применения list после извлечения данных слева каждый последующий элемент должен быть сдвинут на одну позицию влево, что делает этот тип данных неэффективным.

Алгоритм BFS

Удивительно, но алгоритм поиска в ширину идентичен алгоритму поиска в глубину, только область поиска не стек, а очередь. Такое изменение области поиска меняет последовательность просмотра состояний и гарантирует, что первыми будут просмотрены состояния, ближайшие к начальному (листинг 2.23).

Листинг 2.23. generic_search.py (продолжение)

```
def bfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
        List[T]]) -> Optional[Node[T]]:
    # frontier - это то, что мы только собираемся проверить
    frontier: Queue[Node[T]] = Queue()
    frontier.push(Node(initial, None))
    # explored - это то, что уже проверено
    explored: Set[T] = {initial}

    # продолжаем, пока есть что проверять
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # если мы нашли искомое, то процесс закончен
        if goal_test(current_state):
            return current_node
        # ищем неисследованные ячейки, в которые можно перейти
        for child in successors(current_state):
            if child in explored: # пропустить состояния, которые уже исследовали
                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None # все проверили, пути к целевой точке не нашли
```

Если вы запустите `bfs()`, то увидите, что эта функция всегда находит кратчайшее решение для рассматриваемого лабиринта. Следующий пробный запуск добавляется в раздел файла `if __name__ == "__main__":` сразу после предыдущего кода, так чтобы можно было сравнить результаты для одного и того же лабиринта (листинг 2.24).

Листинг 2.24. maze.py (продолжение)

```
# Тестирование BFS
solution2: Optional[Node[MazeLocation]] = bfs(m.start, m.goal_test,
    m.successors)
if solution2 is None:
    print("No solution found using breadth-first search!")
else:
    path2: List[MazeLocation] = node_to_path(solution2)
    m.mark(path2)
    print(m)
    m.clear(path2)
```

Это удивительно, но мы можем, не меняя алгоритм, просто изменить структуру данных, к которой он обращается, и получить кардинально различные результаты. Далее показан результат вызова `bfs()` для того же лабиринта, для которого ранее мы вызывали `dfs()`. Обратите внимание на то, что на этот раз звездочки обозначают более прямой путь к цели, чем в предыдущем примере:

```
S    X X
*X
*      X
*XX      X
* X
* X  X
*X
*
*    X   X
*********G
```

2.2.5. Поиск по алгоритму A*

Очистка луковицы слой за слоем, как при поиске в ширину, может занять очень много времени. Поиск по алгоритму A*, подобно BFS, стремится найти кратчайший путь от начального к конечному состоянию. В отличие от предыдущей реализации BFS при поиске A* используется объединение функции затрат и эвристической функции, что позволяет сосредоточить поиск на путях, которые, скорее всего, быстро приведут к цели.

Функция затрат $g(n)$ проверяет затраты, необходимые для того, чтобы добраться до определенного состояния. В случае с нашим лабиринтом это было бы количество шагов, которые пришлось бы пройти, чтобы добраться до нужного состояния.

Эвристическая функция $h(n)$ позволяет оценить затраты, необходимые для того, чтобы из заданного состояния достичь целевого состояния. Можно доказать, что если $h(n)$ — *допустимая эвристическая функция*, то найденный конечный путь будет оптимальным. Допустимая эвристическая функция — это функция, которая никогда не переоценивает затраты на достижение цели. На двумерной плоскости примером такой эвристической функции является расстояние по прямой линии, поскольку прямая линия — всегда кратчайший путь[1].

Общие затраты для любого рассматриваемого состояния определяются функцией $f(n)$, которая является простым объединением $g(n)$ и $h(n)$. В сущности, $f(n) = g(n) + h(n)$. При выборе следующего рассматриваемого состояния из области поиска алгоритм A* выбирает состояние с наименьшим $f(n)$. Именно этим он отличается от алгоритмов BFS и DFS.

Очереди с приоритетом

Чтобы выбрать из области поиска состояние с наименьшим $f(n)$, при поиске A* в качестве структуры данных используется *очередь с приоритетом* для данной области поиска. В очереди с приоритетом элементы сохраняются во внутренней последовательности, так что первый извлеченный элемент — это элемент с наивысшим приоритетом. (В нашем случае наиболее приоритетным является элемент с наименьшим значением $f(n)$.) Обычно это означает задействование внутри очереди бинарной кучи, что дает сложность $O(\lg n)$ для помещения в очередь и извлечения из нее.

В стандартной библиотеке Python есть функции `heappush()` и `heappop()`, которые принимают список и сохраняют его в виде бинарной кучи. Мы можем реализовать очередь с приоритетом, создав тонкую обертку вокруг этих стандартных библиотечных функций. Класс `PriorityQueue` будет похож на классы `Stack` и `Queue` с методами `push()` и `pop()`, модифицированными для использования `heappush()` и `heappop()` (листинг 2.25).

Листинг 2.25. generic_search.py (продолжение)

```
class PriorityQueue(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container # не True для пустого контейнера

    def push(self, item: T) -> None:
        heappush(self._container, item) # поместить в очередь по приоритету
```

[1] Для получения дополнительной информации об эвристике см. книгу: *Рассел С., Норвиг П.* Искусственный интеллект: современный подход. 3-е изд. — М.: Вильямс, 2019 (*Russell S., Norvig P.* Artificial Intelligence: A Modern Approach, 3rd ed. — Pearson, 2010).

```
    def pop(self) -> T:
        return heappop(self._container) # извлечь по приоритету

    def __repr__(self) -> str:
        return repr(self._container)
```

Чтобы определить приоритет конкретного элемента по сравнению с аналогичными элементами, в `heappush()` и `heappop()` выполняется сравнение этих элементов с помощью оператора `<`. Вот почему ранее нам нужно было реализовать `__lt__()` для `Node`. Объекты `Node` сравниваются между собой по их значениям $f(n)$, которые являются простой суммой свойств `cost` и `heuristic`.

Эвристика

Эвристика — это интуитивное представление о том, как решить задачу[1]. В случае прохода по лабиринту эвристика стремится выбрать в нем лучшую точку для поиска следующей точки в желании добраться до цели. Другими словами, это обоснованное предположение о том, какие узлы из области поиска находятся ближе всего к цели. Как уже упоминалось, если эвристика, используемая при поиске по алгоритму A*, дает точный относительный результат и является допустимой (никогда не переоценивает расстояние), то A* составит кратчайший путь. Эвристика, которая вычисляет меньшие значения, в итоге дает поиск по большему количеству состояний, тогда как эвристика, значение которой ближе к точному реальному расстоянию (но не больше его, иначе эвристика будет недопустимой), выполняет поиск по меньшему количеству состояний. Следовательно, идеальная эвристика максимально приближается к реальному расстоянию, но не превышает его.

Евклидово расстояние

Как известно из геометрии, кратчайшим путем между двумя точками является прямая. Следовательно, вполне оправданно, что для поиска пути по лабиринту прямолинейная эвристика всегда будет допустимой. Согласно теореме Пифагора евклидово расстояние составляет `distance = √((difference in x)2 + (difference in y)2)`. Для наших лабиринтов разность по оси X эквивалентна разности в столбцах между двумя точками лабиринта, а разность по оси Y — разности в строках. Обратите внимание: мы снова реализуем этот код в `maze.py` (листинг 2.26).

Листинг 2.26. maze.py (продолжение)

```
def euclidean_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = ml.column - goal.column
        ydist: int = ml.row - goal.row
        return sqrt((xdist * xdist) + (ydist * ydist))
    return distance
```

[1] Подробнее об эвристике для поиска пути по алгоритму A* читайте в главе «Эвристика» публикации: *Patel A.* Amit's Thoughts on Pathfinding, http://mng.bz/z7O4.

Функция `euclidean_distance()` возвращает другую функцию. Такие языки, как Python, которые поддерживают функции первого класса, допускают применение этого интересного шаблона. Функция `distance()` фиксирует целевую точку `MazeLocation goal`, которую передает в `euclidean_distance()`. Фиксация целевой точки означает, что `distance()` может ссылаться на эту переменную при каждом вызове (постоянно). Функция, которую она возвращает, использует `goal` для выполнения своих вычислений. Этот шаблон позволяет создать функцию, требующую меньше параметров. Функция, возвращаемая `distance()`, принимает в качестве аргумента только начальную точку прохода по лабиринту и всегда знает конечную точку.

Применение евклидова расстояния для прохода по лабиринту — в данном случае по сетке улиц Манхэттена — показано на рис. 2.6.

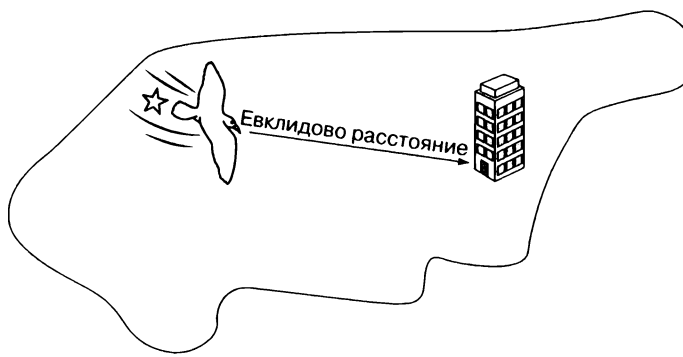


Рис. 2.6. Евклидово расстояние — это длина прямой линии, соединяющей начальную и конечную точки

Манхэттенское расстояние

Евклидово расстояние — это прекрасно, но для нашей конкретной задачи (лабиринт, в котором можно передвигаться только в одном из четырех направлений) можно добиться еще большего. Манхэттенское расстояние определяется навигацией по улицам Манхэттена — самого известного района Нью-Йорка, улицы которого образуют регулярную сетку. Чтобы добраться из любой его точки в любую другую точку, нужно пройти определенное количество кварталов по горизонтали и вертикали. (На Манхэттене почти нет диагональных улиц.) Расстояние в Манхэттене определяется путем простого вычисления разности в строках между двумя точками лабиринта и суммирования ее с разностью в столбцах (листинг 2.27). Вычисление манхэттенского расстояния показано на рис. 2.7.

Листинг 2.27. maze.py (продолжение)

```
def manhattan_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
```

```
        xdist: int = abs(ml.column - goal.column)
        ydist: int = abs(ml.row - goal.row)
        return (xdist + ydist)
    return distance
```

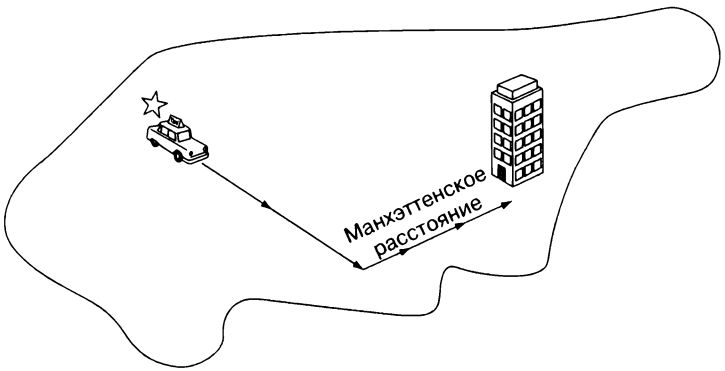


Рис. 2.7. У манхэттенского расстояния нет диагоналей. Путь должен проходить по параллельным или перпендикулярным линиям

Поскольку такая эвристика более точно соответствует действительной навигации по нашим лабиринтам (перемещаясь по вертикали и горизонтали, а не напрямую по диагональным линиям), она оказывается ближе к фактическому расстоянию между любой точкой лабиринта и точкой назначения, чем евклидово расстояние. Поэтому, когда поиск по алгоритму A* для лабиринтов опирается на манхэттенское расстояние, это обеспечивает поиск по меньшему количеству состояний, чем когда поиск A* опирается на евклидово расстояние. Пути решения по-прежнему будут оптимальными, поскольку манхэттенское расстояние допустимо (никогда не переоценивает реальное расстояние) для лабиринтов, в которых разрешены только четыре направления движения.

Алгоритм A*

Чтобы перейти от поиска BFS к поиску A*, нужно внести в код пару незначительных изменений. Прежде всего изменить область поиска с очереди на очередь с приоритетом. Таким образом, в области поиска появятся узлы с наименьшим $f(n)$. Затем нужно заменить исследуемый набор на словарь. Он позволит отслеживать наименьшие затраты ($g(n)$) для каждого узла, который мы можем посетить. Теперь, когда используется эвристическая функция, может оказаться, что некоторые узлы будут проверены дважды, если эвристика окажется несовместимой. Если узел, найденный в новом направлении, требует меньших затрат, чем тогда, когда мы его посетили в прошлый раз, то мы предпочтем новый маршрут.

Для простоты функция `astar()` не принимает в качестве параметра функцию вычисления затрат. Вместо этого мы считаем, что затраты для каждого шага

в лабиринте равны 1. Каждому новому узлу назначаются затраты, основанные на этой простой формуле, а также эвристический показатель, вычисленный с использованием новой функции `heuristic()`, передаваемой функции поиска в качестве параметра. За исключением этих изменений, функция `astar()` очень похожа на `bfs()` (листинг 2.28). Откройте их в соседних окнах для сравнения.

Листинг 2.28. generic_search.py

```
def astar(initial: T, goal_test: Callable[[T], bool], successors:
        Callable[[T], List[T]], heuristic: Callable[[T], float]) ->
        Optional[Node[T]]:
    # frontier - то, куда мы хотим двигаться
    frontier: PriorityQueue[Node[T]] = PriorityQueue()
    frontier.push(Node(initial, None, 0.0, heuristic(initial)))
    # explored - то, что мы уже просмотрели
    explored: Dict[T, float] = {initial: 0.0}

    # продолжаем, пока есть что просматривать
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # если цель найдена, мы закончили
        if goal_test(current_state):
            return current_node
        # проверяем, в какую из неисследованных ячеек направиться
        for child in successors(current_state):
            new_cost: float = current_node.cost + 1
            # 1 - для сетки, для более сложных приложений здесь
            # должна быть функция затрат

            if child not in explored or explored[child] > new_cost:
                explored[child] = new_cost
                frontier.push(Node(child, current_node, new_cost,
                heuristic(child)))
    return None # все проверили, пути к целевой точке не нашли
```

Поздравляю! Если вы завершили этот путь, то узнали не только о том, как пройти по лабиринту, но и о некоторых общих функциях поиска, которые сможете использовать во всевозможных поисковых приложениях. Алгоритмы DFS и BFS подходят для множества небольших наборов данных и пространств состояний, где производительность не очень важна. В некоторых ситуациях DFS превосходит BFS, но преимущество BFS заключается в том, что этот алгоритм всегда гарантирует оптимальный путь. Интересно, что у BFS и DFS почти идентичные реализации, различающиеся только применением очереди вместо стека в качестве области поиска. Чуть более сложный поиск A* в сочетании с хорошей последовательной допустимой эвристикой не только обеспечивает оптимальный путь, но и намного превосходит BFS по производительности. А поскольку все три

функции были реализованы в параметризованном виде, их можно использовать практически в любом пространстве поиска — достаточно просто написать `import generic_search`.

Вперед — попробуйте применить `astar()` к тому же лабиринту, что был задействован в разделе тестирования в файле `maze.py` (листинг 2.29).

Листинг 2.29. maze.py (продолжение)

```
# Тестирование A*
distance: Callable[[MazeLocation], float] = manhattan_distance(m.goal)
solution3: Optional[Node[MazeLocation]] = astar(m.start, m.goal_test,
    m.successors, distance)
if solution3 is None:
    print("No solution found using A*!")
else:
    path3: List[MazeLocation] = node_to_path(solution3)
    m.mark(path3)
    print(m)
```

Интересно, что результат немного отличается от `bfs()`, несмотря на то что и `bfs()`, и `astar()` находят оптимальные (равные по длине) пути. Из-за своей эвристики `astar()` сразу проходит по диагонали к цели. В итоге этот алгоритм проверяет меньше состояний, чем `bfs()`, что обеспечивает более высокую производительность. Если хотите в этом убедиться, добавьте счетчик состояний для каждого алгоритма:

```
S**    X X
 X**
   *     X
 XX*       X
  X*
  X**X
X   ****
       *
     X * X
       **G
```

2.3. Миссионеры и людоеды

Три миссионера и три людоеда находятся на западном берегу реки. У них есть каноэ, в котором помещаются два человека, и все они должны переехать на восточный берег реки. Нельзя, чтобы на любой стороне реки в какой-то момент оказалось больше людоедов, чем миссионеров, иначе каннибалы съедят миссионеров. Кроме того, в каноэ всегда должен находиться хотя бы один человек, чтобы пересечь реку. Какая последовательность переправ позволит успешно перевезти всех через реку? Иллюстрация задачи — на рис. 2.8.

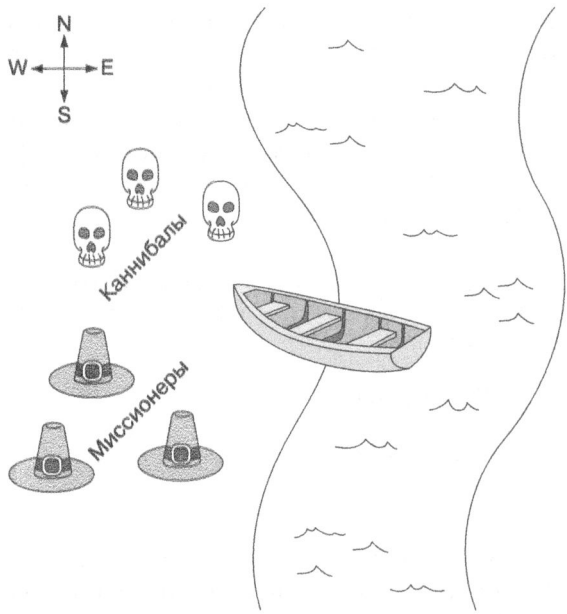


Рис. 2.8. Миссионеры и каннибалы должны использовать свое единственное каноэ, чтобы переправить всех через реку с западного на восточный берег. Если людоедов в какой-то момент окажется больше, чем миссионеров, они их съедят

2.3.1. Представление задачи

Представим задачу с помощью структуры, которая отслеживает ситуацию на западном берегу. Сколько миссионеров и людоедов здесь находятся? Причалена ли к западному берегу лодка? Получив эти знания, мы можем выяснить, кто находится на восточном берегу, потому что все, кто не на западном берегу, находятся на восточном.

Прежде всего создадим небольшую вспомогательную переменную для отслеживания максимального количества миссионеров или людоедов. Затем определим основной класс (листинг 2.30).

Листинг 2.30. missionaries.py

```
from __future__ import annotations
from typing import List, Optional
from generic_search import bfs, Node, node_to_path

MAX_NUM: int = 3

class MCState:
    def __init__(self, missionaries: int, cannibals: int, boat: bool) -> None:
        self.wm: int = missionaries # миссионеры с западного берега
```

```
        self.wc: int = cannibals # людоеды с западного берега
        self.em: int = MAX_NUM - self.wm # миссионеры с восточного берега
        self.ec: int = MAX_NUM - self.wc # людоеды с восточного берега
        self.boat: bool = boat

    def __str__(self) -> str:
        return ("On the west bank there are {} missionaries and {}
        cannibals.\n"
            "On the east bank there are {} missionaries and {} cannibals.\n"
            "The boat is on the {} bank.")\
        .format(self.wm, self.wc, self.em, self.ec, ("west" if self.boat
    else "east"))
```

Класс `MCState` инициализируется в зависимости от количества миссионеров и людоедов, находящихся на западном берегу, а также от местоположения лодки. Он умеет красиво выводить свои данные, что окажется полезным позже, при отображении решения задачи.

Действуя в рамках существующих функций поиска, мы должны определить функцию проверки того, является ли состояние целевым, и функцию для поиска преемников для любого состояния. Функция проверки целевого состояния, как и в задаче прохода по лабиринту, довольно проста. Наша цель состоит в том, чтобы просто достичь разрешенного состояния, при котором все миссионеры и людоеды оказываются на восточном берегу. Мы добавим это в `MCState` в качестве метода (листинг 2.31).

Листинг 2.31. missionaries.py (продолжение)

```
def goal_test(self) -> bool:
    return self.is_legal and self.em == MAX_NUM and self.ec == MAX_NUM
```

Чтобы создать функцию преемников, необходимо перебрать все возможные шаги, которые могут быть сделаны для перевозки с одного берега на другой, и затем проверить, приводит ли каждый из них к разрешенному состоянию. Напомню, что разрешенное состояние — такое, при котором на каждом берегу количество людоедов не превышает количества миссионеров. Чтобы определить это, мы можем создать вспомогательное свойство (как метод в `MCState`), проверяющее, допустимо ли данное состояние (листинг 2.32).

Листинг 2.32. missionaries.py (продолжение)

```
@property
def is_legal(self) -> bool:
    if self.wm < self.wc and self.wm > 0:
        return False
    if self.em < self.ec and self.em > 0:
        return False
    return True
```

Реальная функция преемников немного многословна. Это сделано для того, чтобы она была понятной. Функция пытается добавить каждую возможную

комбинацию из одного или двух человек, пересекающих реку с того берега, на котором в настоящий момент находится каноэ. Когда все возможные ходы будут добавлены, функция отфильтровывает те, что действительно допустимы, используя списковое включение (или генератор списков, list comprehension). Эта функция также является методом `MCState` (листинг 2.33).

Листинг 2.33. missionaries.py (продолжение)

```
def successors(self) -> List[MCState]:
    sucs: List[MCState] = []
    if self.boat: # лодка на западном берегу
        if self.wm > 1:
            sucs.append(MCState(self.wm - 2, self.wc, not self.boat))
        if self.wm > 0:
            sucs.append(MCState(self.wm - 1, self.wc, not self.boat))
        if self.wc > 1:
            sucs.append(MCState(self.wm, self.wc - 2, not self.boat))
        if self.wc > 0:
            sucs.append(MCState(self.wm, self.wc - 1, not self.boat))
        if (self.wc > 0) and (self.wm > 0):
            sucs.append(MCState(self.wm - 1, self.wc - 1, not self.boat))
    else: # лодка на восточном берегу
        if self.em > 1:
            sucs.append(MCState(self.wm + 2, self.wc, not self.boat))
        if self.em > 0:
            sucs.append(MCState(self.wm + 1, self.wc, not self.boat))
        if self.ec > 1:
            sucs.append(MCState(self.wm, self.wc + 2, not self.boat))
        if self.ec > 0:
            sucs.append(MCState(self.wm, self.wc + 1, not self.boat))
        if (self.ec > 0) and (self.em > 0):
            sucs.append(MCState(self.wm + 1, self.wc + 1, not self.boat
    return [x for x in sucs if x.is_legal]
```

2.3.2. Решение

Теперь у нас есть все необходимое для решения задачи. Напомню: когда мы решаем задачу, используя функции поиска `bfs()`, `dfs()` и `astar()`, то получаем узел, который с помощью `node_to_path()` преобразуем в список состояний, приводящий к решению. Еще нам потребуется способ преобразовать этот список в понятную наглядную последовательность шагов для решения задачи о миссионерах и людоедах.

Функция `display_solution()` преобразует путь решения в наглядный вывод — удобочитаемое решение задачи. Эта функция перебирает все состояния, вошедшие в готовый путь, а также отслеживает последнее состояние. Она анализирует разницу между последним состоянием и состоянием, которое отображается в настоящий момент, чтобы выяснить, сколько миссионеров и людоедов в каком направлении пересекло реку (листинг 2.34).

Листинг 2.34. missionaries.py (продолжение)

```
def display_solution(path: List[MCState]):
    if len(path) == 0: # санитарная проверка
        return
    old_state: MCState = path[0]
    print(old_state)
    for current_state in path[1:]:
        if current_state.boat:
            print("{} missionaries and {} cannibals moved
                  from the east to the west bank.\n"
                .format(old_state.em - current_state.em, old_state.ec -
                        current_state.ec))
        else:
            print("{} missionaries and {} cannibals moved from
                  the west to the east bank.\n"
                .format(old_state.wm - current_state.wm, old_state.wc -
                        current_state.wc))
        print(current_state)
        old_state = current_state
```

В функции `display_solution()` использовано преимущество, обусловленное тем, что `MCState` умеет вывести красивую сводку о своих данных с помощью `__str__()`.

Последнее, что нам остается сделать, — на самом деле решить задачу о миссионерах и каннибалах. Для этого удобно будет повторно задействовать функцию поиска, которую мы уже написали, потому что она является параметризованной (листинг 2.35). Здесь применяется `bfs()`, потому что использование `dfs()` потребовало бы пометить ссылочно разные состояния с одинаковым значением как равные, а `astar()` требует эвристики.

Листинг 2.35. missionaries.py (продолжение)

```
if __name__ == "__main__":
    start: MCState = MCState(MAX_NUM, MAX_NUM, True)
    solution: Optional[Node[MCState]] =
      bfs(start, MCState.goal_test, MCState.successors)
    if solution is None:
        print("No solution found!")
    else:
        path: List[MCState] = node_to_path(solution)
        display_solution(path)
```

Приятно видеть, насколько гибкими могут быть параметризованные функции поиска. Их можно легко адаптировать для решения разнообразных задач. Вы должны увидеть что-то вроде следующего (сокращенного) списка:

```
On the west bank there are 3 missionaries and 3 cannibals.
On the east bank there are 0 missionaries and 0 cannibals.
The boast is on the west bank.
0 missionaries and 2 cannibals moved from the west bank to the east bank.

On the west bank there are 3 missionaries and 1 cannibals.
```

```
On the east bank there are 0 missionaries and 2 cannibals.
The boast is on the east bank.
0 missionaries and 1 cannibals moved from the east bank to the west bank.

...

On the west bank there are 0 missionaries and 0 cannibals.
On the east bank there are 3 missionaries and 3 cannibals.
The boast is on the east bank.
```

2.4. Реальные приложения

Поиск играет важную роль во всех полезных программах. В некоторых случаях это центральный элемент приложения (Google Search, Spotlight, Lucene), в других он является основой для использования структур, на которые опирается система хранения данных. Знание правильного алгоритма поиска для применения к структуре данных имеет важное значение для производительности. Например, было бы слишком затратно выполнять в отсортированной структуре данных линейный поиск вместо двоичного.

A* — один из наиболее распространенных алгоритмов поиска пути. Его превосходят только алгоритмы, которые выполняют предварительный расчет в пространстве поиска. В слепом поиске у A* все еще нет конкурентов в любых сценариях, и это сделало его неотъемлемым компонентом всех областей, от планирования маршрута до определения кратчайшего способа синтаксического анализа языка программирования. В большинстве картографических программ, прокладывающих маршрут, таких как Google Maps, для навигации используется алгоритм Дейкстры (вариант A*) (подробнее об алгоритме Дейкстры читайте в главе 4). Всякий раз, когда игровой персонаж с элементами искусственного интеллекта находит кратчайший путь от одной точки игровой карты до другой без вмешательства человека, он, скорее всего, применит алгоритм A*.

Поиск в ширину и поиск в глубину часто являются основой для более сложных алгоритмов, таких как поиск с равномерными затратами и поиск в обратном направлении, с которыми вы познакомитесь в следующей главе. Поиска в ширину часто достаточно для прокладки кратчайшего пути в не очень большом графе. Но поскольку он похож на A*, для большого графа его легко заменить на A*, если существует хорошая эвристика.

2.5. Упражнения

1. Продемонстрируйте преимущество в производительности бинарного поиска по сравнению с линейным поиском, создав список из миллиона чисел и определив, сколько времени потребуется созданным в этой главе функциям `linear_contains()` и `binary_contains()` для поиска в нем различных чисел.

2. Добавьте в `dfs()`, `bfs()` и `astar()` счетчик, который позволит увидеть, сколько состояний просматривает каждая из этих функций в одном и том же лабиринте. Найдите значения счетчика для 100 различных лабиринтов, чтобы получить статистически значимые результаты.
3. Найдите решение задачи о миссионерах и каннибалах для разного начального числа миссионеров и каннибалов. Подсказка: вам может понадобиться добавить в `MCState` переопределения методов `__eq__()` и `__hash__()`.

3 Задачи с ограничениями

Многие задачи, для решения которых используются компьютерные вычисления, можно в целом отнести к категории задач с ограничениями (constraint-satisfaction problems, CSP). CSP-задачи состоят из *переменных*, допустимые значения которых попадают в определенные диапазоны, известные как *области определения*. Для того чтобы решить задачу с ограничениями, необходимо удовлетворить существующие ограничения для переменных. Три основных понятия — переменные, области определения и ограничения — просты и понятны, а благодаря их универсальности задачи с ограничениями получили широкое применение.

Рассмотрим пример такой задачи. Предположим, что вы пытаетесь назначить на пятницу встречу для Джо, Мэри и Сью. Сью должна встретиться хотя бы с одним человеком. В этой задаче планирования переменными могут быть три человека — Джо, Мэри и Сью. Областью определения для каждой переменной могут быть часы, когда свободен каждый из них. Например, у переменной «Мэри» область определения составляет 2, 3 и 4 часа пополудни. У этой задачи есть также два ограничения. Во-первых, Сью должна присутствовать на встрече. Во-вторых, на встрече должны присутствовать по крайней мере два человека. Решение этой задачи с ограничениями определяется тремя переменными, тремя областями определения и двумя ограничениями, тогда задача будет решена и при этом не придется объяснять пользователю, *как именно* (рис. 3.1).

В некоторых языках программирования, таких как Prolog и Picat, есть встроенные средства для решения задач с ограничениями. В других языках обычным подходом является создание структуры, которая включает в себя поиск с возвратами и несколько эвристик для повышения производительности поиска. В этой главе

мы сначала создадим структуру для CSP-задач, которая будет решать их простым рекурсивным поиском с возвратами. Затем воспользуемся этой структурой для решения нескольких примеров таких задач.

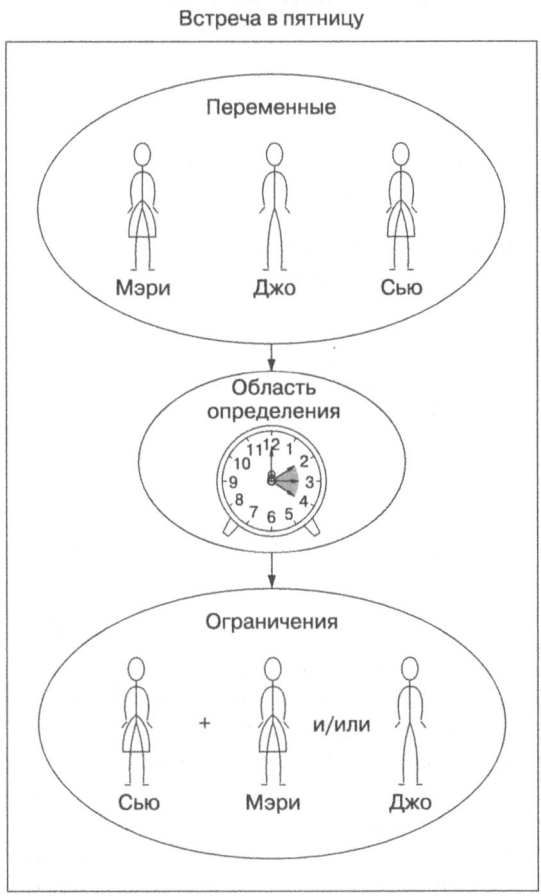


Рис. 3.1. Задачи планирования — это классическое применение структур для удовлетворения ограничений

3.1. Построение структуры для задачи с ограничениями

Определим ограничения посредством класса `Constraint`. Каждое ограничение `Constraint` состоит из переменных `variables`, которые оно ограничивает, и метода `satisfied()`, который проверяет, выполняется ли оно. Определение того, выполняется ли ограничение, является основной логикой, входящей в определение

конкретной задачи с ограничениями. Реализацию по умолчанию нужно переопределить. Именно так и должно быть, потому что мы определяем `Constraint` как абстрактный базовый класс. Абстрактные базовые классы не предназначены для реализации. Только их подклассы, которые переопределяют и реализуют свои абстрактные методы `@abstractmethod`, предназначены для действительного использования (листинг 3.1).

Листинг 3.1. csp.py

```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod

V = TypeVar('V') # Тип variable для переменной
D = TypeVar('D') # Тип domain для области определения

# Базовый класс для всех ограничений
class Constraint(Generic[V, D], ABC):
    # Переменные, для которых существует ограничение
    def __init__(self, variables: List[V]) -> None:
        self.variables = variables

    # Необходимо переопределить в подклассах
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
```

СОВЕТ

Абстрактные базовые классы играют роль шаблонов в иерархии классов. В других языках, таких как C++, они получили более широкое распространение, чем в Python, как свойство, ориентированное на пользователя. В сущности, они появились в Python примерно на середине жизненного пути языка. При этом многие классы коллекций из стандартной библиотеки Python реализованы с помощью абстрактных базовых классов. Общий совет: не используйте их в своем коде, если не уверены, что строите структуру, на основе которой будут создаваться другие классы, а не просто иерархию классов для внутреннего применения. Подробнее об этом читайте в главе 11 книги: *Ramalho L.* Fluent Python (O'Reilly, 2015).

Центральным элементом нашей структуры соответствия ограничениям будет класс с названием `CSP` (листинг 3.2). `CSP` — это место, где собраны все переменные, области определения и ограничения. С точки зрения подсказок типов класс `CSP` использует универсальные средства, чтобы быть достаточно гибким, работать с любыми значениями переменных и областей определения (где `V` — это значения переменных, а `D` — областей определения). В `CSP` коллекции `variables`, `domains` и `constraints` имеют ожидаемые типы. Коллекция `variables` — это `list` для переменных, `domains` — `dict` с соответствием переменных спискам возможных значений

(областям определения этих переменных), а `constraints` — `dict`, где каждой переменной соответствует `list` наложенных на нее ограничений.

Листинг 3.2. csp.py (продолжение)

```
# Задача с ограничениями состоит из переменных типа V,
# которые имеют диапазоны значений, известные как области определения,
# типа D и ограничений, которые определяют, является ли допустимым
# выбор данной области определения для данной переменной
class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
        self.variables: List[V] = variables # переменные, которые будут ограничены
        self.domains: Dict[V, List[D]] = domains # домен каждой переменной
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        for variable in self.variables:
            self.constraints[variable] = []
            if variable not in self.domains:
                raise LookupError("Every variable should have a domain
                                  assigned to it.")

    def add_constraint(self, constraint: Constraint[V, D]) -> None:
        for variable in constraint.variables:
            if variable not in self.variables:
                raise LookupError("Variable in constraint not in CSP")
            else:
                self.constraints[variable].append(constraint)
```

Инициализатор `__init__()` создает `constraints` `dict`. Метод `add_constraint()` просматривает все переменные, к которым относится данное ограничение, и добавляет себя в соответствие `constraints` для каждой такой переменной. Оба метода имеют простейшую проверку ошибок и вызывают исключение, если `variable` отсутствует в области определения или существует `constraint` для несуществующей переменной.

Как узнать, соответствует ли данная конфигурация переменных и выбранных значений области определения заданным ограничениям? Мы будем называть такую заданную конфигурацию *присваиванием*. Нам нужна функция, которая проверяла бы каждое ограничение для заданной переменной по отношению к присваиванию, чтобы увидеть, удовлетворяет ли значение переменной в присваивании этим ограничениям. В листинге 3.3 реализована функция `constant()` как метод класса `CSP`.

Листинг 3.3. csp.py (продолжение)

```
# Проверяем, соответствует ли присваивание значения, проверяя все ограничения
# для данной переменной
def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
    for constraint in self.constraints[variable]:
        if not constraint.satisfied(assignment):
            return False
    return True
```

Метод `constant()` перебирает все ограничения для данной переменной (это всегда будет переменная, только что добавленная в присваивание) и проверяет, выполняется ли ограничение, учитывая новое присваивание. Если присваивание удовлетворяет всем ограничениям, возвращается `True`. Если какое-либо ограничение, наложенное на переменную, не выполняется, возвращается значение `False`.

Для поиска решения задачи в такой структуре выполнения ограничений будет использоваться простой поиск с возвратами. *Возвраты* — это подход, при котором, если поиск зашел в тупик, мы возвращаемся к последней известной точке, где было принято решение, перед тем как зайти в тупик, и выбираем другой путь. Если вам кажется, что это похоже на поиск в глубину из главы 2, то вы правы. Поиск с возвратом, реализованный в следующей функции `backtracking_search()`, является своего рода рекурсивным поиском в глубину, в котором объединены идеи, описанные в главах 1 и 2. Эта функция добавляется в качестве метода в класс `CSP` (листинг 3.4).

Листинг 3.4. csp.py (продолжение)

```
def backtracking_search(self, assignment: Dict[V, D] = {})
    -> Optional[Dict[V, D]]:
    # присваивание завершено, если существует присваивание
    # для каждой переменной (базовый случай)
    if len(assignment) == len(self.variables):
        return assignment

    # получить все переменные из CSP, но не из присваивания
    unassigned: List[V] = [v for v in self.variables if v not in assignment]

    # получить все возможные значения области определения
    # для первой переменной без присваивания
    first: V = unassigned[0]
    for value in self.domains[first]:
        local_assignment = assignment.copy()
        local_assignment[first] = value
        # если нет противоречий, продолжаем рекурсию
        if self.consistent(first, local_assignment):
            result: Optional[Dict[V, D]] =
              self.backtracking_search(local_assignment)
            # если результат не найден, заканчиваем возвраты
            if result is not None:
                return result
    return None
```

Исследуем `backtracking_search()` построчно.

```
if len(assignment) == len(self.variables):
    return assignment
```

Базовый случай для рекурсивного поиска означает, что нужно найти правильное присваивание для каждой переменной. Сделав это, мы возвращаем первый валидный экземпляр решения (и не продолжаем поиск).

```
unassigned: List[V] = [v for v in self.variables if v not in assignment]
first: V = unassigned[0]
```

Чтобы выбрать новую переменную, область определения которой будем исследовать, мы просто просматриваем все переменные и находим первую, которая не имеет присваивания. Для этого создаем список `list` переменных в `self.variables`, но не в `assignment` через генератор списков и называем его `unassigned`. Затем извлекаем из `unassigned` первое значение.

```
for value in self.domains[first]:
    local_assignment = assignment.copy()
    local_assignment[first] = value
```

Мы пытаемся присвоить этой переменной все возможные значения области определения по очереди. Новое присваивание для каждой переменной сохраняется в локальном словаре `local_assignment`.

```
if self.consistent(first, local_assignment):
    result: Optional[Dict[V, D]] =
      self.backtracking_search(local_assignment)
    if result is not None:
        return result
```

Если новое присваивание в `local_assignment` согласуется со всеми ограничениями, что проверяется с помощью `consistent()`, мы продолжаем рекурсивный поиск для нового присваивания. Если новое присваивание оказывается завершенным (базовый случай), передаем его вверх по цепочке рекурсии.

```
return None # нет решения
```

Наконец, если мы рассмотрели все возможные значения области определения для конкретной переменной и не обнаружили решения, в котором использовался бы существующий набор назначений, то возвращаем `None`, что указывает на отсутствие решения. В результате по цепочке рекурсии будет выполнен возврат к точке, в которой могло быть принято другое предварительное присваивание.

3.2. Задача раскраски карты Австралии

Представьте, что у вас есть карта Австралии, на которой вы хотите разными цветами обозначить штаты/территории (мы будем называть те и другие регионами). Никакие две соседние области не должны быть окрашены в одинаковый цвет. Можно ли раскрасить регионы всего тремя разными цветами?

Ответ: да. Попробуйте сами (самый простой способ — напечатать карту Австралии с белым фоном). Мы, люди, можем быстро найти решение путем изучения

карты и небольшого количества проб и ошибок. На самом деле это тривиальная задача, которая отлично подойдет в качестве первой для нашей программы решения задач с ограничениями методом поиска с возвратом (рис. 3.2).

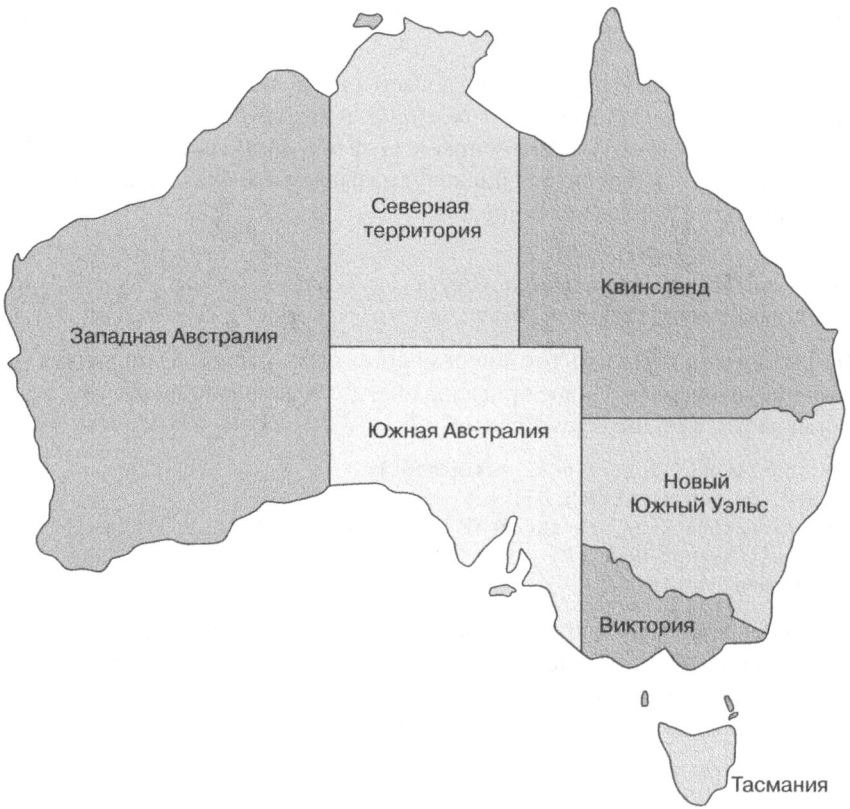


Рис. 3.2. В решении задачи раскраски карты Австралии никакие две смежные ее части не могут быть окрашены в один и тот же цвет

Чтобы смоделировать проблему как CSP, нужно определить переменные, области определения и ограничения. Переменными являются семь регионов Австралии (по крайней мере те семь, которыми мы ограничимся): Западная Австралия, Северная территория, Южная Австралия, Квинсленд, Новый Южный Уэльс, Виктория и Тасмания. В нашем CSP их можно представить как строки. Область определения каждой переменной — это три разных цвета, которые могут быть ей присвоены. (Мы используем красный, зеленый и синий.) Ограничения — сложный вопрос. Никакие две соседние области не могут быть окрашены в один и тот же цвет, поэтому ограничения будут зависеть от того, какие области граничат друг с другом. Мы задействуем так называемые двоичные ограничения — ограничения между двумя переменными. Каждые две области с общей границей

будут иметь двоичное ограничение, указывающее, что им нельзя присвоить один и тот же цвет.

Чтобы реализовать такие двоичные ограничения в коде, нужно создать подкласс класса `Constraint`. Конструктор подкласса `MapColoringConstraint` будет принимать две переменные — две области, имеющие общую границу. Его переопределенный метод `satisfied()` сначала проверит, присвоены ли этим двум областям значения (цвета) из области определения. Если нет, то ограничение считается тривиально выполненным до тех пор, пока цвета не будут присвоены. (Пока у одного из регионов нет цвета, конфликт невозможен.) Затем метод проверит, присвоен ли двум областям один и тот же цвет. Очевидно, что если цвета одинаковы, то существует конфликт, означающий: ограничение не выполняется.

Далее этот класс представлен во всей своей полноте (листинг 3.5). Сам по себе класс `MapColoringConstraint` не универсален с точки зрения аннотации типа, но он является подклассом параметризованной версии универсального класса `Constraint`, которая указывает, что переменные и области определения имеют тип `str`.

Листинг 3.5. map_coloring.py

```
from csp import Constraint, CSP
from typing import Dict, List, Optional

class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
        self.place2: str = place2

    def satisfied(self, assignment: Dict[str, str]) -> bool:
    # если какой-либо регион place отсутствует в присваивании,
    # то его цвета не могут привести к конфликту
    if self.place1 not in assignment or self.place2 not in assignment:
            return True
        # проверяем, не совпадает ли цвет, присвоенный place1,
        # с цветом, присвоенным place2
        return assignment[self.place1] != assignment[self.place2]
```

СОВЕТ

> Для вызова метода суперкласса иногда используется метод super(), но можно взять и имя самого класса, как в Constraint.__init__([place1, place2]). Это особенно полезно во время работы с множественным наследованием, так как позволяет знать, какой именно метод суперкласса вы вызываете.

Теперь, когда есть способ реализации ограничений между регионами, можно легко уточнить задачу о раскраске карты Австралии с помощью программы

решения методом CSP — нужно лишь заполнить области определения и переменные, а затем добавить ограничения (листинг 3.6).

Листинг 3.6. map_coloring.py (продолжение)

```
if __name__ == "__main__":
    variables: List[str] = ["Western Australia", "Northern Territory", "South
        Australia", "Queensland", "New South Wales", "Victoria", "Tasmania"]
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        domains[variable] = ["red", "green", "blue"]
    csp: CSP[str, str] = CSP(variables, domains)
    csp.add_constraint(MapColoringConstraint("Western Australia",
     "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Western Australia",
     "South Australia"))
    csp.add_constraint(MapColoringConstraint("South Australia",
     "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland",
     "Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Queensland", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("New South Wales",
     "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "South Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "New South Wales"))
    csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))
```

Наконец, вызываем `backtracking_search()` для поиска решения (листинг 3.7).

Листинг 3.7. map_coloring.py (продолжение)

```
solution: Optional[Dict[str, str]] = csp.backtracking_search()
if solution is None:
    print("No solution found!")
else:
    print(solution)
```

Правильное решение будет представлять собой список цветов, присвоенных регионам:

```
{'Western Australia': 'red', 'Northern Territory': 'green', 'South
    Australia': 'blue', 'Queensland': 'red', 'New South Wales': 'green',
    'Victoria': 'red', 'Tasmania': 'green'}
```

3.3. Задача восьми ферзей

Шахматная доска — это сетка размером 8 × 8 клеток. Ферзь — шахматная фигура, которая может перемещаться по шахматной доске на любое количество клеток по любой горизонтали, вертикали или диагонали. Если за один ход ферзь может пере-

меститься на клетку, на которой стоит другая фигура, не перепрыгивая ни через какую другую фигуру, то ферзь атакует эту фигуру. (Другими словами, если фигура находится в зоне прямой видимости ферзя, то она подвергается атаке.) Задача восьми ферзей состоит в том, как разместить восемь ферзей на шахматной доске таким образом, чтобы ни один из них не атаковал другого (рис. 3.3).

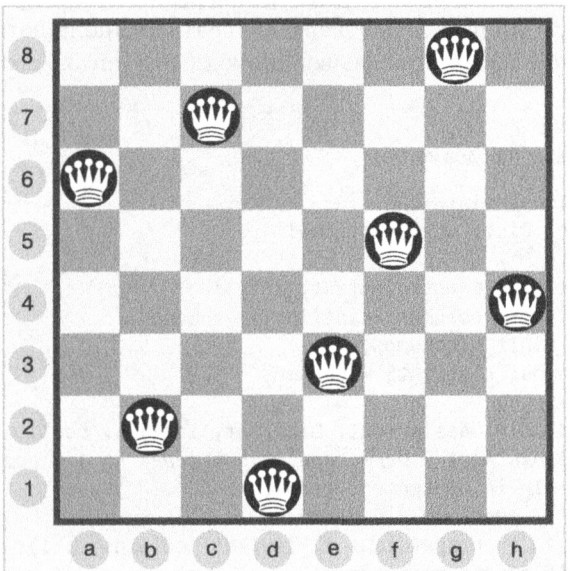


Рис. 3.3. В решении задачи восьми ферзей (существует много решений) никакие два ферзя не могут угрожать друг другу

Чтобы представить клетки на шахматной доске, присвоим каждой из них два целых числа, обозначающих горизонталь и вертикаль. Мы можем гарантировать, что никакая пара из восьми ферзей не находится на одной вертикали, просто присвоив им последовательно номера вертикалей с первого по восьмой. Переменными в задаче с ограничениями могут быть просто вертикали рассматриваемых ферзей. Области определения могут быть допустимыми горизонталями, тоже с номерами с первого по восьмой. В листинге 3.8 показан код, помещенный в конец нашего файла, где присвоены значения этим переменным и областям определения.

Листинг 3.8. queens.py

```
if __name__ == "__main__":
    columns: List[int] = [1, 2, 3, 4, 5, 6, 7, 8]
    rows: Dict[int, List[int]] = {}
    for column in columns:
        rows[column] = [1, 2, 3, 4, 5, 6, 7, 8]
    csp: CSP[int, int] = CSP(columns, rows)
```

Чтобы решить эту задачу, понадобится ограничение, которое проверяло бы, находятся ли любые два ферзя на одной горизонтали или диагонали. (Вначале всем им были присвоены разные номера вертикалей.) Проверка общей горизонтали тривиальна, но проверка общей диагонали требует небольшого применения математики. Если любые два ферзя находятся на одной диагонали, то разность между номерами их горизонталей будет равна разности между номерами их вертикалей. Заметили ли вы, где в `QueensConstraint` выполняются эти проверки? Обратите внимание на то, что следующий код (листинг 3.9) расположен вверху исходного файла.

Листинг 3.9. queens.py (продолжение)

```
from csp import Constraint, CSP
from typing import Dict, List, Optional

class QueensConstraint(Constraint[int, int]):
    def __init__(self, columns: List[int]) -> None:
        super().__init__(columns)
        self.columns: List[int] = columns

    def satisfied(self, assignment: Dict[int, int]) -> bool:
        # q1c = ферзь на 1-й вертикали, q1r = ферзь на 1-й горизонтали
        for q1c, q1r in assignment.items():
            # q2c = ферзь на 2-й вертикали
            for q2c in range(q1c + 1, len(self.columns) + 1):
                if q2c in assignment:
                    q2r: int = assignment[q2c] # q2r = ферзь на 2-й горизонтали
                    if q1r == q2r: # тот же ряд?
                        return False
                    if abs(q1r - q2r) == abs(q1c - q2c): # одна диагональ?
                        return False
        return True # нет конфликтов
```

Осталось только добавить ограничение и запустить поиск. Теперь вернемся в конец файла (листинг 3.10).

Листинг 3.10. queens.py (продолжение)

```
csp.add_constraint(QueensConstraint(columns))
solution: Optional[Dict[int, int]] = csp.backtracking_search()
if solution is None:
    print("No solution found!")
else:
    print(solution)
```

Обратите внимание на то, что мы довольно легко приспособили схему решения задач с ограничениями, созданную для раскраски карты, к совершенно другому

типу задач. В этом и заключается эффективность написания обобщенного кода! Алгоритмы должны быть реализованы настолько широко, насколько это возможно, если только не потребуется специализация для оптимизации производительности конкретного приложения.

Правильное решение присваивает каждому ферзю вертикаль и горизонталь:

```
{1: 1, 2: 5, 3: 8, 4: 6, 5: 3, 6: 7, 7: 2, 8: 4}
```

3.4. Поиск слова

Головоломка «Поиск слова» — это сетка букв со скрытыми словами, расположенными по строкам, столбцам и диагоналям. Разгадывая ее, игрок пытается найти скрытые слова, внимательно рассматривая сетку. Поиск мест для размещения слов таким образом, чтобы все они помещались в сетку, является своего рода задачей с ограничениями. Здесь переменные — это слова, а области определения — возможные их положения (рис. 3.4).

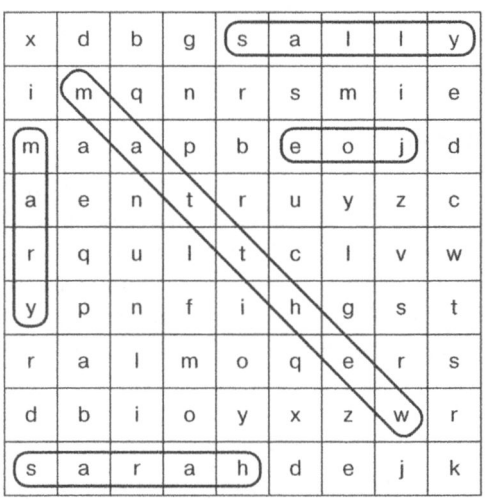


Рис. 3.4. Классическая головоломка по поиску слов, которую можно найти в детской книге головоломок

Из соображения целесообразности поиск слов не будет включать в себя перекрывающиеся слова. Вы можете его улучшить, чтобы учесть перекрывающиеся слова.

Сетка данной задачи не очень отличается от лабиринтов, описанных в главе 2 (листинг 3.11). Некоторые из представленных далее типов данных должны выглядеть знакомо.

Листинг 3.11. word_search.py

```
from typing import NamedTuple, List, Dict, Optional
from random import choice
from string import ascii_uppercase
from csp import CSP, Constraint

Grid = List[List[str]] # вводим псевдоним для сеток

class GridLocation(NamedTuple):
    row: int
    column: int
```

Вначале заполним сетку буквами английского алфавита (`ascii_uppercase`). Нам также понадобится функция для отображения сетки (листинг 3.12).

Листинг 3.12. word_search.py (продолжение)

```
def generate_grid(rows: int, columns: int) -> Grid:
    # инициализируем сетку случайными буквами
    return [[choice(ascii_uppercase) for c in range(columns)]
for r in range(rows)]

def display_grid(grid: Grid) -> None:
    for row in grid:
        print("".join(row))
```

Чтобы выяснить, где можно расположить слова в сетке, сгенерируем их области определения. Область определения слова — это список списков возможных положений всех его букв (`List[List[GridLocation]]`). Однако слова не могут располагаться где попало. Они должны находиться в пределах строки, столбца или диагонали в пределах сетки. Иначе говоря, они не должны выходить за пределы сетки. Цель функции `generate_domain()` — создание таких списков для каждого слова (листинг 3.13).

Листинг 3.13. word_search.py (продолжение)

```
def generate_domain(word: str, grid: Grid) -> List[List[GridLocation]]:
    domain: List[List[GridLocation]] = []
    height: int = len(grid)
    width: int = len(grid[0])
    length: int = len(word)
    for row in range(height):
        for col in range(width):
            columns: range = range(col, col + length + 1)
            rows: range = range(row, row + length + 1)
            if col + length <= width:
                # слева направо
                domain.append([GridLocation(row, c) for c in columns])
                # по диагонали снизу направо
```

```
                if row + length <= height:
                    domain.append([GridLocation(r, col + (r - row))
                      for r in rows])
            if row + length <= height:
                # сверху вниз
                domain.append([GridLocation(r, col) for r in rows])
                # по диагонали снизу налево
                if col - length >= 0:
                    domain.append([GridLocation(r, col - (r - row))
                      for r in rows])
    return domain
```

Для определения диапазона потенциальных положений слова (вдоль строки, столбца или по диагонали) списочные выражения преобразуют диапазон в список `GridLocation` с помощью конструктора этого класса. Поскольку для каждого слова `generate_domain()` перебирает все ячейки сетки от верхней левой до нижней правой, требуется большое количество вычислений. Можете ли вы придумать способ сделать это более эффективно? Что, если одновременно просматривать все слова одинаковой длины внутри цикла?

Чтобы проверить, является ли потенциальное решение допустимым, мы должны реализовать пользовательское ограничение для поиска слова. Метод `satisfied()` в `WordSearchConstraint` просто проверяет, совпадают ли какие-то положения, предложенные для одного слова, с положением, предложенным для другого слова. Это делается с помощью `set`. Преобразование `list` в `set` исключит все дубликаты. Если в `set`, преобразованном из `list`, окажется меньше элементов, чем было в исходном `list`, это означает, что исходный `list` содержал несколько дубликатов. Чтобы подготовить данные для этой проверки, воспользуемся несколько более сложным генератором списков, чтобы объединить несколько подсписков положений каждого слова в присваивании в один большой список положений (листинг 3.14).

Листинг 3.14. word_search.py (продолжение)

```
class WordSearchConstraint(Constraint[str, List[GridLocation]]):
    def __init__(self, words: List[str]) -> None:
        super().__init__(words)
        self.words: List[str] = words

    def satisfied(self, assignment: Dict[str, List[GridLocation]]) -> bool:
        # наличие дубликатов положений сетки означает наличие совпадения
        all_locations = [locs for values in assignment.values()
          for locs in values]
        return len(set(all_locations)) == len(all_locations)
```

Наконец-то мы готовы запустить программу. Для примера есть пять слов в сетке 9 × 9. Решение, которое получим, должно содержать соответствия между каждым словом и местами, где составляющие его буквы могут поместиться в сетке (листинг 3.15).

Листинг 3.15. word_search.py (продолжение)

```
if __name__ == "__main__":
    grid: Grid = generate_grid(9, 9)
    words: List[str] = ["MATTHEW", "JOE", "MARY", "SARAH", "SALLY"]
    locations: Dict[str, List[List[GridLocation]]] = {}
    for word in words:
        locations[word] = generate_domain(word, grid)
    csp: CSP[str, List[GridLocation]] = CSP(words, locations)
    csp.add_constraint(WordSearchConstraint(words))
    solution: Optional[Dict[str, List[GridLocation]]] =
      csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        for word, grid_locations in solution.items():
            # в половине случаев случайным выбором – задом наперед
            if choice([True, False]):
                grid_locations.reverse()
            for index, letter in enumerate(word):
                (row, col) = (grid_locations[index].row,
                              grid_locations[index].column)
                grid[row][col] = letter
        display_grid(grid)
```

В коде есть последний штрих, заполняющий сетку словами. Некоторые слова, выбранные случайным образом, располагаются задом наперед. Это корректно, поскольку данный пример не допускает наложения слов. Конечный результат должен выглядеть примерно так. Сможете ли вы найти здесь имена Matthew, Joe, Mary, Sarah, и Sally?

```
LWEHTTAMJ
MARYLISGO
DKOJYHAYE
IAJYHALAG
GYZJWRLGM
LLOTCAYIX
PEUTUSLKO
AJZYGIKDU
HSLZOFNNR
```

3.5. SEND + MORE = MONEY

SEND + MORE = MONEY – это криптоарифметическая головоломка, где нужно найти такие цифры, которые, будучи подставленными вместо букв, сделают математическое утверждение верным. Каждая буква в задаче представляет одну цифру от 0 до 9. Никакие две разные буквы не могут представлять одну и ту же цифру. Если буква повторяется, это означает, что цифра в решении также повторяется.

Чтобы проще было решить эту задачку вручную, стоит выстроить слова в столбик:

```
  SEND
 +MORE
=MONEY
```

Задача легко решается вручную, потребуется лишь немного алгебры и интуиции. Но довольно простая компьютерная программа поможет сделать это быстрее, используя множество возможных решений. Представим SEND + MORE = MONEY как задачу с ограничениями (листинг 3.16).

Листинг 3.16. send_more_money.py

```
from csp import Constraint, CSP
from typing import Dict, List, Optional

class SendMoreMoneyConstraint(Constraint[str, int]):
    def __init__(self, letters: List[str]) -> None:
        super().__init__(letters)
        self.letters: List[str] = letters

    def satisfied(self, assignment: Dict[str, int]) -> bool:
        # если есть повторяющиеся значения, то решение не подходит
        if len(set(assignment.values())) < len(assignment):
            return False

    # если все переменные назначены, проверяем, правильна ли сумма
    if len(assignment) == len(self.letters):
        s: int = assignment["S"]
        e: int = assignment["E"]
        n: int = assignment["N"]
        d: int = assignment["D"]
        m: int = assignment["M"]
        o: int = assignment["O"]
        r: int = assignment["R"]
        y: int = assignment["Y"]
        send: int = s * 1000 + e * 100 + n * 10 + d
        more: int = m * 1000 + o * 100 + r * 10 + e
        money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
        return send + more == money
    return True # нет противоречий
```

Метод `SendMoreMoneyConstraint` `satisfied()` выполняет несколько действий. Во-первых, он проверяет, соответствуют ли разные буквы одинаковым цифрам. Если это так, то решение неверно и метод возвращает `False`. Затем метод проверяет, всем ли буквам назначены цифры. Если это так, то он проверяет, является ли формула SEND + MORE = MONEY корректной с данным присваиванием. Если да, то решение найдено и метод возвращает `True`. В противном случае возвращается `False`. Наконец, если еще не всем буквам присвоены цифры, то возвращается `True`.

Это сделано для того, чтобы после получения частичного решения работа продолжалась.

Попробуем это запустить (листинг 3.17).

Листинг 3.17. send_more_money.py (продолжение)

```
if __name__ == "__main__":
    letters: List[str] = ["S", "E", "N", "D", "M", "O", "R", "Y"]
    possible_digits: Dict[str, List[int]] = {}
    for letter in letters:
        possible_digits[letter] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    possible_digits["M"] = [1] # мы не принимаем ответы, начинающиеся с 0
    csp: CSP[str, int] = CSP(letters, possible_digits)
    csp.add_constraint(SendMoreMoneyConstraint(letters))
    solution: Optional[Dict[str, int]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

Вы, вероятно, заметили, что мы предварительно задали значение для буквы M. Это сделано для того, чтобы исключить ответ, в котором M соответствует 0, потому что, если подумать, ограничение не учитывает, что число не может начинаться с нуля. Но попробуйте выполнить программу без этого заранее назначенного ответа.

Решение должно выглядеть примерно так:

```
{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}
```

3.6. Размещение элементов на печатной плате

Производитель должен установить прямоугольные микросхемы на прямоугольную плату. По сути, эта задача сводится к вопросу: «Как разместить все прямоугольники разных размеров внутри другого прямоугольника таким образом, чтобы они плотно прилегали друг к другу?» Решить эту задачу можно, применив схему решения задач с ограничениями (рис. 3.5).

Задача размещения элементов на печатной плате похожа на задачу поиска слов. Только вместо $1 \times N$ прямоугольников (слов) в ней представлены $M \times N$ прямоугольников. Как и в задаче поиска слов, они не могут перекрываться. Прямоугольники нельзя размещать по диагонали, так что в этом смысле задача даже проще, чем поиск слов.

Попробуйте самостоятельно переписать решение для поиска слов, чтобы приспособить его к задаче размещения элементов на печатной плате. Можете повторно использовать бóльшую часть кода, включая код для построения сетки.

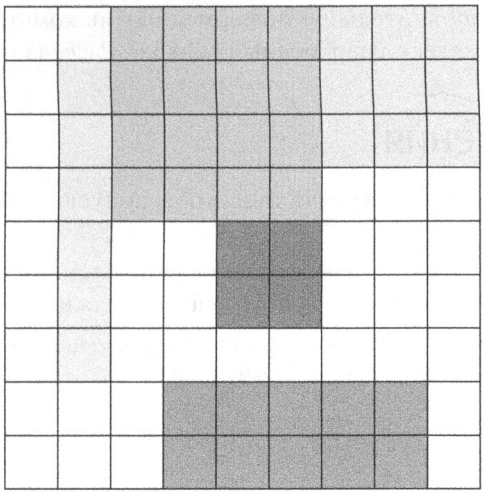


Рис. 3.5. Задача размещения элементов на печатной плате очень похожа на задачу поиска слова, но прямоугольники имеют разную ширину

3.7. Реальные приложения

Как упоминалось во введении к этой главе, программы для решения задач с ограничениями обычно применяются при планировании. На встрече должны присутствовать несколько человек, и они являются переменными. Их области определения состоят из отрезков свободного времени в ежедневниках. Ограничения могут включать в себя то, какие комбинации людей требуются на встрече.

Системы решения задач с ограничениями используются также при планировании движения. Представьте себе руку робота, которая должна помещаться внутри трубы. У нее есть ограничения (стенки трубы), переменные (суставы) и области определения (возможные движения суставов).

Такие приложения существуют и в области вычислительной биологии. Вы можете представить себе ограничения между молекулами, необходимыми для химической реакции. И конечно же, как и в случае с искусственным интеллектом, такие системы применяются в играх. Одним из упражнений является написание системы решения судоку, но многие логические головоломки могут быть решены как задачи с ограничениями.

В этой главе мы построили простую систему поиска с возвратами, поиска в глубину и схему решения задач. Но ее можно значительно улучшить, добавив эвристику (помните A*?) — интуитивную функцию, способную помочь при поиске. Еще одним эффективным способом разработки реальных приложений является более новая методика, чем поиск с возвратом, известная как *распространение ограничений (constraint propagation)*.

Для получения дополнительной информации ознакомьтесь с главой 6 книги «Искусственный интеллект: современный подход» Рассела и Норвига.

3.8. Упражнения

1. Измените `WordSearchConstraint` так, чтобы допускались перекрывающиеся буквы.
2. Создайте систему решения задачи размещения элементов на печатной плате, описанной в разделе 3.6, если вы этого еще не сделали.
3. Создайте программу, которая может решать судоку, используя систему решения задач с ограничениями, описанную в этой главе.

4 Графовые задачи

Граф — это абстрактная математическая конструкция, которая применяется для моделирования реальной задачи путем ее разделения на множество связанных узлов. Каждый такой узел мы будем называть *вершиной*, а каждое соединение — *ребром*. Так, карту метро можно рассматривать как граф, представляющий транспортную сеть. Каждая из ее точек — это станция, а каждая из линий — маршрут между двумя станциями. В терминологии графов мы будем называть станции вершинами, а маршруты — ребрами.

В чем здесь польза? Графы не только помогают абстрактно представить задачу, но и позволяют задействовать несколько широко распространенных и эффективных методов поиска и оптимизации. Так, в примере с метро предположим, что мы хотим узнать кратчайший маршрут от одной станции до другой. Или нужно получить минимальное количество перегонов, необходимых для соединения всех станций. Графовые алгоритмы, которые вы изучите в данной главе, помогут решить обе эти задачи. Кроме того, такие алгоритмы применимы не только к транспортным сетям, но и к любым сетевым задачам, возникающим, например, в компьютерных, распределительных и инженерных сетях. Задачи поиска и оптимизации во всех этих пространствах могут быть решены с помощью графовых алгоритмов.

4.1. Карта как граф

В этой главе мы будем работать не с графом станций метро, а с городами Соединенных Штатов Америки и возможными маршрутами между ними. На рис. 4.1 представлена карта континентальной части США с 15 крупнейшими муниципальными

статистическими районами (Metropolitan Statistical Area, MSA) страны, согласно оценкам Бюро переписи США[1].

Рис. 4.1. Карта 15 крупнейших муниципальных статистических районов США

Известный предприниматель Илон Маск предложил построить высокоскоростную транспортную сеть, состоящую из капсул, перемещающихся в герметичных трубах. Маск утверждает, что капсулы будут передвигаться со скоростью 700 миль в час и позволят создать экономически эффективный междугородный транспорт для расстояний до 900 миль[2]. Маск называет эту новую транспортную систему Hyperloop. В этой главе мы рассмотрим классические графовые задачи в контексте построения такой транспортной сети.

Сначала Маск предложил идею Hyperloop для соединения Лос-Анджелеса и Сан-Франциско. Если бы потребовалось построить национальную сеть Hyperloop, то было бы целесообразно соединить крупнейшие мегаполисы Америки. На рис. 4.2 удалены границы штатов, которые были обозначены на рис. 4.1. Кроме того, каждый муниципальный статистический район связан с несколькими соседними. Это соседи не всегда являются ближайшими соседними MSA что делает граф намного интереснее..

На рис. 4.2 показан граф с вершинами, соответствующими 15 крупнейшим MSA Соединенных Штатов, и ребрами, обозначающими потенциальные маршруты Hyperloop между городами. Маршруты были выбраны в иллюстративных целях. Конечно, в состав сети Hyperloop могут входить и другие потенциальные маршруты.

---

[1] Данные предоставлены американским Бюро переписи США (American Fact Finder), https://factfinder.census.gov/.

[2] *Musk E.* Hyperloop Alpha, http://mng.bz/chmu.

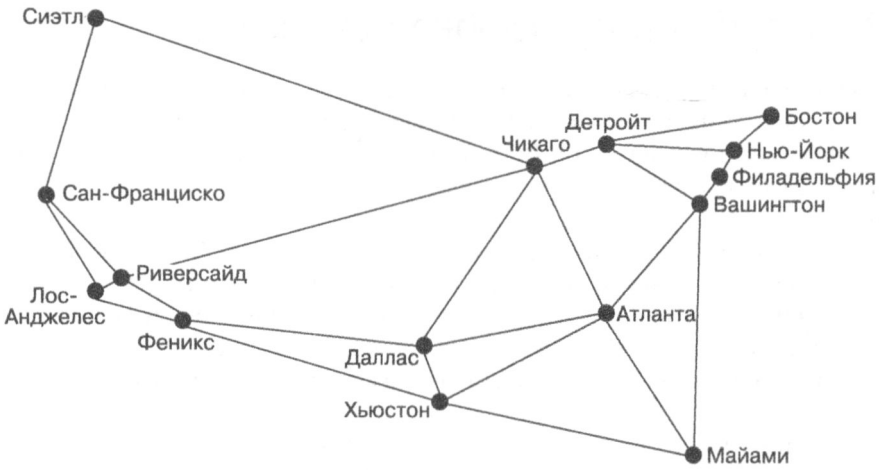


Рис. 4.2. Граф, в котором вершины представляют 15 крупнейших муниципальных статистических районов США, а ребра — потенциальные маршруты Hyperloop между ними

Такое абстрактное представление реальной задачи подчеркивает эффективность графов. Благодаря этой абстракции можно игнорировать географию Соединенных Штатов и сосредоточиться на изучении потенциальной сети Hyperloop только в контексте соединения городов. В сущности, мы можем представить задачу в виде любого графа, если только его ребра остаются неизменными. Например, на рис. 4.3 изменено положение Майами. Граф, изображенный здесь, будучи абстрактным представлением, позволяет решать те же фундаментальные вычислительные задачи, что и граф на рис. 4.2, несмотря на то что Майами на нем находится не там, где мы ожидаем. Но из соображений здравого смысла будем придерживаться представления, приведенного на рис. 4.2.

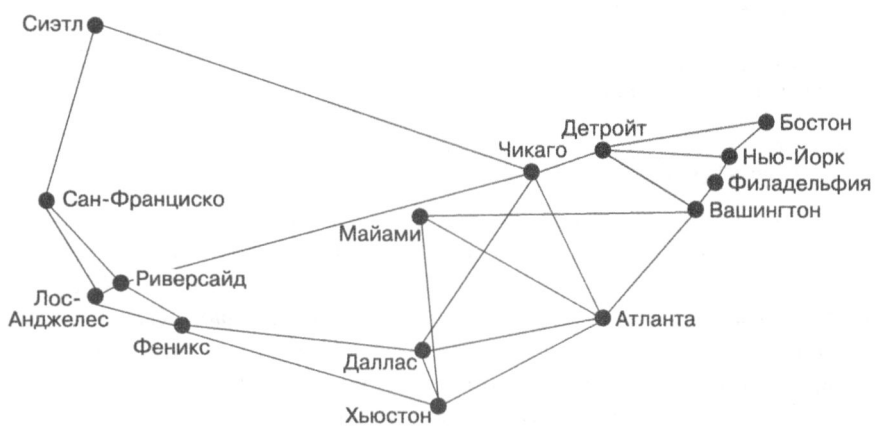


Рис. 4.3. График, эквивалентный показанному на рис. 4.2, с измененным положением Майами

4.2. Построение графовой структуры

Python позволяет программировать в самых разных стилях. Но по своей сути это объектно-ориентированный язык. В данном разделе мы построим два типа графов: невзвешенный и взвешенный. Во взвешенных графах, которые будут описаны позже, с каждым ребром ассоциируется вес — определенное число, такое как длина в нашем примере.

Мы используем модель наследования, которая является фундаментальной для объектно-ориентированных иерархий классов Python, поэтому не станем дублировать усилия. Взвешенные классы в нашей модели данных будут подклассами их невзвешенных аналогов. Это позволит таким классам наследовать бóльшую часть функциональности с небольшими изменениями, отличающими взвешенный граф от невзвешенного.

Мы хотим, чтобы графовая структура была настолько гибкой, насколько возможно, и могла представлять как можно больше различных задач. Для достижения этой цели применим параметризацию, чтобы абстрагироваться от типа вершин. В итоге каждой вершине присвоим целочисленный индекс, который будет сохранен как универсальный тип, определяемый пользователем.

Начнем построение структуры с того, что определим класс `Edge`, который является простейшим механизмом в графовой структуре (листинг 4.1).

Листинг 4.1. edge.py

```
from __future__ import annotations
from dataclasses import dataclass

@dataclass
class Edge:
    u: int # вершина "откуда"
    v: int # вершина "куда"

    def reversed(self) -> Edge:
        return Edge(self.v, self.u)

    def __str__(self) -> str:
        return f"{self.u} -> {self.v}"
```

`Edge` определяется как связь между двумя вершинами, каждая из которых представлена целочисленным индексом. По соглашению `u` определяет первую вершину, а `v` — вторую. Другими словами, `u` — «откуда», а `v` — «куда». В этой главе мы будем работать только с ненаправленными графами (графами с ребрами, которые допускают перемещение в обоих направлениях), но в направленных графах, также известных как диграфы, ребра могут быть односторонними. Метод `reversed()` должен возвращать ребро `Edge`, допускающему перемещение в направлении, противоположном направлению ребра, к которому применяется этот метод.

ПРИМЕЧАНИЕ

В классе Edge используется новая функциональность, которая появилась в Python 3.7, — классы данных. Класс, помеченный декоратором @dataclass, выполняет рутинную работу, автоматически создавая метод __init__(), который создает экземпляры переменных экземпляра для любых переменных, объявленных с аннотациями типов в теле класса. Классы данных также могут автоматически создавать другие специальные методы класса. То, какие именно специальные методы создаются автоматически, настраивается с помощью декоратора. Подробнее о классах данных читайте в документации по Python (https://docs.python.org/3/library/dataclasses.html). Проще говоря, класс данных — это способ сэкономить на вводе букв.

Класс `Graph` играет в графе важную роль — связывает вершины с ребрами. Здесь мы также хотим, чтобы действительные типы вершин могли быть любыми, какие только пожелает пользователь структуры. Это позволяет применять ее для решения широкого круга задач, не создавая промежуточных структур данных, которые склеивали бы все воедино. Например, на графе, подобном графу для маршрутов Hyperloop, можно определить тип вершин как `str`, потому что мы будем использовать в качестве вершин строки наподобие New York и Los Angeles. Начнем с класса `Graph` (листинг 4.2).

Листинг 4.2. graph.py

```
from typing import TypeVar, Generic, List, Optional
from edge import Edge

V = TypeVar('V') # тип вершин графа

class Graph(Generic[V]):
    def __init__(self, vertices: List[V] = []) -> None:
        self._vertices: List[V] = vertices
        self._edges: List[List[Edge]] = [[] for _ in vertices]
```

Список `_vertices` — это сердце графа. Все вершины сохраняются в списке, а впоследствии мы будем ссылаться на них по их целочисленному индексу в этом списке. Сама вершина может быть сложным типом данных, но ее индекс всегда имеет тип `int`, с которым легко работать. На другом уровне, между графовыми алгоритмами и массивом `_vertices`, этот индекс позволяет получить две вершины с одним и тем же именем в одном графе (представьте себе граф, в котором вершинами являются города некоей страны, причем среди них несколько носят название «Спрингфилд»). Несмотря на одинаковые имена, они будут иметь разные целочисленные индексы.

Есть много способов реализации структуры данных графа, но самыми распространенными являются использование матрицы вершин и списков смежности. В матрице вершин каждая ячейка представляет собой пересечение двух вершин графа, и значение этой ячейки указывает на связь (или ее отсутствие) между этими

вершинами. В нашей структуре данных графа задействуются списки смежности. В таком представлении у каждой вершины есть список вершин, с которыми она связана. В нашем конкретном представлении применяется список списков ребер, поэтому для каждой вершины существует список ребер, которыми она связана с другими вершинами. Этот список списков называется `_edges`.

Далее представлена остальная часть класса `Graph` во всей своей полноте (листинг 4.3). Обратите внимание на использование коротких, в основном однострочных методов с подробными и понятными именами. Благодаря этому остальная часть класса становится в значительной степени очевидной, однако она снабжена краткими комментариями, чтобы не оставалось места для неправильной интерпретации.

Листинг 4.3. graph.py (продолжение)

```
@property
def vertex_count(self) -> int:
    return len(self._vertices) # Количество вершин

@property
def edge_count(self) -> int:
    return sum(map(len, self._edges)) # Количество ребер

# Добавляем вершину в граф и возвращаем ее индекс
def add_vertex(self, vertex: V) -> int:
    self._vertices.append(vertex)
    self._edges.append([]) # Добавляем пустой список для ребер
    return self.vertex_count - 1 # Возвращаем индекс по добавленным вершинам

# Это ненаправленный граф,
# поэтому мы всегда добавляем вершины в обоих направлениях
def add_edge(self, edge: Edge) -> None:
    self._edges[edge.u].append(edge)
    self._edges[edge.v].append(edge.reversed())

# Добавляем ребро, используя индексы вершин (удобный метод)
def add_edge_by_indices(self, u: int, v: int) -> None:
    edge: Edge = Edge(u, v)
    self.add_edge(edge)

# Добавляем ребро, просматривая индексы вершин (удобный метод)
def add_edge_by_vertices(self, first: V, second: V) -> None:
    u: int = self._vertices.index(first)
    v: int = self._vertices.index(second)
    self.add_edge_by_indices(u, v)

# Поиск вершины по индексу
def vertex_at(self, index: int) -> V:
    return self._vertices[index]
```

```
# Поиск индекса вершины в графе
def index_of(self, vertex: V) -> int:
    return self._vertices.index(vertex)

# Поиск вершин, с которыми связана вершина с заданным индексом
def neighbors_for_index(self, index: int) -> List[V]:
    return list(map(self.vertex_at, [e.v for e in self._edges[index]]))

# Поиск индекса вершины; возвращает ее соседей (удобный метод)
def neighbors_for_vertex(self, vertex: V) -> List[V]:
    return self.neighbors_for_index(self.index_of(vertex))

# Возвращает все ребра, связанные с вершиной, имеющей заданный индекс
def edges_for_index(self, index: int) -> List[Edge]:
    return self._edges[index]

# Поиск индекса вершины; возвращает ее ребра (удобный метод)
def edges_for_vertex(self, vertex: V) -> List[Edge]:
    return self.edges_for_index(self.index_of(vertex))

# Упрощенный красивый вывод графа
def __str__(self) -> str:
    desc: str = ""
    for i in range(self.vertex_count):
        desc += f"{self.vertex_at(i)} -> {self.neighbors_for_index(i)}\n"
    return desc
```

Давайте ненадолго вернемся назад и посмотрим, почему большинство методов этого класса существует в двух версиях. Из определения класса мы знаем, что список `_vertices` содержит элементы типа `V`, который может быть любым классом Python. Таким образом, у нас есть вершины типа `V`, которые хранятся в списке `_vertices`. Но если мы хотим получить эти вершины и впоследствии ими манипулировать, то нужно знать, где они хранятся в данном списке. Следовательно, с каждой вершиной в этом массиве связан индекс (целое число). Если индекс вершины неизвестен, то его нужно найти, просматривая `_vertices`. Именно для этого нужны две версии каждого метода: одна оперирует внутренними индексами, другая — самим `V`. Методы, которые работают с `V`, ищут соответствующие индексы и вызывают основанную на индексе функцию. Поэтому их можно считать удобными.

Назначение большинства функций очевидно, но `neighbors_for_index()` заслуживает небольшого разъяснения. Эта функция возвращает соседей данной вершины. Соседи вершины — это все вершины, которые напрямую связаны с ней посредством ребер. Например, на рис. 4.2 Нью-Йорк и Вашингтон являются единственными соседями Филадельфии. Чтобы найти соседей вершины, нужно перебрать концы (значения `v`) всех выходящих из нее ребер:

```
def neighbors_for_index(self, index: int) -> List[V]:
    return list(map(self.vertex_at, [e.v for e in self._edges[index]]))
```

`_edges [index]` — это список смежности, то есть список ребер, посредством которых рассматриваемая вершина связана с другими вершинами. В списковом включении, передаваемом функции `map()`, `e` означает одно конкретное ребро, а `e.v` — индекс соседней вершины, с которой соединено данное ребро. `map()` возвращает все вершины, а не только их индексы, потому что `map()` применяет метод `vertex_at()` к каждому индексу `e.v`.

Еще один важный момент, на который стоит обратить внимание, — это то, как работает `add_edge()`. Сначала `add_edge()` добавляет ребро в список смежности вершины «откуда» (`u`), а затем добавляет обратную версию ребра в список смежности вершины «куда» (`v`). Второй шаг необходим, потому что этот граф ненаправленный. Мы хотим, чтобы каждое ребро было добавлено в обоих направлениях, это означает, что `u` будет соседней вершиной для `v`, а `v` — соседней для `u`. Ненаправленный граф можно представить себе как двунаправленный, если это поможет вам помнить, что каждое его ребро может быть пройдено в любом направлении:

```
def add_edge(self, edge: Edge) -> None:
    self._edges[edge.u].append(edge)
    self._edges[edge.v].append(edge.reversed())
```

Как уже упоминалось, в этой главе мы рассматриваем именно ненаправленные графы. Помимо того что графы могут быть направленными или ненаправленными, они могут быть также взвешенными или невзвешенными. Взвешенный граф — это такой граф, у которого с каждым из ребер связано некое сопоставимое значение, обычно числовое. В потенциальной сети Hyperloop мы могли бы представить вес ребер как расстояния между станциями. Однако сейчас будем иметь дело с невзвешенной версией графа. Невзвешенное ребро — это просто связь между двумя вершинами, следовательно, классы `Edge` и `Graph` являются невзвешенными. Другой способ показать это: о невзвешенном графе известно только то, какие вершины связаны, а о взвешенном — то, какие вершины связаны, и кое-какая дополнительная информация об этих связях.

4.2.1. Работа с Edge и Graph

Теперь, когда у нас есть конкретные реализации `Edge` и `Graph`, можем создать представление потенциальной сети Hyperloop. Вершины и ребра в `city_graph` соответствуют вершинам и ребрам, представленным на рис. 4.2. Используя параметризацию, мы можем указать, что вершины будут иметь тип `str` (`Graph[str]`). Другими словами, тип `str` заменяет переменную типа `V` (листинг 4.4).

Листинг 4.4. graph.py (продолжение)

```
if __name__ == "__main__":
    # тест простейшей графовой конструкции
    city_graph: Graph[str] = Graph(["Seattle", "San Francisco", "Los
     Angeles", "Riverside", "Phoenix", "Chicago", "Boston", "New York",
     "Atlanta", "Miami", "Dallas", "Houston", "Detroit", "Philadelphia",
     "Washington"])
```

```
city_graph.add_edge_by_vertices("Seattle", "Chicago")
city_graph.add_edge_by_vertices("Seattle", "San Francisco")
city_graph.add_edge_by_vertices("San Francisco", "Riverside")
city_graph.add_edge_by_vertices("San Francisco", "Los Angeles")
city_graph.add_edge_by_vertices("Los Angeles", "Riverside")
city_graph.add_edge_by_vertices("Los Angeles", "Phoenix")
city_graph.add_edge_by_vertices("Riverside", "Phoenix")
city_graph.add_edge_by_vertices("Riverside", "Chicago")
city_graph.add_edge_by_vertices("Phoenix", "Dallas")
city_graph.add_edge_by_vertices("Phoenix", "Houston")
city_graph.add_edge_by_vertices("Dallas", "Chicago")
city_graph.add_edge_by_vertices("Dallas", "Atlanta")
city_graph.add_edge_by_vertices("Dallas", "Houston")
city_graph.add_edge_by_vertices("Houston", "Atlanta")
city_graph.add_edge_by_vertices("Houston", "Miami")
city_graph.add_edge_by_vertices("Atlanta", "Chicago")
city_graph.add_edge_by_vertices("Atlanta", "Washington")
city_graph.add_edge_by_vertices("Atlanta", "Miami")
city_graph.add_edge_by_vertices("Miami", "Washington")
city_graph.add_edge_by_vertices("Chicago", "Detroit")
city_graph.add_edge_by_vertices("Detroit", "Boston")
city_graph.add_edge_by_vertices("Detroit", "Washington")
city_graph.add_edge_by_vertices("Detroit", "New York")
city_graph.add_edge_by_vertices("Boston", "New York")
city_graph.add_edge_by_vertices("New York", "Philadelphia")
city_graph.add_edge_by_vertices("Philadelphia", "Washington")
print(city_graph)
```

У `city_graph` есть вершины типа `str` — мы указываем для каждой название MSA, который она представляет. Последовательность, в которой добавляем ребра в `city_graph`, не имеет значения. Поскольку мы реализовали `__str__()` с красиво напечатанным описанием графа, теперь можно структурно распечатать (это настоящий термин!) граф. У вас должен получиться результат, подобный следующему:

```
Seattle -> ['Chicago', 'San Francisco']
San Francisco -> ['Seattle', 'Riverside', 'Los Angeles']
Los Angeles -> ['San Francisco', 'Riverside', 'Phoenix']
Riverside -> ['San Francisco', 'Los Angeles', 'Phoenix', 'Chicago']
Phoenix -> ['Los Angeles', 'Riverside', 'Dallas', 'Houston']
Chicago -> ['Seattle', 'Riverside', 'Dallas', 'Atlanta', 'Detroit']
Boston -> ['Detroit', 'New York']
New York -> ['Detroit', 'Boston', 'Philadelphia']
Atlanta -> ['Dallas', 'Houston', 'Chicago', 'Washington', 'Miami']
Miami -> ['Houston', 'Atlanta', 'Washington']
Dallas -> ['Phoenix', 'Chicago', 'Atlanta', 'Houston']
Houston -> ['Phoenix', 'Dallas', 'Atlanta', 'Miami']
Detroit -> ['Chicago', 'Boston', 'Washington', 'New York']
Philadelphia -> ['New York', 'Washington']
Washington -> ['Atlanta', 'Miami', 'Detroit', 'Philadelphia']
```

4.3. Поиск кратчайшего пути

Hyperloop — настолько быстрый транспорт, что для оптимизации времени в пути между станциями, вероятно, расстояние между ними имеет гораздо меньшее значение, чем то, сколько прыжков потребуется (сколько станций придется посетить), чтобы добраться от одной станции до другой. Каждая станция может означать задержку, поэтому, как и при полетах на самолете, чем меньше остановок, тем лучше.

В теории графов множество ребер, соединяющих две вершины, называется *путем*. Другими словами, путь — это способ перехода от одной вершины к другой. В контексте сети Hyperloop набор труб (ребер) представляет собой путь из одного города (вершины) в другой (другая вершина). Поиск оптимальных путей между вершинами является одной из наиболее распространенных задач, для которых используются графы.

Неформально мы также можем представить как путь список вершин, последовательно соединенных друг с другом ребрами. Такое описание — это, в сущности, другая сторона той же монеты. Это все равно что взять список ребер, выяснить, какие вершины они соединяют, сохранить список вершин и отбросить ребра. В следующем кратком примере мы найдем список вершин, соединяющий два города в сети Hyperloop.

4.3.1. Пересмотр алгоритма поиска в ширину

В невзвешенном графе поиск кратчайшего пути означает поиск пути с наименьшим количеством ребер между начальной и конечной вершинами. При построении сети Hyperloop, возможно, имеет смысл сначала подключить самые отдаленные города на густонаселенных побережьях. Возникает вопрос: какой путь между Бостоном и Майами является кратчайшим?

СОВЕТ

> В этом разделе предполагается, что вы прочитали главу 2. Прежде чем продолжить, убедитесь, что знакомы с алгоритмом поиска в ширину (BFS), описанным в ней.

К счастью, у нас уже есть алгоритм поиска кратчайших путей, и с его помощью мы можем ответить на этот вопрос. Поиск в ширину, описанный в главе 2, подходит для графов так же, как и для лабиринтов. В сущности, лабиринты, с которыми мы работали в главе 2, на самом деле являются графами. Вершины — это точки лабиринта, а ребра — ходы, которые можно сделать, чтобы попасть из одной точки в другую. В невзвешенном графе поиск в ширину найдет кратчайший путь между любыми двумя вершинами.

Мы можем повторно использовать реализацию поиска в ширину, представленную в главе 2, для работы с `Graph`. Да что там, ее можно применить вообще без изменений. Вот что значит параметризованный код!

Напомню, что функции `bfs()`, описанной в главе 2, требуются три параметра: начальное состояние, `Callable` (объект, похожий на функцию чтения) для проверки того, достигнута ли цель, и `Callable` для поиска состояний — наследников данного состояния. Начальным состоянием будет вершина, представленная строкой `"Boston"`, целевым тестом — лямбда-функция, которая проверяет, является ли данная вершина вершиной `"Miami"`. Наконец, вершины-наследники могут быть сгенерированы с помощью метода `neighbors_for_vertex()` из класса `Graph`.

Учитывая этот план, мы можем добавить соответствующий код в конец основного раздела `graph.py`, чтобы найти на `city_graph` кратчайший маршрут между Бостоном и Майами.

ПРИМЕЧАНИЕ

В листинге 4.5 bfs, Node и node_to_path импортируются из модуля generic_search, принадлежащего пакету Chapter2. Для этого в путь поиска Python ('..') добавляется родительский каталог graph.py. Это работает, поскольку структура кода в репозитории книги такова, что каждой главе соответствует отдельный каталог, так что в нашей структуре каталогов есть примерно такие пути: Book ▸ Chapter2 ▸ generic_search.py и Book ▸ Chapter4 ▸ graph.py. Если у вас структура каталогов существенно иная, то нужно будет найти способ добавить generic_search.py к своему пути и, возможно, изменить оператор импорта. В худшем случае можете просто скопировать файл generic_search.py в тот же каталог, в котором содержится graph.py, и изменить оператор импорта на from generic_search import bfs, Node, node_to_path.

Листинг 4.5. graph.py (продолжение)

```
# Повторное использование BFS из главы 2 для city_graph
import sys
sys.path.insert(0, '..') # так мы можем получить доступ к пакету Chapter2
                         # в родительском каталоге

from Chapter2.generic_search import bfs, Node, node_to_path

bfs_result: Optional[Node[V]] = bfs("Boston", lambda x: x == "Miami",
    city_graph.neighbors_for_vertex)
if bfs_result is None:
    print("No solution found using breadth-first search!")
else:
    path: List[V] = node_to_path(bfs_result)
    print("Path from Boston to Miami:")
    print(path)
```

Результат должен выглядеть примерно так:

```
Path from Boston to Miami:
['Boston', 'Detroit', 'Washington', 'Miami']
```

Путь Бостон — Детройт — Вашингтон — Майами, состоящий из трех ребер, — это самый короткий маршрут между Бостоном и Майами с точки зрения количества ребер (рис. 4.4).

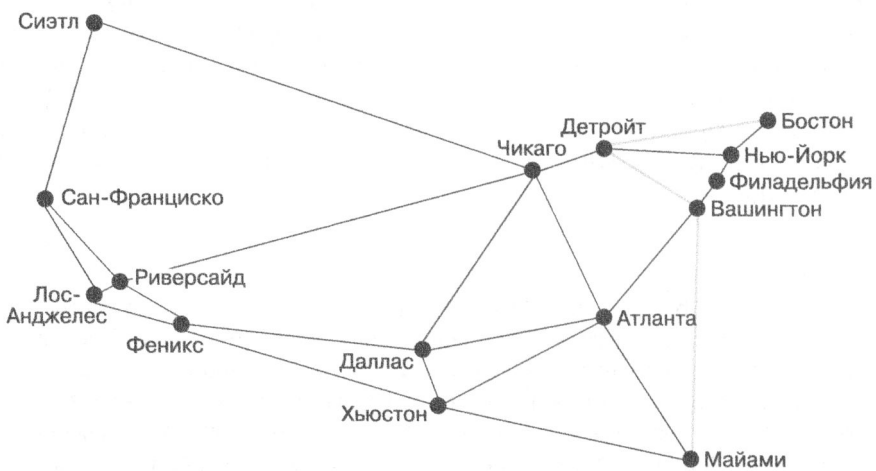


Рис. 4.4. Здесь выделен кратчайший маршрут между Бостоном и Майами с точки зрения количества ребер

4.4. Минимизация затрат на построение сети

Предположим, что мы хотим подключить к сети Hyperloop все 15 крупнейших MSA. Наша цель — минимизировать затраты на развертывание сети, что означает прокладку минимального количества трасс. Тогда возникает вопрос: «Как соединить все MSA, используя минимальное количество трасс?»

4.4.1. Работа с весами

Чтобы понять, сколько трасс может потребоваться для конкретного ребра, нам нужно знать расстояние, которое ему соответствует. Это дает возможность заново ввести понятие весов. В сети Hyperloop вес ребра — это расстояние между двумя MSA, которые оно соединяет. Рисунок 4.5 аналогичен рис. 4.2, за исключением того, что на нем указан вес каждого ребра, представляющий собой расстояние в милях между двумя вершинами, которые оно соединяет.

Для обработки весов понадобятся `WeightedEdge` (подкласс `Edge`) и `WeightedGraph` (подкласс `Graph`). С каждым `WeightedEdge` будет связано значение типа `float`.

Алгоритму Ярника, о котором мы вскоре расскажем, требуется возможность сравнивать ребра между собой, чтобы выбрать из них ребро с меньшим весом. Это легко реализовать посредством числовых весов (листинг 4.6).

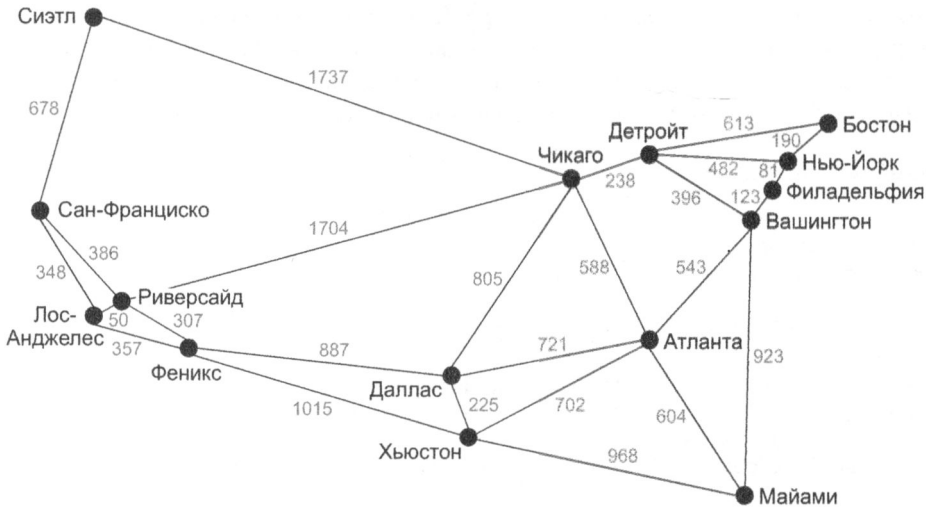


Рис. 4.5. Взвешенный граф с 15 самыми большими MSA Соединенных Штатов, вес каждого ребра представляет собой расстояние между двумя MSA в милях

Листинг 4.6. weighted_edge.py

```
from __future__ import annotations
from dataclasses import dataclass
from edge import Edge

@dataclass
class WeightedEdge(Edge):
    weight: float

    def reversed(self) -> WeightedEdge:
        return WeightedEdge(self.v, self.u, self.weight)

    # так можно упорядочить ребра по весу и найти ребро с минимальным весом
    def __lt__(self, other: WeightedEdge) -> bool:
        return self.weight < other.weight

    def __str__(self) -> str:
        return f"{self.u} {self.weight}> {self.v}"
```

Реализация `WeightedEdge` не сильно отличается от `Edge`. Разница только в том, что добавлено свойство `weight` и оператор `<` реализован с помощью `__lt__()`, благодаря чему два объекта `WeightedEdge` можно сравнивать. Оператор `<` просматривает только веса ребер, а не унаследованные свойства `u` и `v`, так как алгоритм Ярника занимается поиском наименьшего по весу ребра.

Класс `WeightedGraph` наследует бóльшую часть своей функциональности от `Graph`. Помимо этого, у данного класса есть методы инициализации и удобные

методы сложения объектов `WeightedEdge`. К тому же в нем реализована собственная версия `__str__()`. Появился также новый метод `neighbors_for_index_with_weights()`, который возвращает не только все соседние вершины, но и веса ведущих к ним ребер. Этот метод будет полезен в новой версии `__str__()` (листинг 4.7).

Листинг 4.7. weighted_graph.py

```
from typing import TypeVar, Generic, List, Tuple
from graph import Graph
from weighted_edge import WeightedEdge

V = TypeVar('V') # тип вершин в графе

class WeightedGraph(Generic[V], Graph[V]):
    def __init__(self, vertices: List[V] = []) -> None:
        self._vertices: List[V] = vertices
        self._edges: List[List[WeightedEdge]] = [[] for _ in vertices]

    def add_edge_by_indices(self, u: int, v: int, weight: float) -> None:
        edge: WeightedEdge = WeightedEdge(u, v, weight)
        self.add_edge(edge) # вызываем версию суперкласса

    def add_edge_by_vertices(self, first: V, second: V, weight: float)
      -> None:
        u: int = self._vertices.index(first)
        v: int = self._vertices.index(second)
        self.add_edge_by_indices(u, v, weight)

    def neighbors_for_index_with_weights(self, index: int)
      -> List[Tuple[V, float]]:
        distance_tuples: List[Tuple[V, float]] = []
        for edge in self.edges_for_index(index):
            distance_tuples.append((self.vertex_at(edge.v), edge.weight))
        return distance_tuples

    def __str__(self) -> str:
        desc: str = ""
        for i in range(self.vertex_count):
            desc += f"{self.vertex_at(i)} ->
              {self.neighbors_for_index_with_weights(i)}\n"
        return desc
```

Теперь можно наконец определить взвешенный граф. Взвешенный граф, с которым мы будем работать (см. рис. 4.5), называется `city_graph2` (листинг 4.8).

Листинг 4.8. weighted_graph.py (продолжение)

```
if __name__ == "__main__":
    city_graph2: WeightedGraph[str] = WeightedGraph(["Seattle", "San
     Francisco", "Los Angeles", "Riverside", "Phoenix", "Chicago",
```

```
    "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston",
    "Detroit", "Philadelphia", "Washington"])
city_graph2.add_edge_by_vertices("Seattle", "Chicago", 1737)
city_graph2.add_edge_by_vertices("Seattle", "San Francisco", 678)
city_graph2.add_edge_by_vertices("San Francisco", "Riverside", 386)
city_graph2.add_edge_by_vertices("San Francisco", "Los Angeles", 348)
city_graph2.add_edge_by_vertices("Los Angeles", "Riverside", 50)
city_graph2.add_edge_by_vertices("Los Angeles", "Phoenix", 357)
city_graph2.add_edge_by_vertices("Riverside", "Phoenix", 307)
city_graph2.add_edge_by_vertices("Riverside", "Chicago", 1704)
city_graph2.add_edge_by_vertices("Phoenix", "Dallas", 887)
city_graph2.add_edge_by_vertices("Phoenix", "Houston", 1015)
city_graph2.add_edge_by_vertices("Dallas", "Chicago", 805)
city_graph2.add_edge_by_vertices("Dallas", "Atlanta", 721)
city_graph2.add_edge_by_vertices("Dallas", "Houston", 225)
city_graph2.add_edge_by_vertices("Houston", "Atlanta", 702)
city_graph2.add_edge_by_vertices("Houston", "Miami", 968)
city_graph2.add_edge_by_vertices("Atlanta", "Chicago", 588)
city_graph2.add_edge_by_vertices("Atlanta", "Washington", 543)
city_graph2.add_edge_by_vertices("Atlanta", "Miami", 604)
city_graph2.add_edge_by_vertices("Miami", "Washington", 923)
city_graph2.add_edge_by_vertices("Chicago", "Detroit", 238)
city_graph2.add_edge_by_vertices("Detroit", "Boston", 613)
city_graph2.add_edge_by_vertices("Detroit", "Washington", 396)
city_graph2.add_edge_by_vertices("Detroit", "New York", 482)
city_graph2.add_edge_by_vertices("Boston", "New York", 190)
city_graph2.add_edge_by_vertices("New York", "Philadelphia", 81)
city_graph2.add_edge_by_vertices("Philadelphia", "Washington", 123)

print(city_graph2)
```

Поскольку в `WeightedGraph` реализован метод `__str__()`, мы можем структурно распечатать `city_graph2`. При выводе вы увидите и вершины, с которыми соединена данная вершина, и веса этих соединений:

```
Seattle -> [('Chicago', 1737), ('San Francisco', 678)]
San Francisco -> [('Seattle', 678), ('Riverside', 386), ('Los Angeles', 348)]
Los Angeles -> [('San Francisco', 348), ('Riverside', 50), ('Phoenix', 357)]
Riverside -> [('San Francisco', 386), ('Los Angeles', 50), ('Phoenix', 307),
  ('Chicago', 1704)]
Phoenix -> [('Los Angeles', 357), ('Riverside', 307), ('Dallas', 887),
  ('Houston', 1015)]
Chicago -> [('Seattle', 1737), ('Riverside', 1704), ('Dallas', 805),
    ('Atlanta', 588), ('Detroit', 238)]
Boston -> [('Detroit', 613), ('New York', 190)]
New York -> [('Detroit', 482), ('Boston', 190), ('Philadelphia', 81)]
Atlanta -> [('Dallas', 721), ('Houston', 702), ('Chicago', 588),
  ('Washington', 543), ('Miami', 604)]
Miami -> [('Houston', 968), ('Atlanta', 604), ('Washington', 923)]
Dallas -> [('Phoenix', 887), ('Chicago', 805), ('Atlanta', 721),
  ('Houston', 225)]
```

```
Houston -> [('Phoenix', 1015), ('Dallas', 225), ('Atlanta', 702),
  ('Miami', 968)]
Detroit -> [('Chicago', 238), ('Boston', 613), ('Washington', 396),
  ('New York', 482)]
Philadelphia -> [('New York', 81), ('Washington', 123)]
Washington -> [('Atlanta', 543), ('Miami', 923), ('Detroit', 396),
  ('Philadelphia', 123)]
```

4.4.2. Поиск минимального связующего дерева

Дерево — это особый вид графа, в котором между любыми двумя вершинами существует один и только один путь. Это подразумевает, что в дереве нет *циклов* (такие графы иногда называют *ациклическими*). Цикл можно представить как петлю: если возможно из начальной вершины пройти по всему графу, никогда не проходя повторно ни по одному из ребер, и вернуться к начальной вершине, то в этом графе есть цикл. Любой граф, не являющийся деревом, может им стать, если удалить из него некоторые ребра. На рис. 4.6 показано удаление ребер, позволяющее превратить граф в дерево.

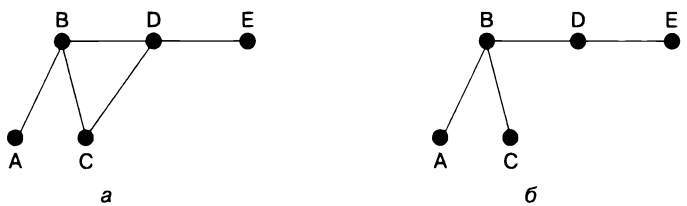


Рис. 4.6. На левом графе существует цикл между вершинами B, C и D, поэтому он не является деревом. На правом графе ребро, соединяющее C и D, удалено, так что этот граф — дерево

Связный граф — это граф, для которого существует способ добраться из любой вершины в любую другую вершину (все графы, которые мы рассматриваем в этой главе, являются связными). *Связующее дерево* — это дерево, которое соединяет все вершины графа. *Минимальное связующее дерево* — это дерево, которое соединяет все вершины во взвешенном графе и имеет минимальный общий вес по сравнению с другими связующими деревьями. Для каждого взвешенного графа можно найти минимальное связующее дерево.

Фуууух, слишком много терминологии! Дело в том, что найти минимальное связующее дерево — это то же самое, что найти способ связать все вершины во взвешенном графе с минимальным весом. При проектировании любой сети — транспортной, компьютерной и т. д. — возникает важная практическая проблема: как с минимальными затратами подключить к сети все узлы? Этими затратами могут быть провода, трассы, дороги или что-то еще. Например, для телефонной сети другой способ постановки этой задачи таков: «Какова минимальная длина кабеля, необходимого для подключения всех телефонов?»

Пересмотр очереди с приоритетом

Очереди с приоритетом были описаны в главе 2. Нам понадобится очередь с приоритетом для алгоритма Ярника. Можете импортировать класс `PriorityQueue` из пакета к главе 2 (подробнее об этом читайте в примечании, предшествующем листингу 4.5) или же скопировать класс в новый файл, чтобы использовать пакет для этой главы. Для полноты картины воссоздадим `PriorityQueue` из главы 2 здесь со специальными инструкциями `import`, которые предполагают, что этот класс будет помещен в отдельный файл (листинг 4.9).

Листинг 4.9. priority_queue.py

```
from typing import TypeVar, Generic, List
from heapq import heappush, heappop

T = TypeVar('T')

class PriorityQueue(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container

    # не true для пустого контейнера
    def push(self, item: T) -> None:
        heappush(self._container, item) # включить по приоритету

    def pop(self) -> T:
        return heappop(self._container) # исключить по приоритету

    def __repr__(self) -> str:
        return repr(self._container)
```

Вычисление общего веса взвешенного пути

Прежде чем разработать метод поиска минимального связующего дерева, создадим функцию, которую сможем использовать для проверки суммарного веса решения. Найденное минимальное связующее дерево будет представлять собой список взвешенных ребер, из которых оно состоит. Сначала мы определим `WeightedPath` как список `WeightedEdge`. Затем определим функцию `total_weight()`, которая принимает список `WeightedPath` и находит общий вес, получаемый в результате сложения всех весов ее ребер (листинг 4.10).

Листинг 4.10. mst.py

```
from typing import TypeVar, List, Optional
from weighted_graph import WeightedGraph
```

```
from weighted_edge import WeightedEdge
from priority_queue import PriorityQueue

V = TypeVar('V') # тип вершин в графе
WeightedPath = List[WeightedEdge] # псевдоним типа для путей

def total_weight(wp: WeightedPath) -> float:
    return sum([e.weight for e in wp])
```

Алгоритм Ярника

Алгоритм Ярника для поиска минимального связующего дерева решает задачу посредством деления графа на две части: вершины в формируемом минимальном связующем дереве и вершины, еще не входящие в минимальное связующее дерево. Алгоритм выполняет следующие шаги.

1. Выбрать произвольную вершину для включения в минимальное связующее дерево.
2. Найти ребро с наименьшим весом, соединяющее минимальное связующее дерево с вершинами, еще не входящими в минимальное связующее дерево.
3. Добавить вершину, расположенную на конце этого минимального ребра, к минимальному связующему дереву.
4. Повторять шаги 2 и 3, пока все вершины графа не будут включены в минимальное связующее дерево.

ПРИМЕЧАНИЕ

Алгоритм Ярника часто называют алгоритмом Прима. Два чешских математика, Отакар Борувка и Войцех Ярник, заинтересовавшись минимизацией затрат на прокладку электрических линий в конце 1920-х годов, придумали алгоритмы для решения задачи поиска минимального связующего дерева. Их алгоритмы были «переоткрыты» другими учеными несколько десятков лет спустя[1].

Для эффективного выполнения алгоритма Ярника используется очередь с приоритетом. Каждый раз, когда новая вершина добавляется в минимальное связующее дерево, все ее исходящие ребра, которые связываются с вершинами вне дерева, добавляются в очередь с приоритетом. Ребро с наименьшим весом всегда первым извлекается из очереди с приоритетом, и алгоритм продолжает выполняться до тех пор, пока очередь с приоритетом не опустеет. Это гарантирует, что

[1] *Durnová H.* Otakar Borůvka (1899–1995) and the Minimum Spanning Tree. — Institute of Mathematics of the Czech Academy of Sciences, 2006. http://mng.bz/O2vj.

ребра с наименьшим весом всегда будут первыми добавляться в дерево. Ребра, которые соединяются с вершинами, уже находящимися в дереве, игнорируются при извлечении из очереди.

Далее представлены код функции `mst()` — полная реализация алгоритма Ярника[1], а также вспомогательная функция для печати `WeightedPath` (листинг 4.11).

ПРЕДУПРЕЖДЕНИЕ

Алгоритм Ярника не всегда правильно работает для графов с направленными ребрами. Он не подействует и для несвязных графов.

Листинг 4.11. mst.py (продолжение)

```
def mst(wg: WeightedGraph[V], start: int = 0) -> Optional[WeightedPath]:
    if start > (wg.vertex_count - 1) or start < 0:
        return None
    result: WeightedPath = [] # содержит окончательное MST
    pq: PriorityQueue[WeightedEdge] = PriorityQueue()
    visited: [bool] = [False] * wg.vertex_count # здесь мы уже были

    def visit(index: int):
        visited[index] = True # пометить как прочитанное
        for edge in wg.edges_for_index(index):
            # добавляем все ребра отсюда в pq
            if not visited[edge.v]:
                pq.push(edge)

    visit(start) # первая вершина, с которой все начинается

    while not pq.empty: # продолжаем, пока остаются необработанные вершины
        edge = pq.pop()
        if visited[edge.v]:
            continue # никогда не просматриваем дважды
            # на данный момент это минимальный вес, так что добавляем в дерево
            result.append(edge)
            visit(edge.v)

        return result

    def print_weighted_path(wg: WeightedGraph, wp: WeightedPath) -> None:
        for edge in wp:
            print(f"{wg.vertex_at(edge.u)} {edge.weight}>
              {wg.vertex_ at(edge.v)}")
        print(f"Total eight: {total_weight(wp)}")
```

[1] *Седжвик Р., Уэйн К.* Алгоритмы на Java. 4-е изд. — М.: Вильямс, 2013.

Алгоритм возвращает необязательный объект `WeightedPath`, представляющий собой минимальное связующее дерево. Не имеет значения, с какой вершины начинается алгоритм (при условии, что граф является связным и ненаправленным), поэтому по умолчанию выбирается начальная вершина с индексом 0. Если значение `start` оказалось неверным, то `mst()` возвращает `None`.

```
result: WeightedPath = [] # содержит окончательное MST
pq: PriorityQueue[WeightedEdge] = PriorityQueue()
visited: [bool] = [False] * wg.vertex_count # здесь мы уже были
```

В итоге `result` будет содержать взвешенный путь, куда входит минимальное связующее дерево. Именно здесь добавляются объекты `WeightedEdge`, так как ребро с наименьшим весом извлекается из очереди и переводит нас к новой части графа. Алгоритм Ярника считается *жадным алгоритмом*, потому что всегда выбирает ребро с наименьшим весом. В `pq` хранятся новые обнаруженные ребра, и отсюда извлекается следующее ребро с наименьшим весом. В `visited` отслеживаются индексы вершин, которые мы уже просмотрели. Это можно сделать также с помощью `Set`, аналогичного `explored` в `bfs()`:

```
def visit(index: int):
    visited[index] = True # пометить как прочитанное
    for edge in wg.edges_for_index(index):
        if not visited[edge.v]:
            pq.push(edge)
```

`visit()` — это внутренняя вспомогательная функция, которая отмечает вершину как просмотренную и добавляет в `pq` все ее ребра, которые соединяются с еще не просмотренными вершинами. Обратите внимание на то, как легко модель списка смежности облегчает поиск ребер, принадлежащих определенной вершине:

```
visit(start) # первая вершина, с которой все начинается
```

Если граф несвязный, то не имеет значения, какая вершина рассматривается первой. Если несвязный граф состоит из разъединенных компонентов, то `mst()` вернет дерево, охватывающее тот *компонент*, которому принадлежит начальная вершина:

```
while not pq.empty: # продолжаем, пока остаются необработанные вершины
    edge = pq.pop()
    if visited[edge.v]:
        continue # никогда не просматриваем дважды
        # на данный момент это минимальный вес, так что добавляем в дерево
        result.append(edge)
        visit(edge.v)

    return result
```

Пока в очереди с приоритетом остаются ребра, мы извлекаем их оттуда и проверяем, ведут ли они к вершинам, которых еще нет в дереве. Поскольку очередь

с приоритетом является возрастающей, из нее сначала извлекаются ребра с наименьшим весом. Это гарантирует, что результат действительно имеет минимальный общий вес. Если ребро, извлеченное из очереди, не приводит к неисследованной вершине, то оно игнорируется. В противном случае, поскольку ребро имеет минимальный из всех видимых весов, оно добавляется к результирующему набору и исследуется новая вершина, к которой ведет это ребро. Когда неисследованных ребер не остается, возвращается результат.

Теперь мы наконец вернемся к задаче соединения всех 15 крупнейших MSA Соединенных Штатов в сеть Hyperloop минимальным количеством трасс. Маршрут, выполняющий эту задачу, является просто минимальным связующим деревом для `city_graph2`. Попробуем запустить `mst()` для `city_graph2` (листинг 4.12).

Листинг 4.12. mst.py (продолжение)

```
if __name__ == "__main__":
    city_graph2: WeightedGraph[str] = WeightedGraph(["Seattle", "San
     Francisco", "Los Angeles", "Riverside", "Phoenix", "Chicago",
     "Boston", "New York", "Atlanta", "Miami", "Dallas", "Houston",
     "Detroit", "Philadelphia", "Washington"])

    city_graph2.add_edge_by_vertices("Seattle", "Chicago", 1737)
    city_graph2.add_edge_by_vertices("Seattle", "San Francisco", 678)
    city_graph2.add_edge_by_vertices("San Francisco", "Riverside", 386)
    city_graph2.add_edge_by_vertices("San Francisco", "Los Angeles", 348)
    city_graph2.add_edge_by_vertices("Los Angeles", "Riverside", 50)
    city_graph2.add_edge_by_vertices("Los Angeles", "Phoenix", 357)
    city_graph2.add_edge_by_vertices("Riverside", "Phoenix", 307)
    city_graph2.add_edge_by_vertices("Riverside", "Chicago", 1704)
    city_graph2.add_edge_by_vertices("Phoenix", "Dallas", 887)
    city_graph2.add_edge_by_vertices("Phoenix", "Houston", 1015)
    city_graph2.add_edge_by_vertices("Dallas", "Chicago", 805)
    city_graph2.add_edge_by_vertices("Dallas", "Atlanta", 721)
    city_graph2.add_edge_by_vertices("Dallas", "Houston", 225)
    city_graph2.add_edge_by_vertices("Houston", "Atlanta", 702)
    city_graph2.add_edge_by_vertices("Houston", "Miami", 968)
    city_graph2.add_edge_by_vertices("Atlanta", "Chicago", 588)
    city_graph2.add_edge_by_vertices("Atlanta", "Washington", 543)
    city_graph2.add_edge_by_vertices("Atlanta", "Miami", 604)
    city_graph2.add_edge_by_vertices("Miami", "Washington", 923)
    city_graph2.add_edge_by_vertices("Chicago", "Detroit", 238)
    city_graph2.add_edge_by_vertices("Detroit", "Boston", 613)
    city_graph2.add_edge_by_vertices("Detroit", "Washington", 396)
    city_graph2.add_edge_by_vertices("Detroit", "New York", 482)
    city_graph2.add_edge_by_vertices("Boston", "New York", 190)
    city_graph2.add_edge_by_vertices("New York", "Philadelphia", 81)
    city_graph2.add_edge_by_vertices("Philadelphia", "Washington", 123)

    result: Optional[WeightedPath] = mst(city_graph2)
```

```
    if result is None:
        print("No solution found!")
    else:
        print_weighted_path(city_graph2, result)
```

Благодаря методу структурной печати `printWeightedPath()` минимальное связующее дерево легко читается:

```
Seattle 678> San Francisco
San Francisco 348> Los Angeles
Los Angeles 50> Riverside
Riverside 307> Phoenix
Phoenix 887> Dallas
Dallas 225> Houston
Houston 702> Atlanta
Atlanta 543> Washington
Washington 123> Philadelphia
Philadelphia 81> New York
New York 190> Boston
Washington 396> Detroit
Detroit 238> Chicago
Atlanta 604> Miami
Total Weight: 5372
```

Другими словами, это суммарно кратчайший набор ребер, который соединяет все MSA во взвешенном графе. Минимальная длина трассы, необходимой для соединения всех ребер, составляет 5372 мили. Минимальное связующее дерево показано на рис. 4.7.

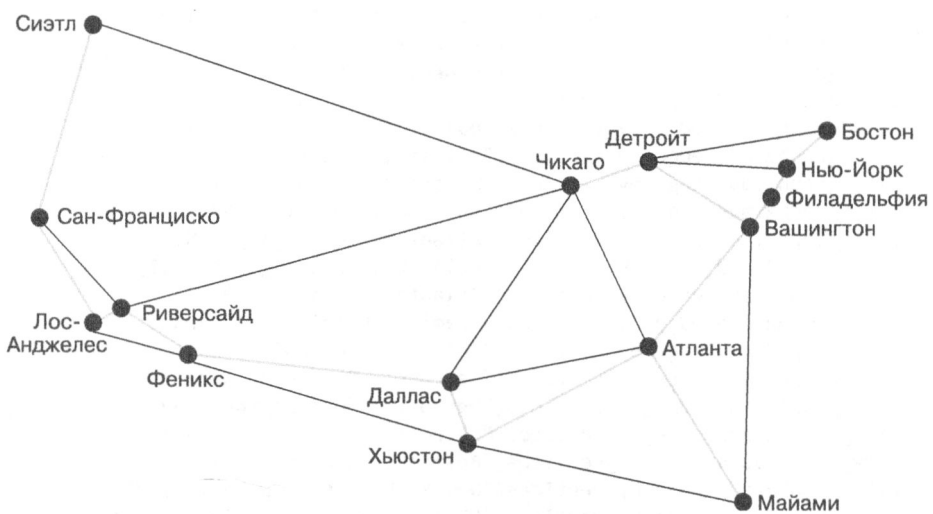


Рис. 4.7. Выделенные ребра образуют минимальное связующее дерево, которое соединяет все 15 MSA

4.5. Поиск кратчайших путей во взвешенном графе

Пока сеть Hyperloop находится в процессе создания, маловероятно, что ее разработчики настолько амбициозны, чтобы хотеть опутать ею сразу всю страну. Вместо этого они, скорее всего, захотят минимизировать затраты на прокладку трасс между ключевыми городами. Затраты на расширение сети до отдельных городов, очевидно, будут зависеть от того, с чего строители начнут работы.

Определение затрат для любого заданного начального города — пример задачи поиска кратчайшего пути из одного источника. Эта задача гласит: «Каков кратчайший путь с точки зрения общего веса ребра от некоторой вершины к любой другой вершине во взвешенном графе?»

4.5.1. Алгоритм Дейкстры

Алгоритм Дейкстры решает задачу поиска кратчайшего пути из одной вершины. Задается начальная вершина, и алгоритм возвращает путь с наименьшим весом к любой другой вершине во взвешенном графе. Он также возвращает минимальный общий вес для пути из начальной вершины к каждой из оставшихся. Алгоритм Дейкстры начинается из одной исходной вершины, а затем постоянно исследует ближайшие к ней вершины. По этой причине алгоритм Дейкстры, как и алгоритм Ярника, является жадным. Когда алгоритм Дейкстры исследует новую вершину, он проверяет, как далеко она находится от начальной вершины, и обновляет это значение, если находит более короткий путь. Подобно алгоритму поиска в ширину, он отслеживает, какие ребра ведут к каждой вершине.

Алгоритм Дейкстры выполняет такие шаги.

1. Добавить начальную вершину в очередь с приоритетом.
2. Извлечь из очереди с приоритетом ближайшую вершину (вначале это только исходная вершина) — назовем ее текущей.
3. Исследовать все соседние вершины, связанные с текущей. Если они ранее не были записаны или если ребро предлагает новый кратчайший путь, то для каждой из этих вершин записать расстояние до начальной вершины, указать ребро, соответствующее этому расстоянию, и добавить новую вершину в очередь с приоритетом.
4. Повторять шаги 2 и 3, пока очередь с приоритетом не опустеет.
5. Вернуть кратчайшее расстояние до каждой вершины от начальной и путь, позволяющий добраться до каждой из них.

Код алгоритма Дейкстры включает в себя `DijkstraNode` — простую структуру данных для отслеживания затрат, связанных с каждой исследованной вершиной, и сравнения вершин. Это похоже на класс `Node`, описанный в главе 2. Он также

включает в себя вспомогательные функции для преобразования возвращенного массива расстояний во что-то более простое, удобное для поиска по вершине и вычисления кратчайшего пути до конкретной конечной вершины из словаря путей, возвращаемого `dijkstra()`.

Чем долго рассуждать, лучше сразу приведем код алгоритма Дейкстры (листинг 4.13). Разберем его построчно.

Листинг 4.13. dijkstra.py

```
from __future__ import annotations
from typing import TypeVar, List, Optional, Tuple, Dict
from dataclasses import dataclass
from mst import WeightedPath, print_weighted_path
from weighted_graph import WeightedGraph
from weighted_edge import WeightedEdge
from priority_queue import PriorityQueue

V = TypeVar('V') # тип вершин в графе

@dataclass
class DijkstraNode:
    vertex: int
    distance: float

    def __lt__(self, other: DijkstraNode) -> bool:
        return self.distance < other.distance

    def __eq__(self, other: DijkstraNode) -> bool:
        return self.distance == other.distance

def dijkstra(wg: WeightedGraph[V], root: V) -> Tuple[List[Optional[float]],
        Dict[int, WeightedEdge]]:
    first: int = wg.index_of(root) # найти начальную вершину
    # вначале расстояния неизвестны
    distances: List[Optional[float]] = [None] * wg.vertex_count
    distances[first] = 0 # корневая вершина равна 0
    path_dict: Dict[int, WeightedEdge] = {} # как добраться до каждой вершины
    pq: PriorityQueue[DijkstraNode] = PriorityQueue()
    pq.push(DijkstraNode(first, 0))

    while not pq.empty:
        u: int = pq.pop().vertex # исследовать ближайшую вершину
        dist_u: float = distances[u] # это мы уже, должно быть, видели
        # рассмотреть все ребра и вершины для данной вершины
        for we in wg.edges_for_index(u):
            # старое расстояние до этой вершины
            dist_v: float = distances[we.v]
            # старого расстояния не существует или найден более короткий путь
            if dist_v is None or dist_v > we.weight + dist_u:
                # изменить расстояние до этой вершины
                distances[we.v] = we.weight + dist_u
```

```
                # заменить ребро на более короткий путь к этой вершине
                path_dict[we.v] = we
                # вскоре мы это проверим
                pq.push(DijkstraNode(we.v, we.weight + dist_u))

    return distances, path_dict

# Вспомогательная функция для удобного доступа к результатам
# алгоритма Дейкстры
def distance_array_to_vertex_dict(wg: WeightedGraph[V], distances:
        List[Optional[float]]) -> Dict[V, Optional[float]]:
    distance_dict: Dict[V, Optional[float]] = {}
    for i in range(len(distances)):
        distance_dict[wg.vertex_at(i)] = distances[i]
    return distance_dict

# Принимает словарь ребер, позволяющих достичь каждого узла,
# и возвращает список ребер от start до end
def path_dict_to_path(start: int, end: int, path_dict: Dict[int,
        WeightedEdge]) -> WeightedPath:
    if len(path_dict) == 0:
        return []
    edge_path: WeightedPath = []
    e: WeightedEdge = path_dict[end]
    edge_path.append(e)
    while e.u != start:
        e = path_dict[e.u]
        edge_path.append(e)
    return list(reversed(edge_path))
```

В первых строках `dijkstra()` используются структуры данных, с которыми вы уже знакомы, за исключением `distances` — заполнителя для расстояний до каждой вершины графа от `root`. Первоначально все эти расстояния равны `None`, потому что мы еще не знаем значения каждого из них, — именно для их определения мы применяем алгоритм Дейкстры!

```
def dijkstra(wg: WeightedGraph[V], root: V) -> Tuple[List[Optional[float]],
        Dict[int, WeightedEdge]]:
    first: int = wg.index_of(root) # найти начальную вершину
    # поначалу расстояния неизвестны
    distances: List[Optional[float]] = [None] * wg.vertex_count
    distances[first] = 0 # корневая вершина равна 0
    path_dict: Dict[int, WeightedEdge] = {} # как добраться до каждой вершины
    pq: PriorityQueue[DijkstraNode] = PriorityQueue()
    pq.push(DijkstraNode(first, 0))
```

Первый узел, помещенный в очередь с приоритетом, содержит корневую вершину.

```
while not pq.empty:
    u: int = pq.pop().vertex # исследовать ближайшую вершину
    dist_u: float = distances[u] # это мы уже, должно быть, видели
```

Мы будем выполнять алгоритм Дейкстры до тех пор, пока очередь с приоритетом не опустеет. `U` — текущая вершина, с которой начинается поиск, а `dist_u` — сохраненное расстояние, позволяющее добраться до `U` по известным маршрутам. Все вершины, исследованные на этом этапе, уже найдены, поэтому для них расстояние уже известно.

```
# рассмотреть все ребра и вершины для данной вершины
for we in wg.edges_for_index(u):
    # старое расстояние до этой вершины
    dist_v: float = distances[we.v]
```

Затем исследуются все ребра, связанные с `u`. `dist_v`, — это расстояние до всех известных вершин, соединенных ребром с `u`.

```
# старого расстояния не существует или найден более короткий путь
if dist_v is None or dist_v > we.weight + dist_u:
    # изменить расстояние до этой вершины
    distances[we.v] = we.weight + dist_u
    # заменить ребро на более короткий путь к этой вершине
    path_dict[we.v] = we
    # вскоре мы это проверим
    pq.push(DijkstraNode(we.v, we.weight + dist_u))
```

Если мы обнаружили вершину, которая еще не была исследована (значение `dist_v` равно `None`), или нашли новый, более короткий путь к ней, то записываем новое кратчайшее расстояние до `v` и ребро, которое привело нас туда. Наконец, помещаем все вершины с новыми путями в очередь с приоритетом.

```
return distances, path_dict
```

`dijkstra()` возвращает расстояния от корневой вершины до каждой вершины взвешенного графа, и `path_dict`, что позволяет определить кратчайшие пути к ним.

Теперь можно безопасно использовать алгоритм Дейкстры. Начнем с определения расстояния от Лос-Анджелеса до всех остальных MSA на графе. Тогда мы найдем кратчайший путь между Лос-Анджелесом и Бостоном (листинг 4.14). А в конце воспользуемся `print_weighted_path()`, чтобы красиво распечатать результат.

Листинг 4.14. dijkstra.py (продолжение)

```
if __name__ == "__main__":
    city_graph2: WeightedGraph[str] = WeightedGraph(["Seattle", "San
     Francisco", "Los Angeles", "Riverside", "Phoenix", "Chicago", "Boston",
     "New York", "Atlanta", "Miami", "Dallas", "Houston", "Detroit",
     "Philadelphia", "Washington"])

    city_graph2.add_edge_by_vertices("Seattle", "Chicago", 1737)
    city_graph2.add_edge_by_vertices("Seattle", "San Francisco", 678)
    city_graph2.add_edge_by_vertices("San Francisco", "Riverside", 386)
    city_graph2.add_edge_by_vertices("San Francisco", "Los Angeles", 348)
    city_graph2.add_edge_by_vertices("Los Angeles", "Riverside", 50)
    city_graph2.add_edge_by_vertices("Los Angeles", "Phoenix", 357)
```

```
    city_graph2.add_edge_by_vertices("Riverside", "Phoenix", 307)
    city_graph2.add_edge_by_vertices("Riverside", "Chicago", 1704)
    city_graph2.add_edge_by_vertices("Phoenix", "Dallas", 887)
    city_graph2.add_edge_by_vertices("Phoenix", "Houston", 1015)
    city_graph2.add_edge_by_vertices("Dallas", "Chicago", 805)
    city_graph2.add_edge_by_vertices("Dallas", "Atlanta", 721)
    city_graph2.add_edge_by_vertices("Dallas", "Houston", 225)
    city_graph2.add_edge_by_vertices("Houston", "Atlanta", 702)
    city_graph2.add_edge_by_vertices("Houston", "Miami", 968)
    city_graph2.add_edge_by_vertices("Atlanta", "Chicago", 588)
    city_graph2.add_edge_by_vertices("Atlanta", "Washington", 543)
    city_graph2.add_edge_by_vertices("Atlanta", "Miami", 604)
    city_graph2.add_edge_by_vertices("Miami", "Washington", 923)
    city_graph2.add_edge_by_vertices("Chicago", "Detroit", 238)
    city_graph2.add_edge_by_vertices("Detroit", "Boston", 613)
    city_graph2.add_edge_by_vertices("Detroit", "Washington", 396)
    city_graph2.add_edge_by_vertices("Detroit", "New York", 482)
    city_graph2.add_edge_by_vertices("Boston", "New York", 190)
    city_graph2.add_edge_by_vertices("New York", "Philadelphia", 81)
    city_graph2.add_edge_by_vertices("Philadelphia", "Washington", 123)

    distances, path_dict = dijkstra(city_graph2, "Los Angeles")
    name_distance: Dict[str, Optional[int]] = distance_array_to_vertex_
      dict(city_graph2, distances)
    print("Distances from Los Angeles:")
    for key, value in name_distance.items():
        print(f"{key} : {value}")
    print("") # пустая строка

    print("Shortest path from Los Angeles to Boston:")
    path: WeightedPath = path_dict_to_path(city_graph2.index_of("Los
      Angeles"), city_graph2.index_of("Boston"), path_dict)
    print_weighted_path(city_graph2, path)
```

Результат должен выглядеть примерно так:

```
Distances from Los Angeles:
Seattle : 1026
San Francisco : 348
Los Angeles : 0
Riverside : 50
Phoenix : 357
Chicago : 1754
Boston : 2605
New York : 2474
Atlanta : 1965
Miami : 2340
Dallas : 1244
Houston : 1372
Detroit : 1992
```

```
Philadelphia : 2511
Washington : 2388

Shortest path from Los Angeles to Boston:
Los Angeles 50> Riverside
Riverside 1704> Chicago
Chicago 238> Detroit
Detroit 613> Boston
Total Weight: 2605
```

Возможно, вы заметили, что у алгоритма Дейкстры есть некоторое сходство с алгоритмом Ярника. Оба они жадные, и при желании их можно реализовать, используя довольно похожий код. Другой алгоритм, похожий на алгоритм Дейкстры, — это A* из главы 2. A* можно рассматривать как модификацию алгоритма Дейкстры. Добавьте эвристику и ограничьте алгоритм Дейкстры поиском только одного пункта назначения — и оба алгоритма окажутся одинаковыми.

ПРИМЕЧАНИЕ

Алгоритм Дейкстры предназначен для графов с положительными весами. Графы с отрицательно взвешенными ребрами могут создать проблему и потребуют модификации или альтернативного алгоритма.

4.6. Реальные приложения

Очень многое в нашем мире можно представить в виде графов. В этой главе вы увидели, как эффективны графы при работе с транспортными сетями. С подобными важными проблемами оптимизации сталкиваются и многие другие виды сетей: телефонные, компьютерные, инженерные сети (электричество, водопровод и т. п.). Графовые алгоритмы необходимы для эффективного решения задач в области телекоммуникаций, судоходства, транспорта и коммунального хозяйства.

Предприятиям розничной торговли приходится решать сложные задачи дистрибуции. Магазины и склады можно рассматривать как вершины, а расстояния между ними — как ребра графов, к которым применяются все те же алгоритмы. Сам Интернет — это гигантский граф, в котором каждое подключенное устройство представляет собой вершину, а каждое проводное или беспроводное соединение — ребро. Минимальное связующее дерево и поиск кратчайшего пути помогут узнать, экономит ли компания топливо или провода для соединений, — эти алгоритмы полезны не только для игр. Некоторые из наиболее известных мировых брендов стали успешными благодаря оптимизации с использованием графовых задач: например, компания Walmart построила эффективную дистрибьюторскую сеть, Google проиндексировала Интернет (гигантский граф), а FedEx нашла правильный набор концентраторов для подключения к адресам по всему миру.

Одни из самых очевидных приложений графовых алгоритмов — это социальные сети и картографические приложения. В социальной сети люди играют роль

вершин, а связи между ними (такие как дружба в Facebook) — ребер. Один из известнейших инструментов разработчика Facebook так и называется — Graph API (https://developers.facebook.com/docs/graph-api). В картографических приложениях, таких как Apple Maps и Google Maps, графовые алгоритмы задействуются для указания направлений и расчета времени в пути.

Несколько популярных видеоигр также явно используют графовые алгоритмы. MiniMetro и Ticket to Ride — лишь два примера игр, моделирующих задачи, которые мы решали в этой главе.

4.7. Упражнения

1. Добавьте в графовую структуру возможность удаления ребер и вершин.
2. Добавьте в графовую структуру поддержку направленных графов (диграфов).
3. Применив графовую структуру, описанную в этой главе, докажите или опровергните классическую задачу о кенигсбергских мостах, описанную в «Википедии»: https://ru.wikipedia.org/wiki/Задача_о_семи_кёнигсбергских_мостах.

5 Генетические алгоритмы

Генетические алгоритмы не относятся к используемым для решения повседневных задач программирования. Они нужны тогда, когда традиционных алгоритмических подходов недостаточно для решения проблемы в разумные сроки. Другими словами, генетические алгоритмы обычно приберегают для сложных задач, не имеющих простых решений. Если вы хотите понять, что представляют собой некоторые из этих сложных проблем, не стесняйтесь обратиться к разделу 5.7, прежде чем продолжить чтение книги. Вот только один интересный пример: построение соединений белка и лиганда и разработка лекарств. Специалисты в области вычислительной биологии разрабатывают молекулы, которые будут связываться с рецепторами, чтобы доставить лекарства. Для конструирования конкретной молекулы может не существовать очевидного алгоритма, но, как вы увидите, иногда генетические алгоритмы позволяют дать ответ, не имея четкого определения цели поставленной задачи.

5.1. Немного биологической теории

В биологии теория эволюции объясняет, каким образом генетическая мутация в сочетании с ограничениями окружающей среды с течением времени приводит к изменениям в организме, включая видообразование — создание новых видов. Механизм, обусловливающий то, что хорошо адаптированные организмы процветают, а хуже адаптированные погибают, известен как *естественный отбор*. В каждое поколение определенного вида входят особи с различными, а иногда

и новыми чертами, которые возникают в результате генетической мутации. Все особи соревнуются за ограниченные ресурсы, чтобы выжить, и поскольку живых особей больше, чем ресурсов, то некоторые из них должны умереть.

Если особь имеет мутацию, которая делает ее лучше приспособленной к окружающей среде, то у нее выше вероятность выживания и воспроизводства. Со временем у особей, хорошо адаптированных к окружающей среде, будет больше потомков, которые унаследуют эту мутацию. Следовательно, мутация, способствующая выживанию, скорее всего, в итоге распространится на всю популяцию.

Например, если бактерии гибнут от определенного антибиотика, но у одной из них в популяции есть мутация в гене, которая делает ее более устойчивой к этому антибиотику, то эта бактерия с большей вероятностью выживет и размножится. Если антибиотик постоянно применяется в течение длительного времени, то потомки этой бактерии, унаследовавшие ее ген устойчивости к антибиотикам, также с большей вероятностью будут размножаться и иметь собственных потомков. В конце концов эта мутация может распространиться на всю популяцию бактерий, поскольку продолжающееся воздействие антибиотика будет убивать не обладающих ею особей. Антибиотик не вызывает развитие мутации, но приводит к размножению имеющих ее особей.

Естественный отбор существует не только в биологии. Социальный дарвинизм — это естественный отбор в сфере социальной теории. В информатике генетические алгоритмы — это моделирование естественного отбора для решения вычислительных задач.

Генетический алгоритм подразумевает наличие *популяции* (группы) особей, известных как *хромосомы*. Хромосомы, каждая из которых состоит из *генов*, определяющих ее свойства, конкурируют за решение некоей проблемы. То, насколько успешно хромосома решает проблему, определяется *функцией жизнеспособности*.

Генетический алгоритм обрабатывает *поколения*. В каждом поколении с большей вероятностью будут *отобраны* для размножения наиболее подходящие хромосомы. Также в каждом поколении существует вероятность того, что какие-то две хромосомы объединят свои гены. Это явление известно под названием *«кроссинговер»*. И наконец, в каждом поколении существует важная возможность того, что какой-то из генов хромосомы может *мутировать* — измениться случайным образом.

Если функция жизнеспособности какой-либо из особей популяции пересекает некий заданный порог или алгоритм проходит некоторое указанное максимальное число поколений, возвращается наилучшая особь — та, которая набрала наибольшее количество баллов в функции жизнеспособности.

Генетические алгоритмы — не самое хорошее решение для любых задач. Они зависят от трех частично или полностью *стохастических* (случайно определенных) операций: отбора, кроссинговера и мутации, — поэтому могут не найти оптимального решения в разумные сроки. Для большинства задач существуют более детерминированные алгоритмы, дающие лучшие гарантии. Но бывают задачи, для которых не существует быстрых детерминированных алгоритмов. В таких случаях генетические алгоритмы являются хорошим выбором.

5.2. Обобщенный генетический алгоритм

Генетические алгоритмы часто являются узкоспециализированными и рассчитаны на конкретное применение. В этой главе мы построим обобщенный генетический алгоритм. Его можно использовать для множества задач, но он не особенно хорошо приспособлен ни для одной из них. Алгоритм будет включать в себя некоторые настраиваемые параметры, но наша цель состоит в том, чтобы показать его основы, а не настраиваемость.

Прежде всего определим интерфейс для особей, с которыми может работать универсальный алгоритм. Абстрактный класс `Chromosome` определяет четыре основных свойства. Хромосома должна уметь:

- определять собственную жизнеспособность;
- создавать экземпляр со случайно выбранными генами (для использования при заполнении первого поколения);
- реализовывать кроссинговер (объединяться с другим объектом того же типа, чтобы создавать потомков), другими словами, скрещиваться с другой хромосомой;
- мутировать — вносить в себя небольшое случайное изменение.

Вот код `Chromosome`, в котором реализованы эти четыре свойства (листинг 5.1).

Листинг 5.1. chromosome.py

```
from __future__ import annotations
from typing import TypeVar, Tuple, Type
from abc import ABC, abstractmethod

T = TypeVar('T', bound='Chromosome') # чтобы возвращать self

# Базовый класс для всех хромосом; все методы должны быть переопределены
class Chromosome(ABC):
    @abstractmethod
    def fitness(self) -> float:
        ...

    @classmethod
    @abstractmethod
    def random_instance(cls: Type[T]) -> T:
        ...

    @abstractmethod
    def crossover(self: T, other: T) -> Tuple[T, T]:
        ...

    @abstractmethod
    def mutate(self) -> None:
        ...
```

СОВЕТ

Возможно, вы обратили внимание на то, что в конструкторе этого класса TypeVar T связан с Chromosome. Это означает, что все, что заполняет переменную типа T, должно быть экземпляром класса Chromosome или подкласса Chromosome.

Мы реализуем сам алгоритм (код, который будет манипулировать хромосомами) как параметризованный класс, открытый для создания подклассов для будущих специализированных приложений. Но прежде, чем сделать это, вернемся к описанию генетического алгоритма, представленного в начале главы, и четко определим шаги, которые он должен выполнить.

1. Создать начальную популяцию случайных хромосом для первого поколения алгоритма.
2. Измерить жизнеспособность каждой хромосомы в этом поколении популяции. Если жизнеспособность какой-то из них превышает пороговое значение, то вернуть его и закончить работу алгоритма.
3. Выбрать для размножения несколько особей. С самой высокой вероятностью для размножения выбираются наиболее жизнеспособные особи.
4. Скрестить (объединить) с некоторой вероятностью часть из выбранных хромосом, чтобы создать потомков, представляющих популяцию следующего поколения.
5. Выполнить мутацию: мутируют, как правило, с низкой вероятностью некоторые из хромосом. На этом формирование популяции нового поколения завершено, и она заменяет популяцию предыдущего поколения.
6. Если максимально допустимое количество поколений не получено, то вернуться к шагу 2. Если максимально допустимое количество поколений получено, вернуть наилучшую хромосому, найденную до сих пор.

На обобщенной схеме генетического алгоритма (рис. 5.1) недостает многих важных деталей. Сколько хромосом должно быть в популяции? Какой порог останавливает алгоритм? Как следует выбирать хромосомы для размножения? Как они должны скрещиваться и с какой вероятностью? С какой вероятностью должны происходить мутации? Сколько поколений нужно создать?

Все эти точки будут настраиваться в классе `GeneticAlgorithm` (листинг 5.2). Мы будем определять этот класс по частям, чтобы обсуждать каждую отдельно.

Листинг 5.2. genetic_algorithm.py

```
from __future__ import annotations
from typing import TypeVar, Generic, List, Tuple, Callable
from enum import Enum
from random import choices, random
from heapq import nlargest
```

```
from statistics import mean
from chromosome import Chromosome

C = TypeVar('C', bound=Chromosome) # тип хромосом

class GeneticAlgorithm(Generic[C]):
    SelectionType = Enum("SelectionType", "ROULETTE TOURNAMENT")
```

Рис. 5.1. Обобщенная схема генетического алгоритма

`GeneticAlgorithm` принимает параметризованный тип, который соответствует `Chromosome` и называется `C`. Перечисление `SelectionType` является внутренним типом, используемым для определения метода отбора, применяемого в алгоритме. Существует два наиболее распространенных метода отбора для генетического алгоритма — *отбор методом рулетки* (иногда называемый *пропорциональным отбором по жизнеспособности*) и *турнирный отбор*. Первый дает каждой хромосоме шанс быть отобранной пропорционально ее жизнеспособности. При турнирном отборе определенное количество случайно выбранных хромосом борются друг с другом и выбираются обладающие лучшей жизнеспособностью (листинг 5.3).

Листинг 5.3. genetic_algorithm.py (продолжение)

```
def __init__(self, initial_population: List[C], threshold: float,
     max_generations: int = 100, mutation_chance: float = 0.01,
     crossover_chance: float = 0.7, selection_type: SelectionType =
     SelectionType.TOURNAMENT) -> None:
    self._population: List[C] = initial_population
    self._threshold: float = threshold
    self._max_generations: int = max_generations
    self._mutation_chance: float = mutation_chance
    self._crossover_chance: float = crossover_chance
    self._selection_type: GeneticAlgorithm.SelectionType = selection_type
    self._fitness_key: Callable = type(self._population[0]).fitness
```

В приведенном коде представлены все свойства генетического алгоритма, которые будут настроены в момент создания посредством `__init__()`. `initial_population`, — хромосомы первого поколения алгоритма. `threshold` — порог жизнеспособности, который указывает на то, что решение задачи, над которым бьется генетический алгоритм, найдено. `max_generations` — максимальное число поколений, которое можно пройти. Если мы прошли столько поколений и не было найдено решение с уровнем жизнеспособности, превышающим `threshold`, то будет возвращено лучшее из найденных решений. `mutation_chance` — это вероятность мутации каждой хромосомы в каждом поколении. `crossover_chance` — вероятность того, что у двух родителей, отобранных для размножения, появятся потомки, представляющие собой смесь родительских генов, в противном случае они будут просто дубликатами родителей. Наконец, `selection_type` — это тип метода отбора, описанный в `enum SelectionType`.

Описанный ранее метод `init` принимает длинный список параметров, большинство которых имеют значения по умолчанию. Они устанавливают версии экземпляров настраиваемых свойств, которые мы только что обсуждали. В наших примерах `_population` инициализируется случайным набором хромосом с помощью метода `random_instance()` класса `Chromosome`. Другими словами, первое поколение хромосом состоит из случайных особей. Это точка потенциальной оптимизации для более сложного генетического алгоритма. Вместо того чтобы начинать с чисто случайных особей, первое поколение благодаря некоторому знанию конкретной задачи может содержать особей, находящихся ближе к решению. Это называется *посевом*.

`_fitness_key` — ссылка на метод, который мы будем использовать в `GeneticAlgorithm` для расчета жизнеспособности хромосомы. Напомню, что этот класс должен работать с любым подклассом `Chromosome`. Следовательно, функции `_fitness_key` будут разными в разных подклассах. Чтобы получить их, воспользуемся методом `type()` для ссылки на конкретный подкласс `Chromosome`, для которого мы хотим определить жизнеспособность.

Теперь рассмотрим два метода отбора, которые поддерживает наш класс (листинг 5.4).

Листинг 5.4. genetic_algorithm.py (продолжение)

```
# Используем метод рулетки с нормальным распределением,
# чтобы выбрать двух родителей
# Примечание: не работает при отрицательных значениях жизнеспособности
def _pick_roulette(self, wheel: List[float]) -> Tuple[C, C]:
    return tuple(choices(self._population, weights=wheel, k=2))
```

Выбор метода рулетки основан на отношении жизнеспособности каждой хромосомы к суммарной жизнеспособности всех хромосом данного поколения. Хромосомы с самой высокой жизнеспособностью имеют больше шансов быть отобранными. Значения, которые соответствуют жизнеспособности хромосомы, указаны в параметре `wheel`. Реальный выбор удобно выполнять с помощью функции `choices()` из модуля `random` стандартной библиотеки Python. Эта функция принимает список элементов, из которых мы хотим сделать выбор, список такой же длины, содержащий веса всех элементов первого списка, и число выбираемых элементов.

Если бы мы реализовывали это сами, то могли бы рассчитать в процентах долю от общей жизнеспособности для каждого элемента (пропорциональной жизнеспособности), представленную значениями с плавающей запятой от 0 до 1. Случайное число (`pick`) от 0 до 1 можно использовать для вычисления того, какую хромосому отобрать. Алгоритм будет работать, последовательно уменьшая `pick` на величину пропорциональной жизнеспособности каждой хромосомы. Когда `pick` станет меньше 0, это и будет хромосома для отбора.

Понимаете ли вы, почему этот процесс приводит к тому, что каждая хромосома выбирается по ее пропорциональной жизнеспособности? Если нет, подумайте об этом с карандашом и бумагой. Попробуйте нарисовать метод пропорционального отбора, как показано на рис. 5.2.

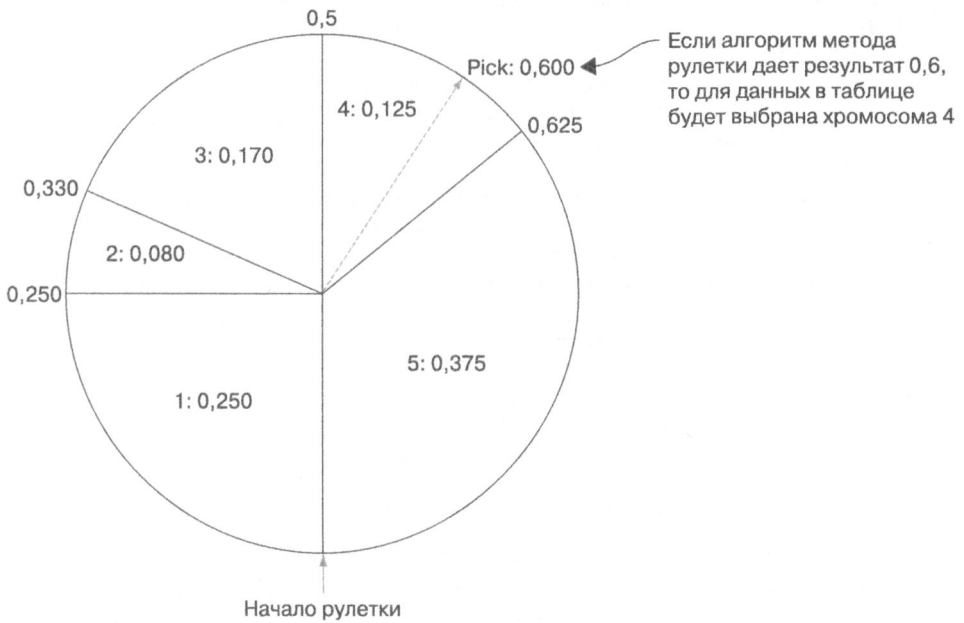
Простейшая форма турнирного отбора проще, чем отбор методом рулетки. Вместо того чтобы вычислять пропорции, мы просто выбираем случайным образом k хромосом из всей популяции. В отборе побеждают две хромосомы с наилучшей жизнеспособностью из случайно выбранной группы (листинг 5.5).

Листинг 5.5. genetic_algorithm.py (продолжение)

```
# Выбираем случайным образом num_participants и берем из них две лучших
def _pick_tournament(self, num_participants: int) -> Tuple[C, C]:
    participants: List[C] = choices(self._population, k=num_participants)
    return tuple(nlargest(2, participants, key=self._fitness_key))
```

В коде для `_pick_tournament()` вначале используется `choices()`, чтобы случайным образом выбрать `num_participants` из `_population`. Затем с помощью функции `nlargest()` из модуля `heapq` находим двух самых больших индивидуумов по `_fitness_key`. Каково правильное количество для `num_participants`? Как и во многих других параметрах генетического алгоритма, наилучшим способом определения этого значения может быть метод проб и ошибок. Следует иметь в виду, что чем больше участников турнира, тем меньше разнообразие в популяции, потому что хромосомы с плохой жизнеспособностью с большей вероятностью будут устра-

нены в процессе поединков между особями[1]. Более сложные формы турнирного отбора могут выбирать особей, которые являются не самыми лучшими, а лишь вторыми или третьими по жизнеспособности, основываясь на некоторой модели убывающей вероятности.



Хромосома	Жизнеспособность	Вероятность	Доля
1	54,5	25 %	0,250
2	17,44	8 %	0,080
3	37,06	17 %	0,170
4	27,25	13 %	0,125
5	81,75	38 %	0,375
Всего		100 %	

Рис. 5.2. Пример отбора методом рулетки

Методы `_pick_roulette()` и `_pick_tournament()` используются для отбора, выполняемого в процессе размножения. Воспроизводство реализовано в методе `_reproduce_and_replace()`, который заботится о том, чтобы на смену хромосомам последнего поколения пришла новая популяция с тем же количеством хромосом (листинг 5.6).

[1] *Sokolov A., Whitley D.* Unbiased Tournament Selection // GECCO'05. – June 25–29. – Washington, D. C., U.S.A., 2005. http://mng.bz/S7l6.

Листинг 5.6. genetic_algorithm.py (продолжение)

```
# Замена популяции новым поколением особей
def _reproduce_and_replace(self) -> None:
    new_population: List[C] = []
    # продолжаем, пока не заполним особями все новое поколение
    while len(new_population) < len(self._population):
        # выбор двух родителей
        if self._selection_type == GeneticAlgorithm.SelectionType.ROULETTE:
            parents: Tuple[C, C] = self._pick_roulette([x.fitness() for x in
            self._population])
        else:
            parents = self._pick_tournament(len(self._population) // 2)
        # потенциальное скрещивание двух родителей
        if random() < self._crossover_chance:
            new_population.extend(parents[0].crossover(parents[1]))
        else:
            new_population.extend(parents)
    # если число нечетное, то один лишний, поэтому удаляем его
    if len(new_population) > len(self._population):
        new_population.pop()
    self._population = new_population # заменяем ссылку
```

В `_reproduce_and_replace()` выполняются следующие основные операции.

1. Две хромосомы, называемые родителями (`parents`), отбираются для воспроизводства посредством одного из двух методов отбора. При турнирном отборе всегда проводится турнир среди половины популяции, но этот способ можно настраивать в конфигурации.
2. Существует вероятность `_crossover_chance` того, что для получения двух новых хромосом два родителя будут объединены. В этом случае они добавляются в новую популяцию (`new_population`). Если потомков нет, то два родителя просто добавляются в `new_population`.
3. Если новая популяция `new_population` содержит столько же хромосом, сколько и старая `_population`, то она ее заменяет. В противном случае возвращаемся к шагу 1.

Метод `_mutate()`, который реализует мутацию, очень прост, подробная реализация мутации предоставляется отдельным хромосомам (листинг 5.7).

Листинг 5.7. genetic_algorithm.py (продолжение)

```
# Каждая особь мутирует с вероятностью _mutation_chance
def _mutate(self) -> None:
    for individual in self._population:
        if random() < self._mutation_chance:
            individual.mutate()
```

Теперь у нас есть все строительные блоки, необходимые для запуска генетического алгоритма. Метод `run()` координирует этапы измерений, воспроизводства (включая отбор) и мутации, в процессе которых одно поколение популяции заменяется другим. Этот метод также отслеживает лучшие, наиболее жизнеспособные хромосомы, обнаруженные на любом этапе поиска (листинг 5.8).

Листинг 5.8. genetic_algorithm.py (продолжение)

```
# Выполнение генетического алгоритма для max_generations итераций
# и возвращение лучшей из найденных особей
def run(self) -> C:
    best: C = max(self._population, key=self._fitness_key)
    for generation in range(self._max_generations):
        # ранний выход, если превышен порог
        if best.fitness() >= self._threshold:
            return best
        print(f"Generation {generation} Best {best.fitness()} Avg
             {mean(map(self._fitness_key, self._population))}")
        self._reproduce_and_replace()
        self._mutate()
        highest: C = max(self._population, key=self._fitness_key)
        if highest.fitness() > best.fitness():
            best = highest # найден новый лучший результат
    return best # лучший найденный результат из _max_generations
```

Значение `best` позволяет отслеживать лучшую из найденных до сих пор хромосом. Основной цикл выполняется `_max_generations` раз. Если какая-либо хромосома превышает порог жизнеспособности, то она возвращается и метод заканчивается. В противном случае метод вызывает `_reproduce_and_replace()` и `_mutate()` для создания следующего поколения и повторного запуска цикла. Если достигается значение `_max_generations`, то возвращается лучшая хромосома из найденных до сих пор.

5.3. Примитивный тест

Параметризованный генетический алгоритм `GeneticAlgorithm` будет работать с любым типом, который реализует `Chromosome`. В качестве теста выполним простую задачу, которую легко решить традиционными методами. Мы попробуем найти максимизирующие значения для уравнения $6x - x^2 + 4y - y^2$ — другими словами, значения x и y, при которых результат этого уравнения будет наибольшим.

Для того чтобы найти максимизирующие значения, можно использовать математический анализ — взять частные производные и установить каждую из них равной нулю. Результатом будет $x = 3$ и $y = 2$. Сможет ли наш генетический алгоритм получить тот же результат без применения математического анализа? Давайте разберемся (листинг 5.9).

Листинг 5.9. simple_equation.py

```
from __future__ import annotations
from typing import Tuple, List
from chromosome import Chromosome
from genetic_algorithm import GeneticAlgorithm
from random import randrange, random
from copy import deepcopy

class SimpleEquation(Chromosome):
    def __init__(self, x: int, y: int) -> None:
        self.x: int = x
        self.y: int = y

    def fitness(self) -> float: # 6x - x^2 + 4y - y^2
        return 6 * self.x - self.x * self.x + 4 * self.y - self.y * self.y

    @classmethod
    def random_instance(cls) -> SimpleEquation:
        return SimpleEquation(randrange(100), randrange(100))

    def crossover(self, other: SimpleEquation) -> Tuple[SimpleEquation,
        SimpleEquation]:
        child1: SimpleEquation = deepcopy(self)
        child2: SimpleEquation = deepcopy(other)
        child1.y = other.y
        child2.y = self.y
        return child1, child2

    def mutate(self) -> None:
        if random() > 0.5: # мутация x
            if random() > 0.5:
                self.x += 1
            else:
                self.x -= 1
        else: # иначе - мутация y
            if random() > 0.5:
                self.y += 1
            else:
                self.y -= 1

    def __str__(self) -> str:
        return f"X: {self.x} Y: {self.y} Fitness: {self.fitness()}"
```

`SimpleEquation` соответствует `Chromosome` и в соответствии со своим названием делает это как можно проще. Гены хромосомы `SimpleEquation` можно рассматривать как x и y.

Метод `fitness()` оценивает жизнеспособность x и y с помощью уравнения $6x - x^2 + 4y - y^2$. Чем больше значение, тем выше жизнеспособность данной хромосомы в соответствии с `GeneticAlgorithm`. При использовании случайного экземпля-

ра x и y изначально присваиваются случайные целые числа из диапазона 0... 100, поэтому `random_instance()` нужно только создать новый экземпляр `SimpleEquation` с этими значениями. Чтобы скрестить два `SimpleEquation` в `crossover()`, значения y этих экземпляров просто меняются местами для создания двух дочерних элементов. `mutate()` случайным образом увеличивает или уменьшает x или y. В общих чертах это все.

Поскольку `SimpleEquation` соответствует `Chromosome`, то мы сразу можем подключить его к `GeneticAlgorithm` (листинг 5.10).

Листинг 5.10. simple_equation.py (продолжение)

```
if __name__ == "__main__":
    initial_population: List[SimpleEquation] = [SimpleEquation.random_
     instance() for _ in range(20)]
    ga: GeneticAlgorithm[SimpleEquation] = GeneticAlgorithm(initial_
     population=initial_population, threshold=13.0, max_generations = 100,
     mutation_chance = 0.1, crossover_chance = 0.7)
    result: SimpleEquation = ga.run()
    print(result)
```

Использованные здесь параметры были получены методом догадки и проверки. Вы можете попробовать другие методы. Значение `threshold` было установлено равным 13,0, поскольку мы уже знаем правильный ответ. При $x = 3$ и $y = 2$ значение уравнения равно 13.

Если ответ заранее не известен, вы, возможно, захотите получить наилучший результат, который можно найти за определенное количество поколений. В этом случае можно установить порог равным произвольному большому числу. Помните: поскольку генетические алгоритмы являются стохастическими, все проходы алгоритма будут разными.

Вот пример выходных данных одного из проходов, при котором генетический алгоритм решил уравнение за девять поколений:

```
Generation 0 Best -349 Avg -6112.3
Generation 1 Best 4 Avg -1306.7
Generation 2 Best 9 Avg -288.25
Generation 3 Best 9 Avg -7.35
Generation 4 Best 12 Avg 7.25
Generation 5 Best 12 Avg 8.5
Generation 6 Best 12 Avg 9.65
Generation 7 Best 12 Avg 11.7
Generation 8 Best 12 Avg 11.6
X: 3 Y: 2 Fitness: 13
```

Как видите, алгоритм пришел к правильному решению, полученному ранее с применением математического анализа: $x = 3$, $y = 2$. Вы также могли заметить, что почти в каждом поколении алгоритм все ближе подходил к правильному ответу.

Следует принять во внимание, что для поиска решения генетический алгоритм потребовал больше вычислительных ресурсов, чем другие методы. В реальном

мире применение генетического алгоритма для столь простой задачи, как поиск максимума, едва ли будет хорошей идеей. Но этой простой реализации достаточно, чтобы показать: наш генетический алгоритм работает.

5.4. SEND + MORE = MONEY, улучшенный вариант

В главе 3 мы решили классическую криптоарифметическую цифровую задачу SEND + MORE = MONEY, используя структуру с ограничениями. (Чтобы вспомнить, о чем она, вернитесь к ее описанию в главе 3.) Эта задача может быть решена за разумное время и с помощью генетического алгоритма.

Одной из самых больших трудностей при формулировании задачи для ее решения с помощью генетического алгоритма является определение того, как ее представить. Удобное представление для криптоарифметических задач — использование индексов списка в виде цифр[1]. Таким образом, для представления десяти возможных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) требуется список из десяти элементов. Символы, которые нужно найти в задаче, можно затем перемещать с места на место. Например, если есть подозрение, что решение задачи включает в себя символ E, представленный цифрой 4, то `list[4] = "E"`. В записи SEND + MORE = MONEY используются восемь различных букв (S, E, N, D, M, O, R, Y), так что два слота в массиве останутся пустыми. Их можно заполнить пробелами без указания буквы.

Хромосома, представляющая задачу SEND + MORE = MONEY, выражена в виде `SendMoreMoney2` (листинг 5.11). Обратите внимание на то, что метод `fitness()` поразительно похож на `satisfied()` из `SendMoreMoneyConstraint` в главе 3.

Листинг 5.11. send_more_money2.py

```
from __future__ import annotations
from typing import Tuple, List
from chromosome import Chromosome
from genetic_algorithm import GeneticAlgorithm
from random import shuffle, sample
from copy import deepcopy

class SendMoreMoney2(Chromosome):
    def __init__(self, letters: List[str]) -> None:
        self.letters: List[str] = letters

    def fitness(self) -> float:
        s: int = self.letters.index("S")
```

[1] *Abbasian R., Mazloom M.* Solving Cryptarithmetic Problems Using Parallel Genetic Algorithm // Second International Conference on Computer and Electrical Engineering, 2009. http://mng.bz/RQ7V.

```
        e: int = self.letters.index("E")
        n: int = self.letters.index("N")
        d: int = self.letters.index("D")
        m: int = self.letters.index("M")
        o: int = self.letters.index("O")
        r: int = self.letters.index("R")
        y: int = self.letters.index("Y")
        send: int = s * 1000 + e * 100 + n * 10 + d
        more: int = m * 1000 + o * 100 + r * 10 + e
        money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
        difference: int = abs(money - (send + more))
        return 1 / (difference + 1)

    @classmethod
    def random_instance(cls) -> SendMoreMoney2:
        letters = ["S", "E", "N", "D", "M", "O", "R", "Y", " ", " "]
        shuffle(letters)
        return SendMoreMoney2(letters)

    def crossover(self, other: SendMoreMoney2) -> Tuple[SendMoreMoney2,
       SendMoreMoney2]:
        child1: SendMoreMoney2 = deepcopy(self)
        child2: SendMoreMoney2 = deepcopy(other)
        idx1, idx2 = sample(range(len(self.letters)), k=2)
        l1, l2 = child1.letters[idx1], child2.letters[idx2]
        child1.letters[child1.letters.index(l2)], child1.letters[idx2] =
          child1.letters[idx2], l2
        child2.letters[child2.letters.index(l1)], child2.letters[idx1] =
          child2.letters[idx1], l1
        return child1, child2

    def mutate(self) -> None: # поменять две буквы местами
        idx1, idx2 = sample(range(len(self.letters)), k=2)
        self.letters[idx1], self.letters[idx2] = self.letters[idx2],
          self.letters[idx1]

    def __str__(self) -> str:
        s: int = self.letters.index("S")
        e: int = self.letters.index("E")
        n: int = self.letters.index("N")
        d: int = self.letters.index("D")
        m: int = self.letters.index("M")
        o: int = self.letters.index("O")
        r: int = self.letters.index("R")
        y: int = self.letters.index("Y")
        send: int = s * 1000 + e * 100 + n * 10 + d
        more: int = m * 1000 + o * 100 + r * 10 + e
        money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
        difference: int = abs(money - (send + more))
        return f"{send} + {more} = {money} Difference: {difference}"
```

Однако существует серьезное различие между методом `satisfied()` из главы 3 и методом `fitness()`, задействованным в данной. Здесь мы возвращаем `1/(difference + 1)`. `difference` — это абсолютное значение разности между MONEY и SEND + MORE. Оно показывает, насколько далека данная хромосома от решения задачи. Если бы мы пытались минимизировать `fitness()`, то возвращения одного лишь значения `difference` было бы достаточно. Но поскольку `GeneticAlgorithm` пытается максимизировать `fitness()`, необходимо вычислить обратное значение, чтобы меньшие значения выглядели как большие, и поэтому мы делим 1 на `difference`. Для того чтобы `difference(0)` было равно не `fitness(0)`, а `fitness(1)`, к `difference` сначала прибавляется 1. В табл. 5.1 показано, как это работает.

Таблица 5.1. Как уравнение 1 / (difference + 1) позволяет вычислить жизнеспособность с целью ее максимизации

difference	difference + 1	fitness (1/(difference + 1))
0	1	1,000
1	2	0,500
2	3	0,250
3	4	0,125

Помните: чем меньше разность, тем лучше и чем больше жизнеспособность, тем лучше. Поскольку данная формула объединяет эти два фактора, то она хорошо работает. Деление 1 на значение жизнеспособности — простой способ преобразования задачи минимизации в задачу максимизации. Однако это вносит некоторые сдвиги, так что результат оказывается ненадежным[1].

В методе `random_instance()` используется функция `shuffle()` из модуля `random`. Метод `crossover()` выбирает два случайных индекса в списках `letters` обеих хромосом и переставляет буквы так, чтобы одна из букв первой хромосомы оказалась на том же месте во второй хромосоме и наоборот. Он выполняет эти перестановки в дочерних хромосомах, так что размещение букв в двух дочерних хромосомах оказывается комбинацией родительских хромосом. `mutate()` меняет местами две случайные буквы в списке `letters`.

`SendMoreMoney2` подключается к `GeneticAlgorithm` так же легко, как и `SimpleEquation` (листинг 5.12). Имейте в виду: это довольно сложная задача и ее выполнение займет много времени, если неудачно настроить исходные параметры.

[1] Например, мы могли бы получить больше чисел, находящихся ближе к 0, чем к 1, если бы просто делили 1 на равномерное распределение целых чисел, которое с учетом особенностей интерпретации типичными микропроцессорами чисел с плавающей точкой могло бы привести к неожиданным результатам. Альтернативный способ превратить задачу минимизации в задачу максимизации — просто сменить знак (плюс на минус). Однако это будет работать только в том случае, если изначально все значения положительные.

Но даже при правильной настройке существует определенный элемент случайности! Задача может быть решена в течение нескольких секунд или нескольких минут. К сожалению, такова природа генетических алгоритмов.

Листинг 5.12. send_more_money2.py (продолжение)

```
if __name__ == "__main__":
    initial_population: List[SendMoreMoney2] = [SendMoreMoney2.random_
        instance() for _ in range(1000)]
    ga: GeneticAlgorithm[SendMoreMoney2] = GeneticAlgorithm(initial_
        population=initial_population, threshold=1.0,
        max_generations = 1000, mutation_chance = 0.2,
        crossover_chance = 0.7,
        selection_type=GeneticAlgorithm.SelectionType.ROULETTE)
    result: SendMoreMoney2 = ga.run()
    print(result)
```

Следующие результаты были получены при запуске алгоритма, при котором задача была решена за три поколения с использованием 1000 особей в каждом поколении, созданном ранее. Проверьте, удастся ли вам настроить параметры `GeneticAlgorithm` так, чтобы получить аналогичный результат с меньшим количеством особей. Вам не кажется, что этот алгоритм лучше работает с отбором по методу рулетки, чем с турнирным отбором?

```
Generation 0 Best 0.0040650406504065045 Avg 8.854014252391551e-05
Generation 1 Best 0.16666666666666666 Avg 0.001277329479413134
Generation 2 Best 0.5 Avg 0.014920889170684687
8324 + 913 = 9237 Difference: 0
```

Согласно этому решению SEND = 8324, MORE = 913 и MONEY = 9237. Как это возможно? Похоже, в решении буквы отсутствуют. Фактически если M = 0, то существует несколько решений задачи, невозможных в версии, представленной в главе 3. Здесь MORE — это 0913, а MONEY — 09237. Ноль просто игнорируется.

5.5. Оптимизация сжатия списка

Предположим, что у нас есть некая информация, которую мы хотим сжать. Допустим, что это список предметов и их последовательность неважна, лишь бы все они не были повреждены. При какой последовательности элементов степень сжатия будет максимальной? Известно ли вам вообще, что для большинства алгоритмов сжатия порядок элементов влияет на степень сжатия?

Ответ будет зависеть от используемого алгоритма сжатия. В этом примере применим функцию `compress()` из модуля `zlib` со стандартными настройками. Далее показано полное решение для списка из 12 имен (листинг 5.13). Если не брать генетический алгоритм, а просто запустить `compress()` для 12 имен в том порядке, в котором они были представлены, то получим сжатые данные объемом 165 байт.

Листинг 5.13. list_compression.py

```
from __future__ import annotations
from typing import Tuple, List, Any
from chromosome import Chromosome
from genetic_algorithm import GeneticAlgorithm
from random import shuffle, sample
from copy import deepcopy
from zlib import compress
from sys import getsizeof
from pickle import dumps

# сжатые данные объемом 165 байт
PEOPLE: List[str] = ["Michael", "Sarah", "Joshua", "Narine", "David",
    "Sajid", "Melanie", "Daniel", "Wei", "Dean", "Brian", "Murat", "Lisa"]

class ListCompression(Chromosome):
    def __init__(self, lst: List[Any]) -> None:
        self.lst: List[Any] = lst

    @property
    def bytes_compressed(self) -> int:
        return getsizeof(compress(dumps(self.lst)))

    def fitness(self) -> float:
        return 1 / self.bytes_compressed

    @classmethod
    def random_instance(cls) -> ListCompression:
        mylst: List[str] = deepcopy(PEOPLE)
        shuffle(mylst)
        return ListCompression(mylst)

    def crossover(self, other: ListCompression) -> Tuple[ListCompression,
      ListCompression]:
        child1: ListCompression = deepcopy(self)
        child2: ListCompression = deepcopy(other)
        idx1, idx2 = sample(range(len(self.lst)), k=2)
        l1, l2 = child1.lst[idx1], child2.lst[idx2]
        child1.lst[child1.lst.index(l2)], child1.lst[idx2] =
          child1.lst[idx2], l2
        child2.lst[child2.lst.index(l1)], child2.lst[idx1] =
          child2.lst[idx1], l1
        return child1, child2

    def mutate(self) -> None: # поменять два элемента местами
        idx1, idx2 = sample(range(len(self.lst)), k=2)
        self.lst[idx1], self.lst[idx2] = self.lst[idx2], self.lst[idx1]

    def __str__(self) -> str:
        return f"Order: {self.lst} Bytes: {self.bytes_compressed}"
```

```
if __name__ == "__main__":
    initial_population: List[ListCompression] = [ListCompression.random_
      instance() for _ in range(1000)]
    ga: GeneticAlgorithm[ListCompression] = GeneticAlgorithm(initial_
      population=initial_population, threshold=1.0,
      max_generations = 1000, mutation_chance = 0.2,
      crossover_chance = 0.7, selection_
      type=GeneticAlgorithm.SelectionType.TOURNAMENT)
    result: ListCompression = ga.run()
    print(result)
```

Обратите внимание на то, как эта реализация похожа на реализацию из SEND + MORE = MONEY (см. раздел 5.4). Функции `crossover()` и `mutate()` практически одинаковы. В решениях обеих задач мы берем список элементов, постоянно перестраиваем его и проверяем перестановки. Можно написать общий суперкласс для решения обеих задач, который работал бы с широким спектром задач. Любая задача, если ее можно представить в виде списка элементов, для которого требуется найти оптимальный порядок, может быть решена аналогичным образом. Единственная реальная точка настройки для этих подклассов — соответствующая функция жизнеспособности.

Выполнение `list_compression.py` может занять очень много времени. Так происходит потому, что, в отличие от двух предыдущих задач, мы не знаем заранее, что представляет собой правильный ответ, поэтому у нас нет реального порога, к которому следовало бы стремиться. Вместо этого мы устанавливаем число поколений и число особей в каждом поколении как произвольное большое число и надеемся на лучшее. Каково минимальное количество байтов, которое удастся получить при сжатии в результате перестановки 12 имен? Честно говоря, мы не знаем ответа на этот вопрос. В моем лучшем случае, используя конфигурацию из предыдущего решения, после 546 поколений генетический алгоритм нашел последовательность из 12 имен, которая дала сжатие размером 159 байт.

Всего лишь 6 байт по сравнению с первоначальным порядком — экономия составляет примерно 4 %. Можно сказать, что 4 % не имеет значения, но если бы это был гораздо больший список, который многократно передавался бы по Сети, то экономия выросла бы многократно. Представьте, что бы получилось, если бы это был список размером 1 Мбайт, который передавался бы через Интернет 10 000 000 раз. Если бы генетический алгоритм мог оптимизировать порядок списка для сжатия, чтобы сэкономить 4 %, то он сэкономил бы примерно 40 Кбайт на каждую передачу и в итоге 400 Гбайт пропускной способности для всех передач. Это не очень большая экономия, но она может оказаться значительной настолько, что будет иметь смысл один раз запустить алгоритм и получить почти оптимальный порядок сжатия.

Однако подумайте вот о чем: в действительности неизвестно, нашли ли мы оптимальную последовательность для 12 имен, не говоря уже о гипотетическом списке размером 1 Мбайт. Как об этом узнать? Поскольку мы не обладаем глубоким пониманием алгоритма сжатия, следует проверить степень сжатия для всех возможных

последовательностей в списке. Но даже для списка всего лишь из 12 пунктов это трудновыполнимые 479 001 600 возможных вариантов (12!, где знак ! означает факториал). Использование генетического алгоритма, который пытается найти оптимальный вариант, возможно, будет более целесообразным, даже если мы не знаем, является ли его окончательное решение действительно оптимальным.

5.6. Проблемы генетических алгоритмов

Генетические алгоритмы — это не панацея. В сущности, для большинства задач они не подходят. Для любой задачи, для которой существует быстрый детерминированный алгоритм, подход с использованием генетического алгоритма не имеет смысла. Стохастическая природа генетических алгоритмов не позволяет предсказать время их выполнения. Чтобы решить эту проблему, можно прерывать работу алгоритма через определенное количество поколений. Но тогда остается неясным, было ли найдено действительно оптимальное решение.

Стивен Скиена, автор одного из самых популярных текстов об алгоритмах, даже написал следующее: «Я никогда не сталкивался с задачей, для которой генетические алгоритмы показались бы мне подходящим способом решения. Кроме того, мне никогда не встречались вычислительные результаты, полученные с использованием генетических алгоритмов, которые произвели бы на меня благоприятное впечатление»[1].

Мнение Скиены несколько радикально, но оно свидетельствует о том, что к генетическим алгоритмам следует прибегать, только когда вы совершенно уверены: лучшего решения не существует. Еще одна проблема генетических алгоритмов — определить, как представить потенциальное решение задачи в виде хромосомы. Традиционная практика заключается в представлении большинства задач в виде двоичных строк (последовательности 1 и 0, необработанные биты). Зачастую это оптимально с точки зрения использования пространства и позволяет легко выполнять функции кроссинговера. Но большинство сложных задач не так легко представить в виде кратных битовых строк.

Еще одна, более специфичная особенность, на которую стоит обратить внимание, — это проблемы, связанные с отбором методом рулетки, описанным ранее. Отбор методом рулетки, иногда называемый пропорциональным отбором по жизнеспособности, может привести к отсутствию разнообразия в популяции из-за преобладания довольно хорошо подходящих особей при каждом отборе. В то же время, если значения жизнеспособности близки, то отбор методом рулетки может привести к отсутствию давления отбора[2]. Кроме того, этот метод не работает в задачах, в которых жизнеспособность может принимать отрицательные значения, как в простом уравнении в разделе 5.3.

---

[1] *Skiena S.* The Algorithm Design Manual, 2nd ed. Springer. — 2009.

[2] *Eiben A. E., Smith J. E.* Introduction to Evolutionary Computation, 2nd ed. Springer. — 2015.

Короче говоря, для большинства задач, достаточно больших, чтобы для них стоило использовать генетические алгоритмы, последние не могут гарантировать поиск оптимального решения за предсказуемое время. По этой причине такие алгоритмы лучше всего применять в ситуациях, которые требуют найти не оптимальное, а лишь приемлемое решение. Эти алгоритмы довольно легко реализовать, но настройка их параметров может потребовать множества проб и ошибок.

5.7. Реальные приложения

Несмотря на то, что писал Скиена, генетические алгоритмы часто и эффективно применяются в многочисленных пространствах задач. Их часто используют для сложных задач, которые не требуют абсолютно оптимальных решений, таких как задачи с ограничениями, если они слишком велики, чтобы их можно было решить с помощью традиционных методов. Одним из примеров таких задач являются сложные проблемы планирования.

Генетические алгоритмы нашли широкое применение в вычислительной биологии. Они были успешно использованы для соединения белка-лиганда, при котором требовался поиск конфигурации маленькой молекулы, связанной с реципиентом. Эти алгоритмы используются в фармацевтических исследованиях и для того, чтобы лучше понять механизмы природных явлений.

Одной из самых известных задач в области компьютерных наук является задача коммивояжера, к которой мы еще вернемся в главе 9. Бродячий торговец желает найти по карте кратчайший маршрут, чтобы посетить каждый город ровно один раз и вернуться в исходную точку. Эта формулировка может напомнить о минимальных связующих деревьях из главы 4, но в действительности это не так. Решение задачи коммивояжера представляет собой гигантский цикл, который сводит к минимуму затраты на его прохождение, тогда как минимальное связующее дерево минимизирует затраты на подключение каждого города. Человеку, путешествующему через минимальное связующее дерево городов, возможно, придется дважды посетить один и тот же город, чтобы добраться до каждого города. Несмотря на то что задачи выглядят похожими, не существует разумно рассчитанного алгоритма для поиска решения задачи коммивояжера для произвольного числа городов. Как было показано, генетические алгоритмы находят неоптимальные, но довольно хорошие решения за короткий промежуток времени. Эта задача широко применяется для эффективного распределения товаров. Например, диспетчеры грузовых автомобилей служб FedEx и UPS используют программное обеспечение для решения задачи коммивояжера в повседневной деятельности. Алгоритмы, помогающие решить эту задачу, позволяют сократить расходы в самых разных отраслях.

В компьютерном искусстве генетические алгоритмы иногда применяются для имитации фотографий с помощью стохастических методов. Представьте себе 50 многоугольников, случайным образом размещенных на экране и постепенно скручиваемых, поворачиваемых, перемещаемых, изменяющих размеры и цвет, пока

они не будут как можно точнее соответствовать фотографии. Результат выглядит как работа художника-абстракциониста или, если использовать более угловатые формы, как витраж.

Генетические алгоритмы — это часть более широкой области, называемой эволюционными вычислениями. Одной из областей эволюционных вычислений, тесно связанной с генетическими алгоритмами, является *генетическое программирование*, при котором программы и задействуют операции отбора, кроссинговера и мутации, чтобы, изменяя себя, найти неочевидные решения задач программирования. Генетическое программирование — не слишком широко используемая технология, но представьте себе будущее, в котором программы пишут себя сами.

Преимущество генетических алгоритмов состоит в том, что они легко поддаются распараллеливанию. В наиболее очевидной форме каждая популяция может быть смоделирована на отдельном процессоре. В более детализированном варианте для каждого индивида выделяется особый поток, в котором он может мутировать и участвовать в кроссинговере и где вычисляется его жизнеспособность. Существует также множество промежуточных возможностей.

5.8. Упражнения

1. Добавьте в `GeneticAlgorithm` поддержку расширенной формы турнирного отбора, которая может иногда выбирать вторую или третью по жизнеспособности хромосому, основываясь на уменьшающейся вероятности.
2. Добавьте к структуре с ограничениями, описанной в главе 3, новую функцию, которая решает любой произвольный CSP с использованием генетического алгоритма. Возможной мерой жизнеспособности является количество ограничений, которые разрешаются данной хромосомой.
3. Создайте класс `BitString`, который реализует хромосому. Для того чтобы вспомнить, что представляет собой битовая строка, обратитесь к главе 1. Затем примените новый класс для решения задачи простого уравнения из раздела 5.3. Как можно представить эту задачу в виде битовой строки?

6 Кластеризация методом k-средних

Человечество еще никогда не имело так много данных о столь многочисленных аспектах общественной жизни, как сегодня. Компьютеры отлично справляются с хранением наборов данных, но последние мало что значат для общества, пока не будут проанализированы людьми. Вычислительные методы способны направлять людей по пути извлечения смысла из наборов данных.

Кластеризация — это вычислительная технология, которая делит все единицы данных из набора на группы. Успешная кластеризация приводит к созданию групп, которые содержат единицы данных, связанные между собой. Для того чтобы выяснить, являются ли эти отношения значимыми, обычно требуется проверка человеком.

При кластеризации группа, называемая *кластером*, к которой принадлежит единица данных, не предопределяется, а определяется во время выполнения алгоритма кластеризации. В сущности, целью алгоритма не является размещение какой-либо конкретной единицы данных в каком-либо конкретном кластере на основании некоей заранее известной информации. По этой причине кластеризация считается *неконтролируемым* методом при машинном обучении. *Неконтролируемость* можно представить как *независимость от чего-то, что известно заранее.*

Кластеризация — это полезная технология, если требуется изучить структуру набора данных, но заранее ничего не известно о ее составных частях. Например, представьте, что у вас есть продуктовый магазин и вы собираете данные о клиентах и их покупках. Чтобы привлечь клиентов в магазин, вы намерены рассылать мобильную рекламу о специальных предложениях в соответствующие дни недели.

Можете попробовать кластеризовать данные по дням недели и демографической информации. Возможно, вы найдете кластер, указывающий на то, что молодые покупатели предпочитают делать покупки по вторникам, и тогда сможете использовать данную информацию для показа в этот день рекламы, специально ориентированной на них.

6.1. Предварительные сведения

Алгоритм кластеризации потребует некоторых статистических примитивов (среднее значение, стандартное отклонение и т. д.). Начиная с Python версии 3.4, в стандартную библиотеку языка входит несколько полезных статистических примитивов в модуле `statistics`. Следует отметить: хотя в книге мы и придерживаемся стандартной библиотеки, существуют более эффективные сторонние библиотеки для операций с числами, такие как NumPy, — их и следует использовать в приложениях, для которых важна производительность, особенно для работы с большими объемами данных.

Для простоты все наборы данных, с которыми мы будем работать в этой главе, можно выразить с помощью типа `float`, поэтому здесь будет много операций со списками и кортежами данных типа `float`. Статистические примитивы `sum()`, `mean()` и `pstdev()` определены в стандартной библиотеке. Их определения вытекают из формул, которые вы найдете в любом учебнике по статистике. Кроме того, нам понадобится функция для вычисления z-оценок (листинг 6.1).

Листинг 6.1. kmeans.py

```
from __future__ import annotations
from typing import TypeVar, Generic, List, Sequence
from copy import deepcopy
from functools import partial
from random import uniform
from statistics import mean, pstdev
from dataclasses import dataclass
from data_point import DataPoint

def zscores(original: Sequence[float]) -> List[float]:
    avg: float = mean(original)
    std: float = pstdev(original)
    if std == 0: # если нет изменений, то вернуть все нули
        return [0] * len(original)
    return [(x - avg) / std for x in original]
```

СОВЕТ

Функция pstdev() находит стандартное отклонение множества, а stdev(), которую мы не используем, — стандартное отклонение выборки.

Функция `zscores()` преобразует последовательность чисел в список чисел с соответствующими z-оценками исходных чисел относительно всех чисел в исходной последовательности. Подробнее о z-оценках будет рассказано позже в этой главе.

ПРИМЕЧАНИЕ

Обучение элементарной статистике выходит за рамки этой книги, но чтобы разобраться в содержании оставшейся части данной главы, вам понадобятся лишь элементарные знания о среднем и стандартном отклонениях. Если вы успели забыть, что это такое, и хотите освежить знания или же никогда ранее не изучали эти термины, возможно, стоит бегло просмотреть какой-нибудь ресурс, посвященный статистике, где объясняются эти фундаментальные понятия.

Все алгоритмы кластеризации работают с единицами данных, и наша реализация метода k-средних (k-means) не станет исключением. Мы определим общий интерфейс под названием `DataPoint`. Из соображений ясности кода определим его в отдельном файле (листинг 6.2).

Листинг 6.2. data_point.py

```
from __future__ import annotations
from typing import Iterator, Tuple, List, Iterable
from math import sqrt

class DataPoint:
    def __init__(self, initial: Iterable[float]) -> None:
        self._originals: Tuple[float, ...] = tuple(initial)
        self.dimensions: Tuple[float, ...] = tuple(initial)

    @property
    def num_dimensions(self) -> int:
        return len(self.dimensions)

    def distance(self, other: DataPoint) -> float:
        combined: Iterator[Tuple[float, float]] = zip(self.dimensions,
          other.dimensions)
        differences: List[float] = [(x - y) ** 2 for x, y in combined]
        return sqrt(sum(differences))

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, DataPoint):
            return NotImplemented
        return self.dimensions == other.dimensions

    def __repr__(self) -> str:
        return self._originals.__repr__()
```

Должна существовать возможность проверять каждую единицу данных на равенство другим единицам данных того же типа (`__eq__()`), а также выполнять удобочитаемый вывод единицы данных при отладке (`__repr__()`). Каждый тип единиц данных имеет определенное количество измерений (`num_dimensions`). В кортеже `dimensions` хранятся фактические значения для каждого из этих измерений в виде чисел типа `float`. Метод `__init__()` принимает итерируемый объект, содержащий значения нужных измерений. Впоследствии эти измерения могут быть заменены с помощью k-средних на z-оценки, поэтому мы также сохраняем в `_originals` копию исходных данных для последующего вывода.

И последнее, что нам нужно, прежде чем углубиться в алгоритм k-средних, — это способ вычисления расстояния между любыми двумя единицами данных одного типа. Существует много способов вычисления расстояния, но наиболее распространенная форма, используемая для k-средних, — евклидово расстояние. Это формула вычисления расстояния по теореме Пифагора, которую большинство из нас знает из школьных уроков геометрии. В сущности, мы уже обсуждали эту формулу и вывели ее версию для двумерных пространств в главе 2, где с ее помощью искали расстояние между любыми двумя точками в лабиринте. Версия для `DataPoint` должна быть более сложной, поскольку `DataPoint` может включать в себя любое количество измерений.

Эта версия `distance()` очень компактна и будет работать с типами `DataPoint`, имеющими любое количество измерений. Вызов `zip()` создает кортежи, заполненные для каждого измерения парами из двух точек, объединенных в последовательность. Генератор списков (list comprehension) находит разность между точками, составляющими пару, для каждого измерения и вычисляет квадрат этого значения. Функция `sum()` складывает все эти значения, итог, возвращаемый функцией `distance()`, является квадратным корнем из данной суммы.

6.2. Алгоритм кластеризации k-средних

Метод k-средних — это алгоритм кластеризации, который стремится сгруппировать единицы данных в некое заранее определенное количество кластеров, основываясь на относительном расстоянии от центра кластера до каждой единицы. В каждом периоде k-средних вычисляется расстояние между каждой единицей данных и каждым центром кластера — единицей данных, известной как *центроид*. Единицы данных присваиваются кластеру, к центроиду которого они ближе всего. Затем алгоритм пересчитывает все центроиды, находя среднее значение единиц данных, назначенных каждому кластеру, и заменяя старый центроид новым средним. Процесс назначения единиц данных и пересчета центроидов продолжается до тех пор, пока центроиды не перестанут передвигаться или не будет выполнено определенное количество итераций.

Все измерения начальных точек, представленных k-средними, должны быть сопоставимыми по величине. Иначе k-средние будут отклоняться в сторону кластеризации на основе измерений с наибольшим отличием. Процесс сопоставления разных типов данных (в нашем случае разных измерений) называется *нормализацией*. Одним из распространенных способов нормализации данных является приближенная оценка каждого значения на основе его z-оценки, известной также как *стандартная оценка*, относительно других значений того же типа. Z-оценка рассчитывается путем вычитания среднего всех значений из данного значения и деления результата на стандартное отклонение всех значений. Функция `zscores()`, разработанная в начале предыдущего раздела, делает именно это для каждого значения итерируемого объекта, состоящего из значений типа `float`.

Основная сложность, связанная с алгоритмом k-средних, заключается в определении способа выбора начальных центроидов. В простейшей форме алгоритма, которую мы будем реализовывать, начальные центроиды размещаются случайным образом в пределах диапазона данных. Сложно также решить, на сколько кластеров разделить данные (k в k-средних). В классическом алгоритме это число определяет пользователь, но он может не знать правильного числа, и его определение потребует некоторого количества экспериментов. Мы позволим пользователю определить k.

Объединяя все эти этапы и соображения, получим следующий алгоритм кластеризации методом k-средних.

1. Инициализировать все единицы данных и k пустых кластеров.
2. Нормализовать все единицы данных.
3. Создать случайные центроиды, связанные с каждым кластером.
4. Назначить каждую единицу данных кластеру того центроида, к которому она расположена ближе всего.
5. Пересчитать каждый центроид, чтобы он был центром (средним) кластера, с которым связан.
6. Повторять пункты 4 и 5, пока не будет выполнено максимально допустимое количество итераций или пока центроиды не перестанут передвигаться (сходиться).

Концептуально метод k-средних, в сущности, очень прост: на каждой итерации каждая единица данных связана с тем кластером, к центру которого она расположена ближе всего. По мере того как в кластер вносятся новые единицы данных, этот центр перемещается (рис. 6.1).

Далее реализован класс для поддержания состояния и выполнения алгоритма, аналогичный `GeneticAlgorithm`, описанному в главе 5. Теперь мы вернемся к файлу `kmeans.py` (листинг 6.3).

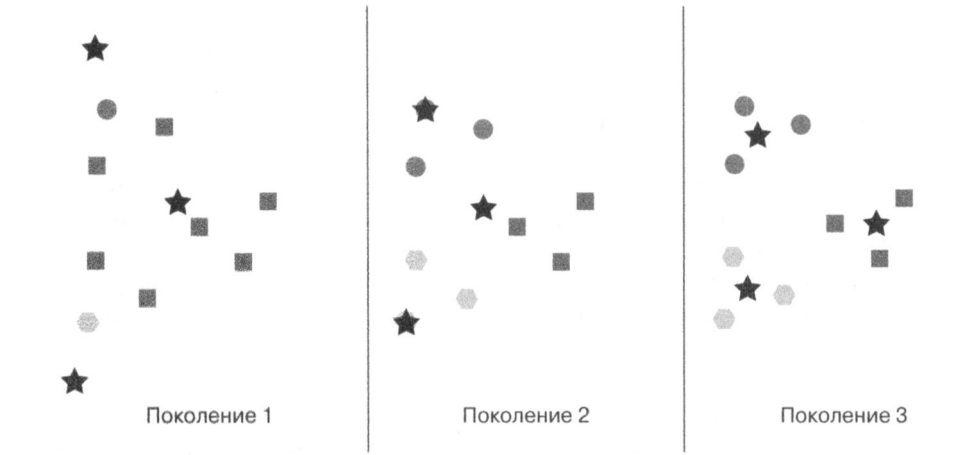


Рис. 6.1. Пример метода k-средних, выполняемого на протяжении трех поколений для произвольного набора данных. Звездочками отмечены центроиды. Оттенком и формой обозначено текущее членство в кластере (оно изменяется)

Листинг 6.3. kmeans.py (продолжение)

```
Point = TypeVar('Point', bound=DataPoint)

class KMeans(Generic[Point]):
    @dataclass
    class Cluster:
        points: List[Point]
        centroid: DataPoint
```

`KMeans` — это параметризованный класс. Он работает с `DataPoint` или с любым подклассом `DataPoint`, как определено значением `bound` типа `Point`. У него есть внутренний класс `Cluster`, который отслеживает отдельные кластеры в процессе выполнения операции. У каждого `Cluster` есть связанные с ним единицы данных и центроид.

Теперь определим метод `__init__()` для внешнего класса (листинг 6.4).

Листинг 6.4. kmeans.py (продолжение)

```
def __init__(self, k: int, points: List[Point]) -> None:
    if k < 1: # Число кластеров, создаваемых методом k-средних,
              # не может быть отрицательным или равным нулю
        raise ValueError("k must be >= 1")
    self._points: List[Point] = points
    self._zscore_normalize()
    # инициализируем пустые кластеры случайными центроидами
    self._clusters: List[KMeans.Cluster] = []
    for _ in range(k):
        rand_point: DataPoint = self._random_point()
```

```
        cluster: KMeans.Cluster = KMeans.Cluster([], rand_point)
        self._clusters.append(cluster)

@property
def _centroids(self) -> List[DataPoint]:
    return [x.centroid for x in self._clusters]
```

С классом `KMeans` связан массив `_points`. Это все единицы данных из набора данных. Затем единицы данных делятся между кластерами, которые хранятся в соответствующей переменной `_clusters`. Когда создается экземпляр `KMeans`, он должен знать, сколько кластеров требуется создать (`k`). Каждый кластер изначально имеет случайный центроид. Все единицы данных, которые будут использоваться в алгоритме, нормализуются по z-оценке. Вычисляемое свойство `_centroids` возвращает все центроиды, связанные с кластерами, создаваемыми алгоритмом (листинг 6.5).

Листинг 6.5. kmeans.py (продолжение)

```
def _dimension_slice(self, dimension: int) -> List[float]:
    return [x.dimensions[dimension] for x in self._points]
```

`_dimension_slice()` — это удобный метод, который можно представить как метод, возвращающий столбец данных. Он возвращает список, состоящий из всех значений для определенного индекса каждой единицы данных. Например, если бы единицы данных имели тип `DataPoint`, то `_dimension_slice(0)` вернул бы список значений первого измерения каждой единицы данных. Это будет полезно в следующем методе нормализации (листинг 6.6).

Листинг 6.6. kmeans.py (продолжение)

```
def _zscore_normalize(self) -> None:
    zscored: List[List[float]] = [[] for _ in range(len(self._points))]
    for dimension in range(self._points[0].num_dimensions):
        dimension_slice: List[float] = self._dimension_slice(dimension)
        for index, zscore in enumerate(zscores(dimension_slice)):
            zscored[index].append(zscore)
    for i in range(len(self._points)):
        self._points[i].dimensions = tuple(zscored[i])
```

`_zscore_normalize()` заменяет значения в кортеже измерений каждой единицы данных эквивалентом ее z-оценки. При этом используется функция `zscores()`, которую мы определили ранее для последовательностей данных типа `float`. Значения в кортеже `dimensions` заменены, но кортеж `_originals` в `DataPoint` не изменился. Это полезно: если значения хранятся в двух местах, то после выполнения алгоритма пользователь может получить исходные значения измерений, какими они были до нормализации (листинг 6.7).

Листинг 6.7. kmeans.py (продолжение)

```
def _random_point(self) -> DataPoint:
    rand_dimensions: List[float] = []
```

```
    for dimension in range(self._points[0].num_dimensions):
        values: List[float] = self._dimension_slice(dimension)
        rand_value: float = uniform(min(values), max(values))
        rand_dimensions.append(rand_value)
    return DataPoint(rand_dimensions)
```

Описанный ранее метод `_random_point()` применяется в методе `__init__()` для создания начальных случайных центроидов для каждого кластера. Это ограничивает случайные значения каждой единицы данных в пределах диапазона значений существующих единиц данных. Метод использует конструктор, описанный ранее в `DataPoint`, чтобы создать новую единицу данных из итерируемого объекта значений.

Теперь рассмотрим метод поиска подходящего кластера для единицы данных (листинг 6.8).

Листинг 6.8. kmeans.py (продолжение)

```
# Найти ближайший центроид кластера для каждой единицы данных
# и назначить единицу данных этому кластеру
def _assign_clusters(self) -> None:
    for point in self._points:
        closest: DataPoint = min(self._centroids,
          key=partial(DataPoint.distance, point))
        idx: int = self._centroids.index(closest)
        cluster: KMeans.Cluster = self._clusters[idx]
        cluster.points.append(point)
```

На протяжении этой книги мы уже создали несколько функций, которые находят минимум или максимум в заданном списке. Эта функция того же рода. В данном случае мы ищем кластерный центроид с минимальным расстоянием до каждой отдельной единицы данных. Затем эта единица данных присваивается кластеру. Единственным сложным моментом является то, что при использовании функции, опосредованной `partial()`, в качестве `key` для `min().partial()` принимается функция и перед применением ей задаются некоторые параметры. В данном случае мы предоставляем методу `DataPoint.distance()` в качестве параметра `other` единицу данных, которую вычисляем. Это приведет к тому, что функция `min()` будет возвращать расстояние каждого центроида до вычисляемой единицы данных и минимальное расстояние до центроида (листинг 6.9).

Листинг 6.9. kmeans.py (продолжение)

```
# найти центр каждого кластера и переместить центроид туда
def _generate_centroids(self) -> None:
    for cluster in self._clusters:
        if len(cluster.points) == 0:
        # оставить тот же центроид, если нет очков
            continue
        means: List[float] = []
        for dimension in range(cluster.points[0].num_dimensions):
```

```
            dimension_slice: List[float] = [p.dimensions[dimension] for p in
              cluster.points]
            means.append(mean(dimension_slice))
        cluster.centroid = DataPoint(means)
```

После того как все единицы данных назначены кластерам, вычисляются новые центроиды. Сюда входит вычисление среднего для каждого измерения каждой единицы данных в кластере. Затем средние значения для каждого измерения объединяются, чтобы найти среднюю точку в кластере, которая становится новым центроидом. Обратите внимание на то, что здесь нельзя использовать `_dimension_slice()`, потому что рассматриваемые единицы данных являются подмножеством всех единиц данных (те, что принадлежат конкретному кластеру). Как можно переписать метод `_dimension_slice()`, чтобы он был более универсальным?

Теперь рассмотрим метод, который фактически выполняет алгоритм (листинг 6.10).

Листинг 6.10. kmeans.py (продолжение)

```
def run(self, max_iterations: int = 100) -> List[KMeans.Cluster]:
    for iteration in range(max_iterations):
        for cluster in self._clusters: # очистить все кластеры
            cluster.points.clear()
        self._assign_clusters()
        # найти кластер, к которому текущая единица данных ближе всего
        old_centroids: List[DataPoint] = deepcopy(self._centroids)
        # записать
        self._generate_centroids() # найти новые центроиды
        if old_centroids == self._centroids: # сместились ли центроиды?
            print(f"Converged after {iteration} iterations")
            return self._clusters
    return self._clusters
```

`run()` — наиболее чистое выражение из всего исходного алгоритма. Единственное изменение алгоритма, которое вы можете посчитать неожиданным, — это удаление всех единиц данных в начале каждой итерации. Если бы этого не произошло, метод `_assign_clusters()` в том виде, как он написан, в итоге поместил бы в каждый кластер дубликаты единиц данных.

Мы можем быстро протестировать алгоритм, используя тестовые объекты `DataPoint` и присвоив `k` значение 2 (листинг 6.11).

Листинг 6.11. kmeans.py (продолжение)

```
if __name__ == "__main__":
    point1: DataPoint = DataPoint([2.0, 1.0, 1.0])
    point2: DataPoint = DataPoint([2.0, 2.0, 5.0])
    point3: DataPoint = DataPoint([3.0, 1.5, 2.5])
    kmeans_test: KMeans[DataPoint] = KMeans(2, [point1, point2, point3])
    test_clusters: List[KMeans.Cluster] = kmeans_test.run()
    for index, cluster in enumerate(test_clusters):
        print(f"Cluster {index}: {cluster.points}")
```

Из-за элемента случайности наши результаты могут различаться. Должно получиться что-то вроде следующего:

```
Converged after 1 iterations
Cluster 0: [(2.0, 1.0, 1.0), (3.0, 1.5, 2.5)]
Cluster 1: [(2.0, 2.0, 5.0)]
```

6.3. Кластеризация губернаторов по возрасту и долготе штата

В каждом американском штате есть губернатор. В июне 2017 года возраст этих чиновников находился в диапазоне от 42 до 79 лет. Если рассмотреть Соединенные Штаты с востока на запад, перебирая все штаты по долготе, то, возможно, удастся сформировать группы штатов с близкой долготой и близким возрастом губернаторов. На рис. 6.2 представлена диаграмма распределения всех 50 губернаторов. По оси X откладывается долгота штата, а по оси Y — возраст губернатора.

Существуют ли на рис. 6.2 какие-либо очевидные кластеры? Оси здесь не нормированы. Напротив, мы рассматриваем необработанные данные. Если бы кластеры всегда были очевидными, то не было бы необходимости в алгоритмах кластеризации.

Попробуем пропустить этот набор данных через алгоритм k-средних. Для этого в первую очередь понадобится способ представления отдельной единицы данных (листинг 6.12).

Листинг 6.12. governors.py

```
from __future__ import annotations
from typing import List
from data_point import DataPoint
from kmeans import Kmeans

class Governor(DataPoint):
    def __init__(self, longitude: float, age: float, state: str) -> None:
        super().__init__([longitude, age])
        self.longitude = longitude
        self.age = age
        self.state = state

    def __repr__(self) -> str:
        return f"{self.state}: (longitude: {self.longitude}, age: {self.age})"
```

У класса `Governor` есть два именованных и сохраненных измерения: `longitude` и `age`. Кроме того, `Governor` не вносит никаких других изменений в механизм своего суперкласса `DataPoint`, кроме переопределенного `__repr__()` для структурной печати (листинг 6.13). Едва ли было бы разумно вводить следующие данные вручную, поэтому обратитесь в репозиторий исходного кода, прилагаемый к этой книге.

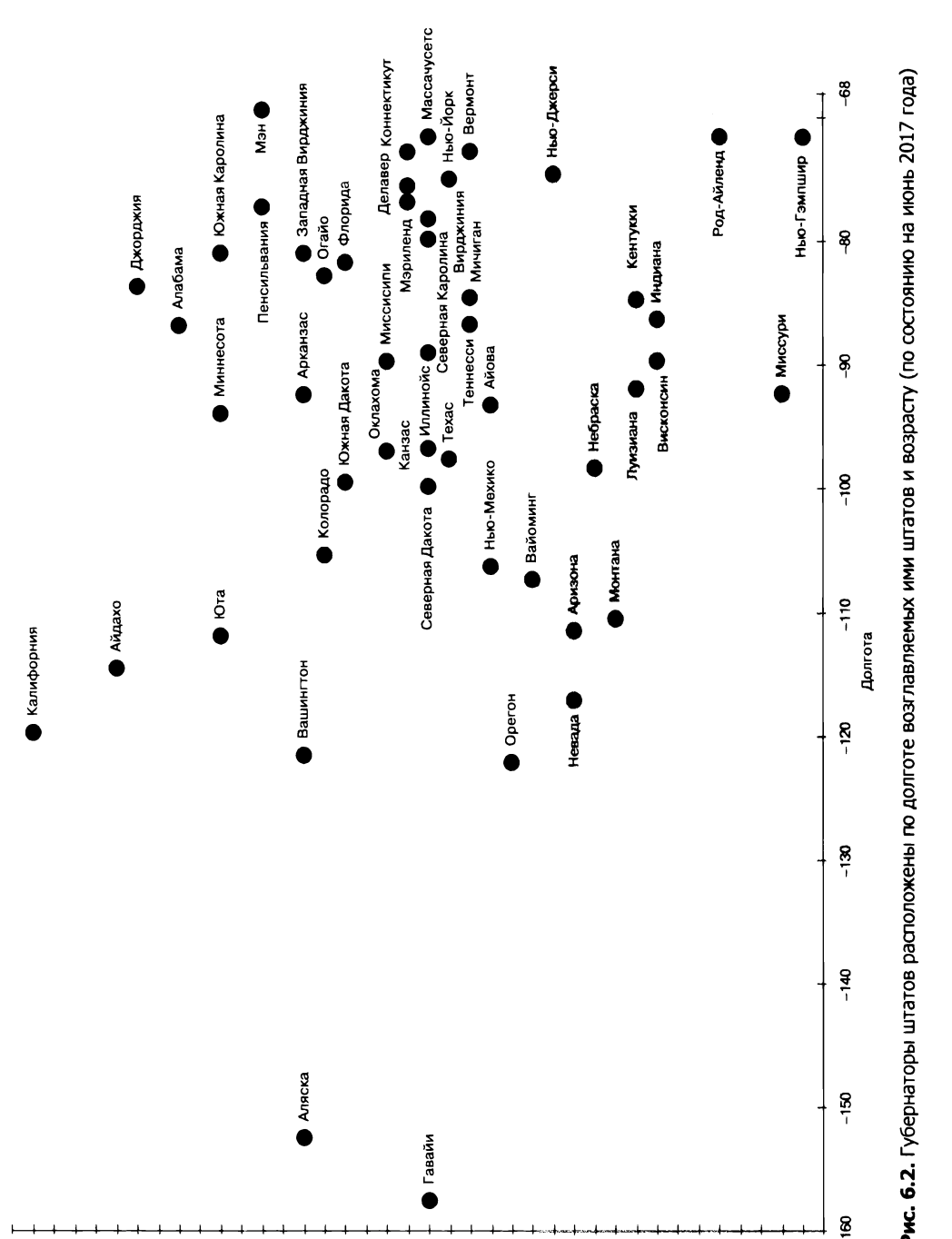


Рис. 6.2. Губернаторы штатов расположены по долготе возглавляемых ими штатов и возрасту (по состоянию на июнь 2017 года)

Листинг 6.13. governors.py (продолжение)

```
if __name__ == "__main__":
    governors: List[Governor] = [Governor(-86.79113, 72, "Alabama"),
        Governor(-152.404419, 66, "Alaska"),
            Governor(-111.431221, 53, "Arizona"), Governor(-92.373123,
              66, "Arkansas"),
            Governor(-119.681564, 79, "California"), Governor(-105.311104,
              65, "Colorado"),
            Governor(-72.755371, 61, "Connecticut"), Governor(-75.507141,
              61, "Delaware"),
            Governor(-81.686783, 64, "Florida"), Governor(-83.643074,
              74, "Georgia"),
            Governor(-157.498337, 60, "Hawaii"), Governor(-114.478828,
              75, "Idaho"),
            Governor(-88.986137, 60, "Illinois"), Governor(-86.258278,
              49, "Indiana"),
            Governor(-93.210526, 57, "Iowa"), Governor(-96.726486, 60,
              "Kansas"),
            Governor(-84.670067, 50, "Kentucky"), Governor(-91.867805,
              50, "Louisiana"),
            Governor(-69.381927, 68, "Maine"), Governor(-76.802101, 61,
              "Maryland"),
            Governor(-71.530106, 60, "Massachusetts"), Governor(-84.536095,
              58, "Michigan"),
            Governor(-93.900192, 70, "Minnesota"), Governor(-89.678696,
              62, "Mississippi"),
            Governor(-92.288368, 43, "Missouri"), Governor(-110.454353,
              51, "Montana"),
            Governor(-98.268082, 52, "Nebraska"), Governor(-117.055374,
              53, "Nevada"),
            Governor(-71.563896, 42, "New Hampshire"), Governor(-74.521011,
              54, "New Jersey"),
            Governor(-106.248482, 57, "New Mexico"), Governor(-74.948051,
              59, "New York"),
            Governor(-79.806419, 60, "North Carolina"), Governor(-99.784012,
              60, "North Dakota"),
            Governor(-82.764915, 65, "Ohio"), Governor(-96.928917, 62,
              "Oklahoma"),
            Governor(-122.070938, 56, "Oregon"), Governor(-77.209755,
              68, "Pennsylvania"),
            Governor(-71.51178, 46, "Rhode Island"), Governor(-80.945007,
              70, "South Carolina"),
            Governor(-99.438828, 64, "South Dakota"), Governor(-86.692345,
              58, "Tennessee"),
            Governor(-97.563461, 59, "Texas"), Governor(-111.862434, 70,
              "Utah"),
            Governor(-72.710686, 58, "Vermont"), Governor(-78.169968,
              60, "Virginia"),
            Governor(-121.490494, 66, "Washington"), Governor(-80.954453,
              66, "West Virginia"),
            Governor(-89.616508, 49, "Wisconsin"), Governor(-107.30249,
              55, "Wyoming")]
```

Запустим алгоритм k-средних с k, равным 2 (листинг 6.14).

Листинг 6.14. governors.py (продолжение)

```
kmeans: KMeans[Governor] = KMeans(2, governors)
gov_clusters: List[KMeans.Cluster] = kmeans.run()
for index, cluster in enumerate(gov_clusters):
    print(f"Cluster {index}: {cluster.points}\n")
```

Поскольку выполнение алгоритма начинается со случайно выбранных центроидов, то каждый запуск KMeans потенциально может возвращать разные кластеры. Чтобы увидеть, действительно ли это правильно выбранные кластеры, требуется анализ результатов человеком. Следующий результат получен после запуска, который дал действительно интересный кластер:

```
Сошлись после пяти итераций
Converged after 5 iterations
Cluster 0: [Alabama: (longitude: -86.79113, age: 72), Arizona: (longitude:
    -111.431221, age: 53), Arkansas: (longitude: -92.373123, age: 66),
    Colorado: (longitude: -105.311104, age: 65), Connecticut: (longitude:
    -72.755371, age: 61), Delaware: (longitude: -75.507141, age: 61),
    Florida: (longitude: -81.686783, age: 64), Georgia: (longitude:
    -83.643074, age: 74), Illinois: (longitude: -88.986137, age: 60),
    Indiana: (longitude: -86.258278, age: 49), Iowa: (longitude: -93.210526,
    age: 57), Kansas: (longitude: -96.726486, age: 60), Kentucky:
    (longitude: -84.670067, age: 50), Louisiana: (longitude: -91.867805,
    age: 50), Maine: (longitude: -69.381927, age: 68), Maryland: (longitude:
    -76.802101, age: 61), Massachusetts: (longitude: -71.530106, age: 60),
    Michigan: (longitude: -84.536095, age: 58), Minnesota: (longitude:
    -93.900192, age: 70), Mississippi: (longitude: -89.678696, age: 62),
    Missouri: (longitude: -92.288368, age: 43), Montana: (longitude:
    -110.454353, age: 51), Nebraska: (longitude: -98.268082, age: 52),
    Nevada: (longitude: -117.055374, age: 53), New Hampshire: (longitude:
    -71.563896, age: 42), New Jersey: (longitude: -74.521011, age: 54), New
    Mexico: (longitude: -106.248482, age: 57), New York: (longitude:
    -74.948051, age: 59), North Carolina: (longitude: -79.806419, age: 60),
    North Dakota: (longitude: -99.784012, age: 60), Ohio: (longitude:
    -82.764915, age: 65), Oklahoma: (longitude: -96.928917, age: 62),
    Pennsylvania: (longitude: -77.209755, age: 68), Rhode Island:
    (longitude: -71.51178, age: 46), South Carolina: (longitude: -80.945007,
    age: 70), South Dakota: (longitude: -99.438828, age: 64), Tennessee:
    (longitude: -86.692345, age: 58), Texas: (longitude: -97.563461, age:
    59), Vermont: (longitude: -72.710686, age: 58), Virginia: (longitude:
    -78.169968, age: 60), West Virginia: (longitude: -80.954453, age: 66),
    Wisconsin: (longitude: -89.616508, age: 49), Wyoming: (longitude:
    -107.30249, age: 55)]
Cluster 1: [Alaska: (longitude: -152.404419, age: 66), California:
    (longitude: -119.681564, age: 79), Hawaii: (longitude: -157.498337, age:
    60), Idaho: (longitude: -114.478828, age: 75), Oregon: (longitude:
    -122.070938, age: 56), Utah: (longitude: -111.862434, age: 70),
    Washington: (longitude: -121.490494, age: 66)]
```

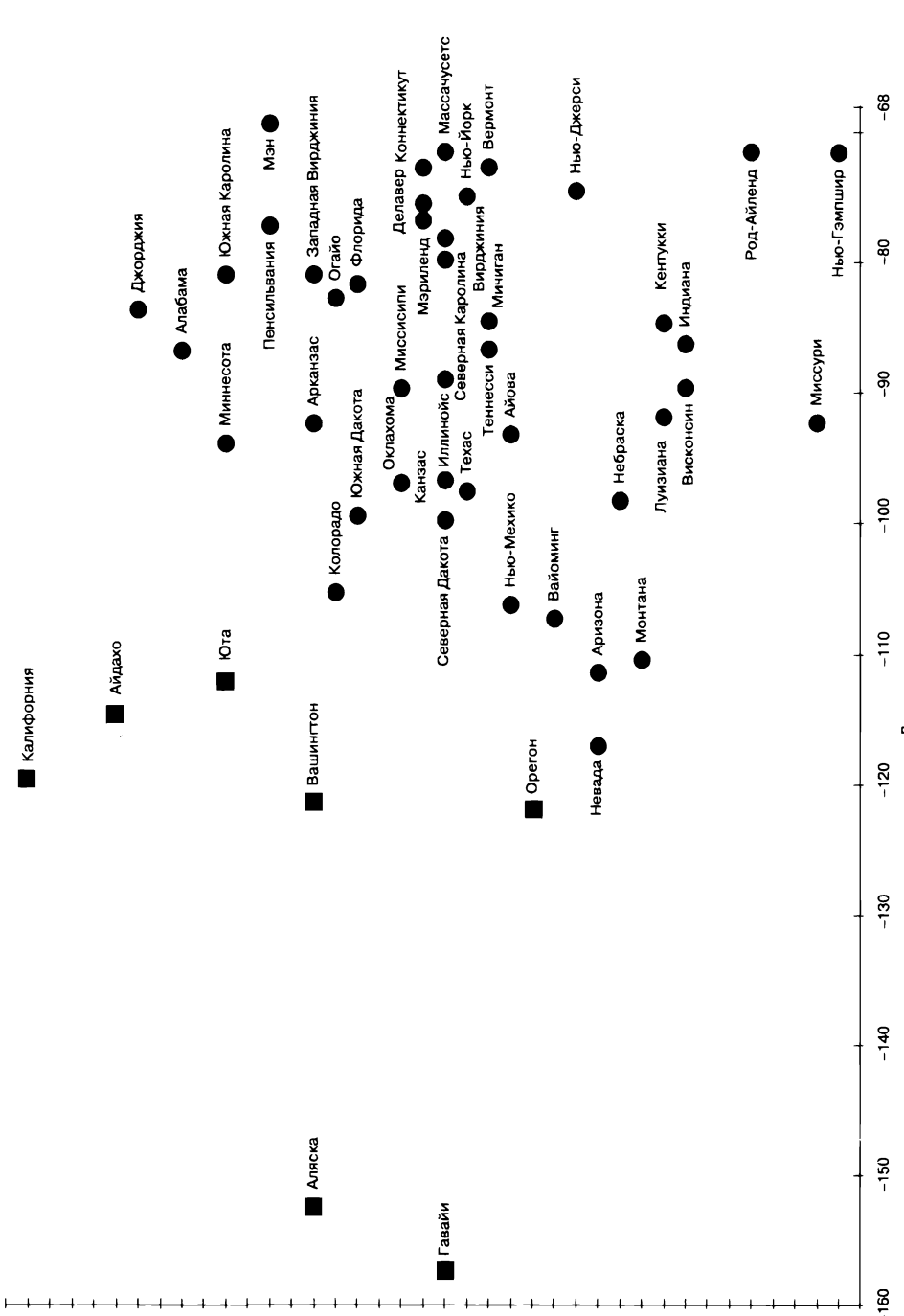


Рис. 6.3. Единицы данных кластера 0 обозначены кружками, а единицы данных кластера 1 — квадратиками

Кластер 1 представляет штаты крайнего Запада, все они географически расположены рядом друг с другом (если считать Аляску и Гавайи штатами Тихоокеанского побережья). Во всех них относительно старые губернаторы, следовательно, эти штаты образуют интересную группу. Население Тихоокеанского побережья предпочитает губернаторов постарше? Мы не можем сделать на основании этих кластеров какие-либо определенные выводы за пределами этой корреляции. Результат показан на рис. 6.3.

СОВЕТ

Необходимо еще и еще раз подчеркнуть, что результаты, полученные методом k-средних со случайной инициализацией центроидов, всегда будут различаться. Обязательно запустите алгоритм k-средних несколько раз, независимо от набора данных.

6.4. Кластеризация альбомов Майкла Джексона по длительности

Майкл Джексон выпустил десять сольных студийных альбомов. В следующем примере мы сгруппируем их, рассмотрев два измерения: длительность (в минутах) и количество дорожек. Этот пример хорошо контрастирует с предыдущим — о губернаторах, поскольку в этом исходном наборе данных кластеры легко увидеть даже без использования метода k-средних. Пример, подобный этому, может быть хорошим способом отладки реализации алгоритма кластеризации.

ПРИМЕЧАНИЕ

Оба примера, представленные в этой главе, задействуют двумерные единицы данных, но метод k-средних позволяет работать с единицами данных, имеющими любое количество измерений.

Этот пример представлен целиком, в виде одного листинга с кодом. Если вы посмотрите на данные об альбомах в листинге 6.15, прежде чем запустить пример, то станет ясно, что самые длинные альбомы Майкл Джексон записал ближе к концу карьеры. Таким образом, вероятно, следует выделить два кластера — более ранние и более поздние альбомы. Альбом *HIStory: Past, Present and Future, Book I* является выбросом и может логически оказаться в отдельном кластере с единственной единицей данных. *Выбросом* называется единица данных, которая выходит за пределы нормальных границ набора данных.

Листинг 6.15. mj.py

```
from __future__ import annotations
from typing import List
```

```
from data_point import DataPoint
from kmeans import KMeans

class Album(DataPoint):
    def __init__(self, name: str, year: int, length: float, tracks: float)
      -> None:
        super().__init__([length, tracks])
        self.name = name
        self.year = year
        self.length = length
        self.tracks = tracks

    def __repr__(self) -> str:
        return f"{self.name}, {self.year}"

if __name__ == "__main__":
    albums: List[Album] = [Album("Got to Be There", 1972, 35.45, 10),
        Album("Ben", 1972, 31.31, 10),
            Album("Music & Me", 1973, 32.09, 10),
        Album("Forever, Michael", 1975, 33.36, 10),
            Album("Off the Wall", 1979, 42.28, 10),
        Album("Thriller", 1982, 42.19, 9),
            Album("Bad", 1987, 48.16, 10), Album("Dangerous",
        1991, 77.03, 14),
            Album("HIStory: Past, Present and Future, Book I",
        1995, 148.58, 30), Album("Invincible", 2001, 77.05, 16)]
    kmeans: KMeans[Album] = KMeans(2, albums)
    clusters: List[KMeans.Cluster] = kmeans.run()
    for index, cluster in enumerate(clusters):
        print(f"Cluster {index} Avg Length {cluster.centroid.dimensions[0]}
        Avg Tracks {cluster.centroid.dimensions[1]}: {cluster.points}\n")
```

Обратите внимание: атрибуты `name` и `year` записываются только для обозначения альбома и не используются при кластеризации. Вот пример вывода результатов:

```
Converged after 1 iterations
Cluster 0 Avg Length -0.5458820039179509 Avg Tracks -0.5009878988684237:
    [Got to Be There, 1972, Ben, 1972, Music & Me, 1973, Forever, Michael,
    1975, Off the Wall, 1979, Thriller, 1982, Bad, 1987]

Cluster 1 Avg Length 1.2737246758085523 Avg Tracks 1.1689717640263217:
    [Dangerous, 1991, HIStory: Past, Present and Future, Book I, 1995,
    Invincible, 2001]
```

Представляют интерес вычисленные средние значения кластеров. Обратите внимание на то, что средние значения являются z-оценками. Три альбома из кластера 1 — последние в творчестве Майкла Джексона — оказались примерно на одно стандартное отклонение длиннее, чем в среднем остальные десять его сольных альбомов.

6.5. Проблемы и расширения кластеризации методом k-средних

Когда кластеризация методом k-средних осуществляется с использованием случайных начальных точек, она может пропустить все полезные точки разделения данных. Это часто требует множества проб и ошибок при выполнении операции. Также определение правильного значения *k* (количества кластеров) сложно и чревато ошибками, если у оператора нет четкого представления о том, сколько должно быть групп данных.

Существуют более сложные версии алгоритма k-средних, в которых можно попытаться сделать обоснованные предположения или выполнить несколько автоматических проб (и допустить ошибки) при определении проблемных переменных. Одним из распространенных вариантов является метод k-средних ++, который пытается решить проблему инициализации, выбирая центроиды не совершенно случайно, а на основе распределения вероятностей расстояния до каждой единицы данных. Еще более удачный вариант для многих приложений — выбор хороших начальных областей для каждого центроида на основе заранее известной информации о данных, иными словами, версия метода k-средних, в которой начальные центроиды выбирает пользователь алгоритма.

Время выполнения кластеризации методом k-средних пропорционально количеству единиц данных, количеству кластеров и количеству измерений для единиц данных. Если имеется множество единиц данных с большим количеством измерений, то этот алгоритм в своей базовой форме может оказаться непригодным. Существуют расширения, которые пытаются не выполнять столько вычислений для каждой единицы данных и каждого центроида, оценивая, действительно ли эта единица данных может переместиться в другой кластер, прежде чем выполнять вычисление. Другой вариант для наборов с многочисленными единицами данных или с большим количеством измерений — просто сделать выборку единиц данных методом k-средних. Это позволит создать приблизительный набор кластеров, которые затем сможет уточнить полный алгоритм k-средних.

Выбросы в наборе данных могут привести к странным результатам работы метода k-средних. Если первоначальный центроид окажется рядом с выбросом, то он может сформировать кластер из одной единицы данных (как с альбомом *HIStory* в примере с Майклом Джексоном). Метод k-средних может работать лучше, если вначале удалить выбросы.

Наконец, среднее значение не всегда считается хорошим показателем для центра. В методе k-медиан серединой кластера является медиана каждого измерения, а в методе k-медоидов — фактическая единица набора данных. Для выбора каждого из этих методов центрирования существуют статистические причины, выходящие за рамки этой книги, но здравый смысл подсказывает, что для сложной задачи, возможно, стоит попробовать каждый из них и выбрать подходящие результаты. Реализации всех этих методов не так уж и различаются.

6.6. Реальные приложения

Кластеризация — это то, чем часто приходится заниматься исследователям данных и аналитикам статистики. Она широко используется для интерпретации данных в различных областях. В частности, кластеризация методом k-средних является полезной технологией в тех случаях, когда о структуре набора данных известно немногое.

Кластеризация — важная методика для анализа данных. Представьте себе полицейское управление, сотрудники которого хотят знать, как расставить полицейские патрули. Или руководителя франшизы предприятия быстрого питания, которому нужно выяснить, где находятся лучшие клиенты, чтобы рассылать рекламные предложения. Или оператора на борту судна, который стремится свести к минимуму аварии, анализируя, когда они происходят и кто является их причиной. Теперь подумайте, как все эти люди могли бы решить свои проблемы с помощью кластеризации.

Кластеризация помогает при распознавании образов. Алгоритм кластеризации позволяет обнаруживать шаблоны, которые пропускает человеческий глаз. Например, кластеризация иногда используется в биологии для выявления групп несоответствующих клеток.

При распознавании изображений кластеризация помогает идентифицировать неочевидные функции. Отдельные пикселы можно рассматривать как единицы данных, их отношение друг к другу определяется расстоянием и разницей цветов.

В политологии кластеризация иногда применяется для поиска избирателей. Может ли политическая партия выявить избирателей, лишенных избирательных прав, сконцентрированных в одном округе, чтобы сосредоточить там свои затраты на избирательную кампанию? Какие проблемы могут вызывать опасения у подобных избирателей?

6.7. Упражнения

1. Создайте функцию, которая импортировала бы данные из CSV-файла в объекты `DataPoint`.
2. Используя внешнюю библиотеку, такую как `matplotlib`, создайте функцию, которая станет строить диаграмму рассеяния с цветовой кодировкой результатов для любого запуска `KMeans` в случае двумерного набора данных.
3. Создайте новый инициализатор для `KMeans`, который будет принимать начальные позиции центроидов извне, а не назначать их случайным образом.
4. Исследуйте и реализуйте алгоритм k-средних++.

7 Простейшие нейронные сети

Достижения в области искусственного интеллекта, о которых мы слышим в наше время — в конце 2010-х годов, обычно имеют отношение к дисциплине, известной как *машинное обучение* (компьютеры изучают некую новую информацию, не получая на то явной команды). Чаще всего эти достижения появились благодаря технологии машинного обучения, известной как *нейронные сети*. Нейронные сети были изобретены пару десятков лет назад, но сейчас переживают что-то вроде возрождения, поскольку усовершенствованное вычислительное оборудование и новые программные технологии, построенные по принципу исследований, позволяют построить новую парадигму, известную как *глубокое обучение*.

Глубокое обучение нашло широкое применение. Оно оказалось полезным в самых разных областях, от алгоритмов хедж-фондов до биоинформатики. Два самых широко известных потребителям способа применения технологии глубокого обучения — это распознавание изображений и распознавание речи. Когда вы спрашиваете своего цифрового помощника о прогнозе погоды или обнаруживаете, что программа обработки фотографий распознала ваше лицо, скорее всего, выполняется некое глубокое обучение.

В методах глубокого обучения используются те же строительные блоки, что и в простейших нейронных сетях. В этой главе мы рассмотрим такие блоки, построив простую нейронную сеть. Это не будет современным решением, но станет основой для понимания глубокого обучения, которое основано на более сложных нейронных сетях, чем построенные нами. На практике при машинном обучении в большинстве случаев нейронные сети не строятся с нуля. Вместо этого

применяются популярные высокооптимизированные готовые платформы, которые и выполняют всю тяжелую работу. Эта глава не поможет вам узнать, как задействовать какую-либо конкретную среду, и сеть, которую мы создадим, будет бесполезной для реального приложения, но она поможет вам понять, как эти платформы работают на низком уровне.

7.1. В основе — биология?

Человеческий мозг — самое невероятное вычислительное устройство из существующих. Он не способен обрабатывать числа так же быстро, как микропроцессор, но с его способностью адаптироваться к незнакомым ситуациям, изучать новые навыки и проявлять творческий подход не сравнится ни одна из известных машин. С момента появления первых компьютеров ученые были заинтересованы в моделировании механизмов мозга. Нервные клетки мозга называются *нейронами*. Нейроны в мозге связаны между собой соединениями, которые называются *синапсами*. Для питания этих сетей нейронов, также известных как *нейронные сети*, по синапсам подается электричество.

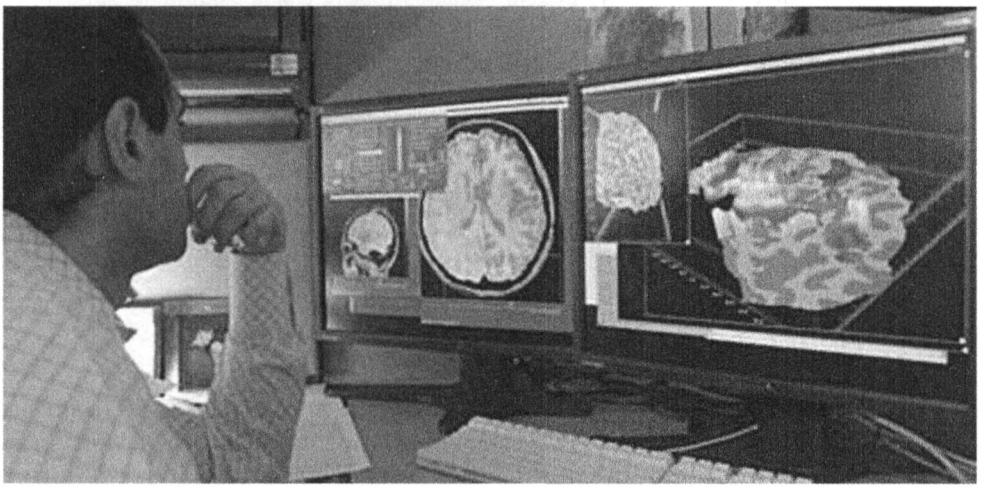
ПРИМЕЧАНИЕ

> Приведенное здесь описание биологических нейронов является грубым упрощением ради аналогии. На самом деле у биологических нейронов есть такие части, как аксоны, дендриты и ядра, о которых вы, возможно, знаете из курса биологии средней школы. А синапсы в действительности являются промежутками между нейронами, в которых выделяются нейротрансмиттеры, обеспечивающие прохождение этих электрических сигналов.

Несмотря на то что ученые выделили части и определили функции нейронов, подробности того, каким образом биологические нейронные сети образуют сложные модели мышления, до сих пор не вполне понятны. Как они обрабатывают информацию? Как формируют оригинальные мысли? Бóльшая часть наших знаний о том, как работает мозг, основана на рассмотрении его на макроуровне. Сканирование мозга методом функциональной магнитно-резонансной томографии (МРТ) показывает, где течет кровь, когда человек выполняет определенную деятельность или обдумывает конкретную мысль (рис. 7.1). Эта и другие макротехнологии могут привести к выводам о том, каким образом связаны различные части мозга, но они не объясняют тайны того, как отдельные нейроны помогают в развитии новых мыслей.

Группы ученых мечутся по всему земному шару, стремясь раскрыть секреты мозга, но вот что следует учесть: в человеческом мозге приблизительно 100 000 000 000 нейронов и каждый из них может быть связан с десятками тысяч других нейронов. Даже на компьютере с миллиардами логических элементов и терабайтами памяти

современные технологии не позволяют построить модель одного человеческого мозга. Очевидно, в обозримом будущем человек по-прежнему будет оставаться самым сложным универсальным объектом изучения.



Общественное достояние США.
Национальный институт психического здоровья

Рис. 7.1. Исследователь изучает изображения головного мозга, полученные посредством МРТ. МРТ-изображения мало что говорят нам о том, как функционируют отдельные нейроны или как организованы нейронные сети

ПРИМЕЧАНИЕ

Универсальная обучающая машина, эквивалентная человеческим способностям, является целью так называемого сильного искусственного интеллекта (ИИ), также известного как общий искусственный интеллект. В данный исторический момент это все еще относится к области научной фантастики. Слабый ИИ — это тип ИИ, который нам встречается ежедневно: компьютеры, интеллектуально решающие конкретные задачи, на которые они были предварительно настроены.

Если биологические нейронные сети все еще не полностью изучены, то каким образом их моделирование стало эффективным вычислительным методом? Действительно, цифровые нейронные сети, известные как *искусственные нейронные сети*, построены под впечатлением от биологических нейронных сетей, но на этом сходство заканчивается. Современные искусственные нейронные сети не претендуют на то, чтобы работать подобно своим биологическим аналогам. На самом деле это было бы невозможно прежде всего потому, что мы не до конца понимаем принципы работы биологических нейронных сетей.

7.2. Искусственные нейронные сети

В этом разделе мы изучим то, что, пожалуй, является самым распространенным типом искусственной нейронной сети, — *сеть прямой связи с обратным распространением*. Именно такую сеть мы позже разработаем. *Прямая связь* означает, что сигнал обычно движется по сети в одном направлении. *Обратное распространение* подразумевает, что мы будем обнаруживать ошибки в конце прохождения каждого сигнала по сети и пытаться распространять исправления этих ошибок в обратном направлении, особенно значительно воздействуя на нейроны, наиболее ответственные за эти ошибки. Существует много других типов искусственных нейронных сетей, и, возможно, эта глава пробудит в вас интерес к дальнейшим исследованиям.

7.2.1. Нейроны

Наименьшей единицей искусственной нейронной сети является нейрон. Он хранит вектор весов, которые представляют собой обычные числа с плавающей точкой. Нейрону передается вектор входных данных, которые также являются обычными числами с плавающей точкой. Нейрон объединяет входные данные с их весами посредством скалярного произведения. Затем запускает *функцию активации* для этого произведения и выдает результат на выходе. Можно считать, что это действие аналогично поведению настоящих нейронов.

Функция активации — это преобразователь выходного сигнала нейрона. Она почти всегда нелинейна, что позволяет нейронным сетям представлять решения нелинейных задач. Если бы не было функций активации, то вся нейронная сеть была бы просто линейным преобразованием. На рис. 7.2 показана работа одного нейрона.

ПРИМЕЧАНИЕ

В этом разделе используется несколько математических терминов, которые вам, возможно, не встречались со времен изучения курсов вычислительной математики или линейной алгебры. Объяснение того, что такое векторы или скалярное произведение, выходит за рамки этой главы, но вы, вероятно, все равно получите представление о том, что делает нейронная сеть, просто читая главу, даже если не вполне понимаете математическую часть. Позже в этой главе будут применены некоторые численные методы, такие как вычисление обычных и частных производных, но даже если вы не понимаете математику, то все равно сможете проследить код. В сущности, здесь не объясняется, как можно получить формулы, по которым выполняются вычисления. Вместо этого мы сосредоточимся на использовании самих вычислений.

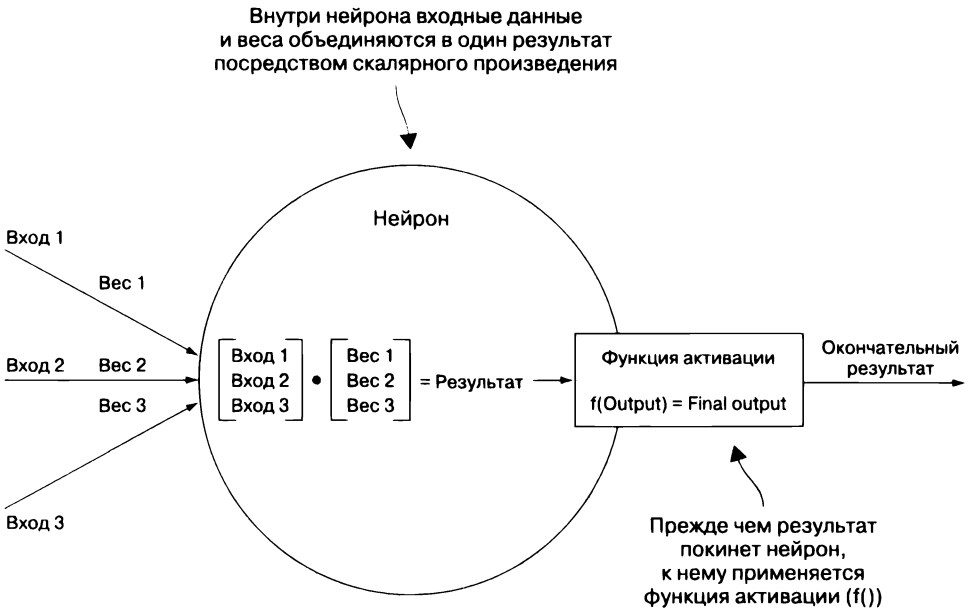


Рис. 7.2. В нейроне его веса объединяются с входными сигналами для получения результирующего сигнала, который модифицируется функцией активации

7.2.2. Слои

В типичной искусственной нейронной сети с прямой связью нейроны образуют слои. Каждый слой состоит из определенного количества нейронов, выстроенных в строку или столбец в зависимости от диаграммы (варианты эквивалентны). В сети с прямой связью, которую мы будем создавать, сигналы всегда передаются в одном направлении от одного слоя к другому. Нейроны в каждом слое посылают выходной сигнал, который является входным для нейронов следующего слоя. Каждый нейрон в каждом слое связан с каждым нейроном в следующем слое.

Первый слой называется *входным*, он получает сигналы от некоторого внешнего объекта. Последний слой известен как *выходной*, и его выходные сигналы обычно должен интерпретировать внешний субъект, чтобы получить осмысленный результат. Слои, расположенные между входным и выходным слоями, называются *скрытыми*. В простых нейронных сетях, таких как та, которую мы будем строить в этой главе, есть только один скрытый слой, но в сетях глубокого обучения слоев много. На рис. 7.3 показаны слои, образующие простую сеть. Обратите внимание на то, как выходные сигналы одного слоя используются в качестве входных для каждого нейрона следующего слоя.

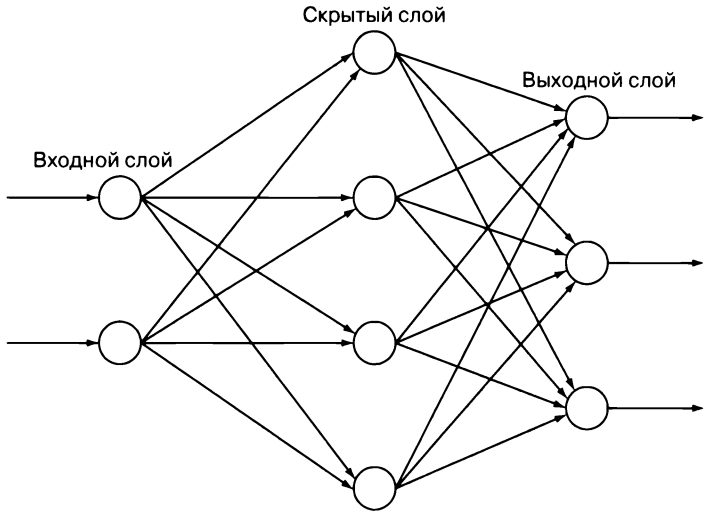


Рис. 7.3. Простая нейронная сеть с одним входным слоем, состоящим из двух нейронов, одним скрытым слоем из четырех нейронов и одним выходным слоем из трех нейронов. Количество нейронов в каждом слое произвольно

Эти слои просто манипулируют числами с плавающей точкой. Входные сигналы для входного слоя и выходные сигналы выходного слоя являются числами с плавающей запятой.

Очевидно, что эти числа должны представлять что-то значимое. Представим сеть, разработанную для классификации небольших черно-белых изображений животных. Возможно, ее входной слой имеет 100 нейронов, представляющих интенсивность оттенков серого для каждого пиксела в изображении животного размером 10 × 10 пикселов, а выходной слой имеет пять нейронов, представляющих вероятность того, что это изображение млекопитающего, рептилии, амфибии, рыбы или птицы. Окончательная классификация может быть определена выходным нейроном с самым высоким выходным сигналом с плавающей точкой. Если бы выходные числа были равны 0,24, 0,65, 0,70, 0,12 и 0,21, то изображение можно было бы классифицировать как амфибию.

7.2.3. Обратное распространение

Последняя и самая сложная часть головоломки — это обратное распространение. Обратное распространение позволяет обнаружить ошибку в выходных данных нейронной сети и задействовать ее для изменения весов нейронов. Сильнее всего изменяются нейроны, больше других ответственные за ошибку. Но откуда берется ошибка? Как ее распознать? Ошибки возникают на этапе использования нейронной сети, называемом *обучением*.

СОВЕТ

В этом разделе обычным языком описаны этапы, которые можно представить несколькими математическими формулами. На рисунках приводятся псевдоформулы (без принятых в математике обозначений). Благодаря такому подходу формулы становятся удобочитаемыми для тех, кто не связан с математикой или не использует математические обозначения. Если вас интересуют более формальная запись и вывод формул, ознакомьтесь с главой 18 книги «Искусственный интеллект» Рассела и Норвига.

Прежде чем с нейронной сетью можно будет работать, ее, как правило, необходимо обучить. Но мы должны знать правильные выходные данные для определенных входных данных, чтобы можно было, используя разницу между ожидаемыми и фактическими выходными данными, находить ошибки и изменять веса. Другими словами, нейронные сети ничего не знают, пока им не сообщат правильные ответы для определенного набора входных данных, чтобы потом они могли подготовиться к другим входным данным. Обратное распространение происходит только во время обучения.

ПРИМЕЧАНИЕ

Поскольку большинство нейронных сетей нуждается в обучении, их относят к типу контролируемого машинного обучения. Напомню: как было показано в главе 6, алгоритм k-средних и другие кластерные алгоритмы считаются формой неконтролируемого машинного обучения, поскольку после их запуска внешнее вмешательство не требуется. Существуют и другие типы нейронных сетей, отличные от описанных в этой главе, которые не требуют предварительной подготовки и считаются формой неконтролируемого обучения.

Первым шагом при обратном распространении является вычисление ошибки между выходным сигналом нейронной сети для некоторого входного сигнала и ожидаемым выходным сигналом. Эта ошибка распространяется на все нейроны в выходном слое. (Для каждого нейрона есть ожидаемый и фактический выходной сигнал.) Затем к тому, что было бы выходным сигналом нейрона до того, как была задействована его функция активации, применяется производная функции активации выходного нейрона. (Мы кэшируем результат его функции перед активацией.) Этот результат умножают на ошибку нейрона, чтобы найти его *дельту*. Формула вычисления дельты использует частную производную (нахождение этой производной выходит за рамки данной книги). Главное, что мы получаем в результате, — то, за какое количество ошибок отвечает каждый выходной нейрон. Диаграмма вычисления показана на рис. 7.4.

Затем необходимо вычислить дельты для всех нейронов скрытого слоя (слоев) данной сети. Нужно определить, насколько каждый нейрон ответственен за неправильный выходной сигнал в выходном слое. Дельты выходного слоя

используются для вычисления дельт в предыдущем скрытом слое. Для каждого предыдущего слоя дельты вычисляются определением скалярного произведения весов следующего слоя по отношению к конкретному рассматриваемому нейрону и уже вычисленных дельт в следующем слое. Чтобы получить дельту нейрона, это значение умножается на производную от функции активации, применяемой к последнему выходному сигналу нейрона, кэшированному перед задействованием функции активации. Эта формула также получена с помощью частной производной, подробнее о которой вы можете прочитать в более специализированных математических текстах.

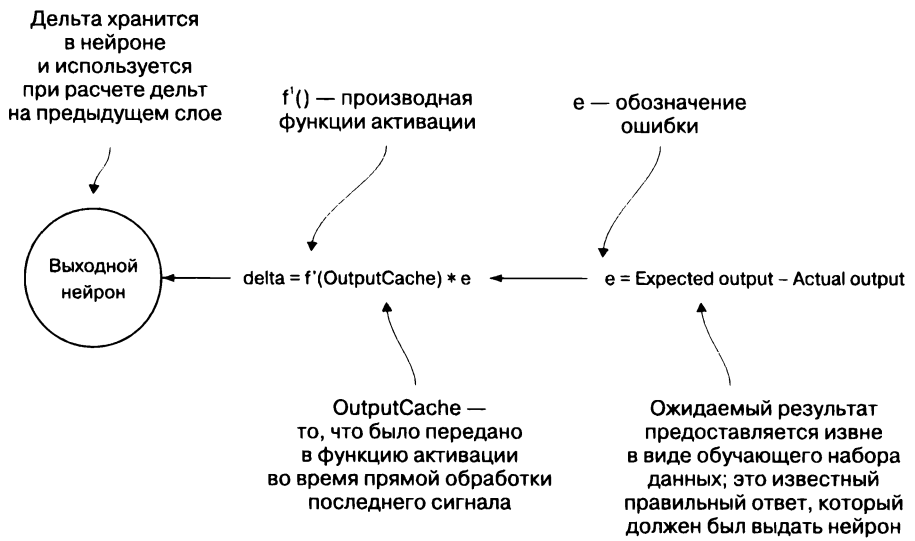


Рис. 7.4. Механизм, с помощью которого вычисляется дельта выходного нейрона на этапе обучения посредством обратного распространения

На рис. 7.5 показан фактический расчет дельт для нейронов в скрытых слоях. В сети с несколькими скрытыми слоями нейроны О1, О2 и О3 могут быть нейронами следующего скрытого слоя, а не нейронами выходного слоя.

И последнее, но самое главное: все веса для каждого нейрона в сети должны быть обновлены путем умножения последнего входного значения каждого отдельного веса на дельту нейрона и нечто, называемое *скоростью обучения*, и прибавления этого значения к существующему весу. Этот метод изменения веса нейрона называется *градиентным спуском*. Он похож на спуск с холма, где холм — это график функции ошибки нейрона, к точке минимальной ошибки. Дельта — это направление, в котором мы хотим спускаться, а скорость обучения определяет то, как быстро будем это делать. Трудно определить хорошую скорость обучения для неизвестной задачи без проб и ошибок. На рис. 7.6 показано, как изменяются веса в скрытом и выходном слоях.

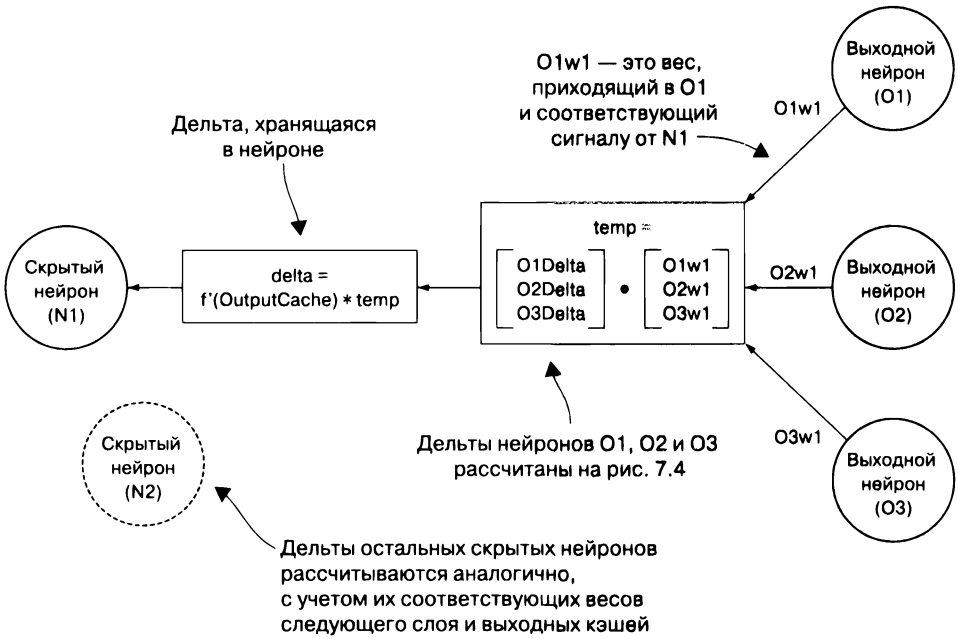


Рис. 7.5. Вычисление дельты для нейрона в скрытом слое

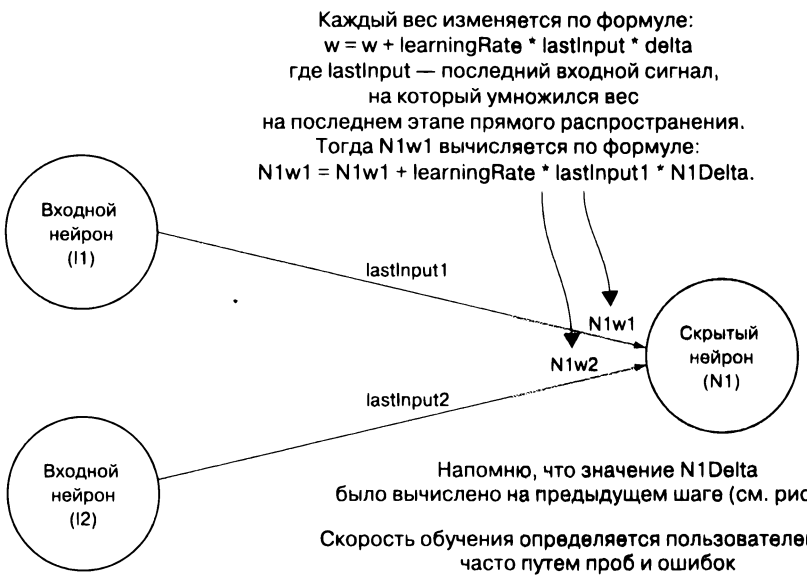


Рис. 7.6. Веса каждого скрытого слоя и нейрона выходного слоя обновляются с использованием дельт, рассчитанных на предыдущих шагах, предыдущих весов, предыдущих входных данных и заданной пользователем скорости обучения

После изменения весов нейронная сеть готова к повторному обучению с другим набором входных и ожидаемых выходных данных. Процесс повторяется до тех пор, пока пользователь нейронной сети не сочтет, что сеть хорошо обучена. Это можно определить, проверив сеть по входным данным с известными правильными выходными данными.

Обратное распространение — сложный процесс. Не беспокойтесь, если вы еще не поняли все его детали. Объяснения, представленного в этом разделе, может оказаться недостаточно. В идеале реализация обратного распространения выведет ваше понимание на новый уровень. При реализации нашей нейронной сети и обратного распространения помните главное: обратное распространение — это способ корректировки каждого отдельного веса в сети в соответствии с тем, насколько верно он определяет неправильные выходные данные.

7.2.4. Ситуация в целом

В этом разделе мы рассмотрели множество вопросов. Даже если вам все еще непонятны некоторые детали, важно помнить об основных свойствах сети прямой связи с обратным распространением.

1. Сигналы (числа с плавающей точкой) проходят через нейроны, расположенные по слоям, в одном направлении. Каждый нейрон каждого слоя связан с каждым нейроном следующего слоя.
2. Каждый нейрон (кроме нейронов входного слоя) обрабатывает получаемые сигналы, комбинируя их с весами (которые также являются числами с плавающей точкой) и применяя к ним функцию активации.
3. Во время процесса, называемого обучением, результаты сети сравниваются с ожидаемыми результатами, чтобы вычислить ошибки.
4. Ошибки распространяются по сети в обратном направлении (туда, откуда они пришли) для изменения весов, чтобы в результате они с большей вероятностью приводили к выработке правильных выходных данных.

Кроме описанных, существует множество других методов обучения нейронных сетей. Есть также много иных способов передачи сигналов внутри нейронных сетей. Описанный здесь метод, который мы и будем реализовывать, является лишь очень распространенной формой, которая служит достойным введением в предмет. В приложении Б приводятся дополнительные источники для получения расширенной информации по нейронным сетям (включая другие типы сетей) и по математике.

7.3. Предварительные замечания

В нейронных сетях применяются математические механизмы, которые требуют множества операций с плавающей точкой. Прежде чем мы разработаем фактические структуры простой нейронной сети, нам понадобятся некоторые математи-

ческие примитивы. Эти простые примитивы широко используются в приводимом далее коде, поэтому, если вы сможете найти способы ускорения их работы, это существенно повысит производительность нейронной сети.

ПРЕДУПРЕЖДЕНИЕ

Сложность кода в данной главе, вероятно, выше, чем в любой другой главе книги. Будет много наслоений, а реальные результаты мы увидим лишь в самом конце. Есть множество ресурсов, посвященных нейронным сетям, которые помогут вам создать такую сеть, написав всего нескольких строк кода, но цель этого примера — изучение механизма и того, как организовать взаимодействие различных компонентов в удобочитаемой и расширяемой форме. Пусть даже код получится довольно длинным и многословным.

7.3.1. Скалярное произведение

Как мы помним, скалярное произведение требуется и на этапе прямой связи, и на этапе обратного распространения. К счастью, скалярное произведение легко реализовать с помощью встроенных в Python функций `zip()` и `sum()`. Будем хранить вспомогательные функции в файле util.py (листинг 7.1).

Листинг 7.1. util.py

```
from typing import List
from math import exp

# скалярное произведение двух векторов
def dot_product(xs: List[float], ys: List[float]) -> float:
    return sum(x * y for x, y in zip(xs, ys))
```

7.3.2. Функция активации

Напомню, что функция активации преобразует выходные данные нейрона перед тем, как сигнал перейдет на следующий слой (см. рис. 7.2). Функция активации имеет две цели: она позволяет нейронной сети представлять решения, которые не являются всего лишь линейными преобразованиями (если сама функция активации является чем-то большим, нежели линейное преобразование), и может делать так, чтобы выходные данные каждого нейрона не выходили за пределы заданного диапазона. Функция активации должна иметь вычислимую производную, чтобы ее можно было использовать для обратного распространения.

Популярным множеством функций активации являются *сигмоидные* функции. Одна из самых широко распространенных сигмоидных функций, часто называемая просто сигмоидной функцией, показана на рис. 7.7 (обозначена $S(x)$). Там же приводятся ее уравнение и производная $S'(x)$. Результатом сигмоидной функции всегда будет значение в диапазоне 0... 1. Как мы вскоре узнаем, то, что это число

всегда находится в пределах от 0 до 1, полезно для сети. В листинге 7.2 формулы, показанные на этом рисунке, реализованы в виде кода.

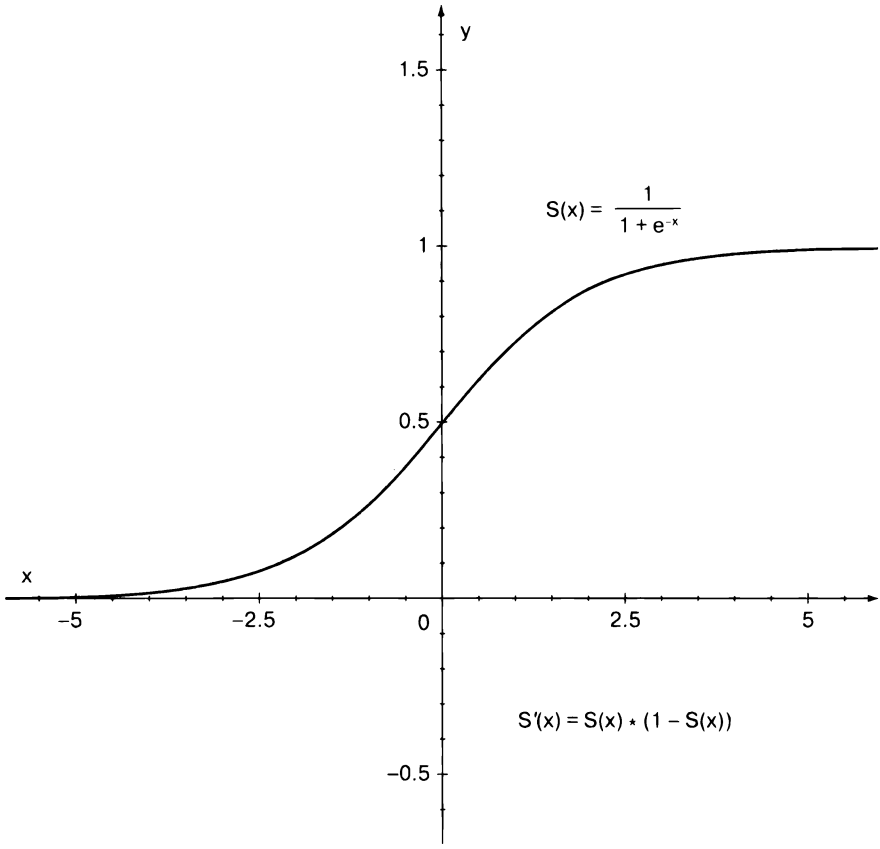


Рис. 7.7. Сигмоидная функция активации S(x) всегда возвращает значение в диапазоне 0... 1. Обратите внимание на то, что ее производную S'(x) также легко вычислить

Существуют и другие функции активации, но мы будем использовать сигмоидную функцию. Вот прямое преобразование формул, показанных на рис. 7.7, в код.

Листинг 7.2. util.py (продолжение)

```
# классическая сигмоидная функция активации
def sigmoid(x: float) -> float:
    return 1.0 / (1.0 + exp(-x))

def derivative_sigmoid(x: float) -> float:
    sig: float = sigmoid(x)
    return sig * (1 - sig)
```

7.4. Построение сети

Создадим классы для моделирования всех трех организационных единиц сети: нейронов, слоев и самой сети. Для простоты начнем с самого маленького компонента — с нейронов, перейдем к центральному организующему компоненту — к слоям, а затем к самому большому — ко всей сети. Переходя от наименьшего компонента к наибольшему, мы инкапсулируем предыдущий уровень. Нейроны знают только о самих себе, слои — о нейронах, которые они содержат, и других слоях, а сеть — обо всех слоях.

ПРИМЕЧАНИЕ

В этой главе приводится много длинных строк кода, которые не помещаются по ширине на печатных страницах книги. Я настоятельно рекомендую при чтении загрузить прилагаемый к ней исходный код из репозитория исходных кодов книги и отслеживать его на экране компьютера: https://github.com/davecom/ClassicComputerScienceProblemsInPython.

7.4.1. Реализация нейронов

Начнем с нейронов. Каждый нейрон должен хранить много элементов состояния, включая вес, дельту, скорость обучения, кэш последних выходных данных, функцию активации, а также производную от нее. Некоторые из этих элементов эффективнее было бы хранить в слое (в будущем классе `Layer`), но они включены в представленный далее класс `Neuron` из соображений наглядности (листинг 7.3).

Листинг 7.3. neuron.py

```
from typing import List, Callable
from util import dot_product

class Neuron:
    def __init__(self, weights: List[float], learning_rate: float,
        activation_function: Callable[[float], float], derivative_activation_
        function: Callable[[float], float]) -> None:
            self.weights: List[float] = weights
            self.activation_function: Callable[[float], float] = activation_
              function
            self.derivative_activation_function: Callable[[float], float] =
              derivative_activation_function
            self.learning_rate: float = learning_rate
            self.output_cache: float = 0.0
            self.delta: float = 0.0

    def output(self, inputs: List[float]) -> float:
        self.output_cache = dot_product(inputs, self.weights)
        return self.activation_function(self.output_cache)
```

Большинство этих параметров инициализируется в методе `__init__()`. Поскольку `delta` и `output_cache` неизвестны при первом создании `Neuron`, они просто инициализируются нулем. Все переменные нейрона изменяемые. На протяжении его жизни (по мере того как мы будем его применять) значения этих переменных могут никогда не измениться, но все же есть причина сделать их изменяемыми: гибкость. Если класс `Neuron` будет использоваться для других типов нейронных сетей, то, возможно, некоторые из этих значений изменятся в процессе выполнения программы. Существуют нейронные сети, которые изменяют скорость обучения по мере приближения к решению и автоматически пробуют разные функции активации. Поэтому мы стремимся сделать класс `Neuron` максимально гибким, чтобы он был пригоден для других приложений нейронных сетей.

Кроме `__init__()`, у класса есть только один метод — `output()`. `output()` принимает входные сигналы (входные данные), поступающие в нейрон, и применяет к ним формулу, рассмотренную ранее в этой главе (см. рис. 7.2). Входные сигналы объединяются с весами посредством скалярного произведения, и результат кэшируется в `output_cache`. Напомню, что это значение, полученное до того, как была задействована функция активации, используется для вычисления дельты (см. раздел об обратном распространении). Наконец, прежде чем сигнал будет отправлен на следующий слой (будучи возвращенным из `output()`), к нему применяется функция активации.

Вот и все! Отдельный нейрон в этой сети довольно прост. Он не может сделать ничего другого, кроме как принять входной сигнал, преобразовать его и передать для дальнейшей обработки. Нейрон поддерживает несколько элементов состояния, которые используются другими классами.

7.4.2. Реализация слоев

Слой в нашей сети должен поддерживать три элемента состояния: его нейроны, предшествующий слой и выходной кэш. Выходной кэш похож на кэш нейрона, но на один слой выше. Он кэширует выходные данные каждого нейрона в данном слое после применения функций активации.

В момент создания слоя основной задачей является инициализация его нейронов. Поэтому методу `__init__()` класса `Layer` нужно знать, сколько нейронов требуется инициализировать, какими должны быть их функции активации и какова скорость обучения. В нашей простой сети у всех нейронов слоя функция активации и скорость обучения одинаковы (листинг 7.4).

Листинг 7.4. layer.py

```
from __future__ import annotations
from typing import List, Callable, Optional
from random import random
from neuron import Neuron
```

```
from util import dot_product

class Layer:
    def __init__(self, previous_layer: Optional[Layer], num_neurons: int,
        learning_rate: float, activation_function: Callable[[float], float],
        derivative_activation_function: Callable[[float], float]) -> None:
            self.previous_layer: Optional[Layer] = previous_layer
            self.neurons: List[Neuron] = []
            # дальше может идти одно большое списковое включение
            for i in range(num_neurons):
                if previous_layer is None:
                    random_weights: List[float] = []
                else:
                    random_weights = [random() for _ in range(len(previous_
                      layer.neurons))]
                neuron: Neuron = Neuron(random_weights, learning_rate,
                  activation_function, derivative_activation_function)
                self.neurons.append(neuron)
            self.output_cache: List[float] =
              [0.0 for _ in range(num_neurons)]
```

По мере того как сигналы передаются через сеть, их должен обрабатывать каждый нейрон `Layer`. (Помните, что каждый нейрон в слое получает сигналы от каждого нейрона предыдущего слоя.) Именно это делает метод `output()`. Он также возвращает результат обработки для передачи по сети на следующий слой и кэширует выходные данные. Если предыдущего слоя нет, значит, данный слой является входным и он просто передает сигналы на следующий слой (листинг 7.5).

Листинг 7.5. layer.py (продолжение)

```
def outputs(self, inputs: List[float]) -> List[float]:
    if self.previous_layer is None:
        self.output_cache = inputs
    else:
        self.output_cache = [n.output(inputs) for n in self.neurons]
    return self.output_cache
```

Существует два типа дельт для вычисления в обратном распространении: дельты для нейронов в выходном слое и дельты для нейронов в скрытых слоях. Их формулы показаны на рис. 7.4 и 7.5, и следующие два метода представляют собой механические переводы этих формул на Python (листинг 7.6). Впоследствии эти методы будут вызываться сетью во время обратного распространения.

Листинг 7.6. layer.py (продолжение)

```
# вызывается только для выходного слоя
def calculate_deltas_for_output_layer(self, expected: List[float]) -> None:
    for n in range(len(self.neurons)):
```

```
        self.neurons[n].delta = self.neurons[n].derivative_activation_
          function(self.neurons[n].output_cache) * (expected[n] -
          self.output_cache[n])

# не вызывается для выходного слоя
def calculate_deltas_for_hidden_layer(self, next_layer: Layer) -> None:
    for index, neuron in enumerate(self.neurons):
        next_weights: List[float] = [n.weights[index] for n in
          next_layer.neurons]
        next_deltas: List[float] = [n.delta for n in next_layer.neurons]
        sum_weights_and_deltas: float = dot_product(next_weights, next_deltas)
        neuron.delta = neuron.derivative_activation_function(neuron.output_
          cache) * sum_weights_and_deltas
```

7.4.3. Реализация сети

Сама сеть хранит только один элемент состояния — слои, которыми она управляет. Класс `Network` отвечает за инициализацию составляющих его слоев.

Метод `__init__()` принимает список элементов типа `int`, описывающий структуру сети. Например, список `[2, 4, 3]` описывает сеть, имеющую два нейрона во входном слое, четыре нейрона — в скрытом и три нейрона — в выходном. Работая над этой простой сетью, предполагаем, что все слои сети используют одну и ту же функцию активации для своих нейронов и имеют одинаковую скорость обучения (листинг 7.7).

Листинг 7.7. network.py

```
from __future__ import annotations
from typing import List, Callable, TypeVar, Tuple
from functools import reduce
from layer import Layer
from util import sigmoid, derivative_sigmoid

T = TypeVar('T') # тип выходных данных в интерпретации нейронной сети

class Network:
    def __init__(self, layer_structure: List[int], learning_rate: float,
        activation_function: Callable[[float], float] = sigmoid, derivative_
        activation_function: Callable[[float], float] = derivative_sigmoid)
          -> None:
            if len(layer_structure) < 3:
                raise ValueError("Error: Should be at least 3 layers (1 input,
                1 hidden, 1 output)")
            self.layers: List[Layer] = []
            # входной слой
            input_layer: Layer = Layer(None, layer_structure[0],
              learning_rate, activation_function,
```

```
            derivative_activation_function)
            self.layers.append(input_layer)
        # скрытые слои и выходной слой
        for previous, num_neurons in enumerate(layer_structure[1::]):
            next_layer = Layer(self.layers[previous], num_neurons,
              learning_rate, activation_function,
              derivative_activation_function)
            self.layers.append(next_layer)
```

Выходные данные нейронной сети — это результат обработки сигналов, проходящих через все ее слои. Обратите внимание на то, как компактно метод `reduce()` используется в `output()` для многократной передачи сигналов между слоями по всей сети (листинг 7.8).

Листинг 7.8. network.py (продолжение)

```
# Помещает входные данные на первый слой, затем выводит их
# с первого слоя и подает на второй слой в качестве входных данных,
# со второго — на третий и т. д.
def outputs(self, input: List[float]) -> List[float]:
    return reduce(lambda inputs, layer: layer.outputs(inputs), self.layers,
        input)
```

Метод `backpropagate()` отвечает за вычисление дельт для каждого нейрона в сети. В этом методе последовательно задействуются методы `Layer calculate_deltas_for_output_layer()` и `calculate_deltas_for_hidden_layer()`. (Напомню, что при обратном распространении дельты вычисляются в обратном порядке.) Функция `backpropagate()` передает ожидаемые значения выходных данных для заданного набора входных данных в функцию `calc_deltas_for_output_layer()`. Эта функция использует ожидаемые значения, чтобы найти ошибку, с помощью которой вычисляется дельта (листинг 7.9).

Листинг 7.9. network.py (продолжение)

```
# Определяет изменения каждого нейрона на основании ошибок
# выходных данных по сравнению с ожидаемым выходом
def backpropagate(self, expected: List[float]) -> None:
    # вычисление дельты для нейронов выходного слоя
    last_layer: int = len(self.layers) - 1
    self.layers[last_layer].calculate_deltas_for_output_layer(expected)
    # вычисление дельты для скрытых слоев в обратном порядке
    for l in range(last_layer - 1, 0, -1):
        self.layers[l].calculate_deltas_for_hidden_layer(self.layers[l + 1])
```

Функция `backpropagate()` отвечает за вычисление всех дельт, но не изменяет веса элементов сети. Для этого после `backpropagate()` должна вызываться функция `update_weights()`, поскольку изменение веса зависит от дельт (листинг 7.10). Этот метод вытекает непосредственно из формулы, представленной на рис. 7.6.

Листинг 7.10. network.py (продолжение)

```
# Сама функция backpropagate() не изменяет веса
# Функция update_weights использует дельты, вычисленные в backpropagate(),
# чтобы действительно изменить веса
def update_weights(self) -> None:
    for layer in self.layers[1:]: # пропустить входной слой
        for neuron in layer.neurons:
            for w in range(len(neuron.weights)):
                neuron.weights[w] = neuron.weights[w] +
                  (neuron.learning_rate *
                  (layer.previous_layer.output_cache[w]) * neuron.delta)
```

Веса нейронов изменяются в конце каждого этапа обучения. Для этого в сеть должны быть поданы обучающие наборы данных (входные данные и ожидаемые результаты). Метод `train()` принимает список списков входных данных и список списков ожидаемых выходных данных.

Каждый набор входных данных пропускается через сеть, после чего ее веса обновляются посредством вызова `backpropagate()` для ожидаемого результата и последующего вызова `update_weights()`. Попробуйте добавить сюда код, который позволит вывести на печать частоту ошибок, когда через сеть проходит обучающий набор данных. Так вы сможете увидеть как постепенно уменьшается частота ошибок сети по мере ее продвижения вниз по склону в процессе градиентного спуска (листинг 7.11).

Листинг 7.11. network.py (продолжение)

```
# Функция train() использует результаты выполнения функции outputs()
# для нескольких входных данных, сравнивает их
# с ожидаемыми результатами и передает полученное
# в backpropagate() и update_weights()
def train(self, inputs: List[List[float]], expecteds: List[List[float]]) ->
None:
    for location, xs in enumerate(inputs):
        ys: List[float] = expecteds[location]
        outs: List[float] = self.outputs(xs)
        self.backpropagate(ys)
        self.update_weights()
```

Наконец, после обучения сеть необходимо протестировать. Функция `validate()` принимает входные данные и ожидаемые выходные данные так же, как `train()`, но, в отличие от `train()`, использует их не для обучения, а для вычисления процента точности, так как предполагается, что сеть уже обучена. Функция `validate()` принимает также функцию `interpret_output()`, с помощью которой интерпретируются выходные данные нейронной сети для сравнения с ожидаемыми выходными данными. (Возможно, ожидаемый вывод — это не набор чисел с плавающей запятой, а строка типа `"Amphibian"`.) Функция `interpret_output()` должна принимать числа

с плавающей точкой, полученные в сети, и преобразовывать их в нечто сопоставимое с ожидаемыми выходными данными. Это специальная функция, созданная для конкретного набора данных. Функция `validate()` возвращает количество правильных классификаций, общее количество протестированных образцов и процент правильных классификаций (листинг 7.12).

Листинг 7.12. network.py (продолжение)

```
# Для параметризованных результатов, которые требуют классификации,
# эта функция возвращает правильное количество попыток
# и процентное отношение по сравнению с общим количеством
def validate(self, inputs: List[List[float]], expecteds: List[T], interpret_
      output: Callable[[List[float]], T]) -> Tuple[int, int, float]:
    correct: int = 0
    for input, expected in zip(inputs, expecteds):
        result: T = interpret_output(self.outputs(input))
        if result == expected:
            correct += 1
    percentage: float = correct / len(inputs)
    return correct, len(inputs), percentage
```

Нейронная сеть готова! Ее можно протестировать на нескольких настоящих задачах. Построенная архитектура достаточно универсальна для того, чтобы с ее помощью можно было решать различные задачи, но мы сосредоточимся на популярной задаче — классификации.

7.5. Задачи классификации

В главе 6 мы классифицировали набор данных посредством кластеризации с помощью метода k-средних, не используя заранее известных представлений о том, к какой категории принадлежит каждый элемент данных. При кластеризации мы знаем, что хотим найти категории данных, но предварительно не знаем, что это за категории. При решении задачи классификации мы тоже пытаемся классифицировать набор данных, но в этом случае существуют заранее определенные категории. Например, если бы мы пытались классифицировать набор изображений животных, то могли бы перед этим выбрать такие категории, как млекопитающие, рептилии, амфибии, рыбы и птицы.

Существует множество методов машинного обучения, которые можно задействовать в задачах классификации. Возможно, вы слышали о методах опорных векторов, деревьях принятия решений и наивных классификаторах Байеса. Существуют и другие методы. В последнее время нейронные сети стали широко применяться в области классификации. Они требуют больше вычислительных ресурсов, чем некоторые другие алгоритмы классификации, но их способность классифицировать, казалось бы, произвольные виды данных делает их эффективной

технологией. Классификаторы нейронных сетей лежат в основе многих интересных методов классификаций изображений, которые применяются в современном программном обеспечении для обработки фотографий.

Почему вновь возник интерес к использованию нейронных сетей для задач классификации? Аппаратное обеспечение стало достаточно быстрым для того, чтобы окупилась необходимость дополнительных вычислений по сравнению с другими алгоритмами.

7.5.1. Нормализация данных

Наборы данных, с которыми мы предполагаем работать, обычно требуют некоторой очистки, прежде чем их можно будет ввести в алгоритмы. Очистка может означать удаление посторонних символов и дубликатов, исправление ошибок и другие вспомогательные операции. Вид очистки, которую нужно выполнить для двух наборов данных, с которыми предстоит работать, — это нормализация. В главе 6 она сделана с помощью метода `zscore_normalize()` в классе `KMeans`. Нормализация — это преобразование атрибутов, записанных в разных масштабах, к единому масштабу.

Благодаря сигмоидной функции активации каждый нейрон в сети выводит значения в диапазоне 0... 1. Логично, что шкала от 0 до 1 будет иметь смысл и для атрибутов в нашем наборе входных данных. Преобразовать шкалу из некоторого диапазона в диапазон 0... 1 не составляет труда. Для любого значения `V` в определенном диапазоне атрибутов с максимальным `max` и минимальным `min` значениями формула имеет вид `newV = (oldV – min)/(max - min)`. Эта операция называется *масштабированием объектов*. Далее представлена реализация формулы на Python, добавленная в файл `util.py` (листинг 7.13).

Листинг 7.13. util.py (продолжение)

```
# Будем считать, что все строки одинаковой длины,
# а каждый столбец масштабирован в диапазоне 0...1
def normalize_by_feature_scaling(dataset: List[List[float]]) -> None:
    for col_num in range(len(dataset[0])):
        column: List[float] = [row[col_num] for row in dataset]
        maximum = max(column)
        minimum = min(column)
        for row_num in range(len(dataset)):
            dataset[row_num][col_num] = (dataset[row_num][col_num] -
              minimum) / (maximum - minimum)
```

Обратите внимание на параметр `dataset`. Он указывает на список списков, который будет изменен в этой функции. Другими словами, функция `normalize_by_feature_scaling()` получает не копию набора данных, а ссылку на исходный набор данных. В этом случае мы хотим внести изменения в значение, а не возвратить его измененную копию.

Обратите также внимание: программа предполагает, что наборы данных являются двумерными списками данных типа `float`.

7.5.2. Классический набор данных радужной оболочки

Подобно существованию классических задач информатики, существуют и классические наборы данных для машинного обучения. Они используются для проверки новых методов и сравнения их с существующими. А также служат хорошей отправной точкой для тех, кто изучает машинное обучение. Возможно, самым известным является набор данных об ирисах. Собранный в 1930-х годах, этот набор данных состоит из 150 образцов ирисов (красивые цветы), разделенных на три вида по 50 растений в каждом. Для всех растений измерены четыре атрибута: длина чашелистика, ширина чашелистика, длина лепестка и ширина лепестка.

Стоит отметить: нейронной сети безразлично, что представляют собой атрибуты. С точки зрения важности ее модель обучения не делает различий между длиной чашелистика и длиной лепестка. Если такое различие должно быть сделано, то пользователю нейронной сети следует выполнить соответствующую корректировку.

В репозитории исходного кода, прилагаемого к этой книге, содержится файл с данными, разделенными запятыми (в формате *CSV*), который содержит набор данных об ирисах[1]. Этот набор получен из репозитория машинного обучения UCI Калифорнийского университета: *Lichman M.* UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science, 2013, http://archive.ics.uci.edu/ml. CSV-файл — это просто текстовый файл со значениями, разделенными запятыми. Это общий формат обмена табличными данными, включая электронные таблицы.

Вот несколько строк из файла `iris.csv`:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

Каждая строка описывает отдельную единицу данных. Четыре числа представляют собой четыре атрибута (длина и ширина чашелистика, длина и ширина лепестка), которые, с нашей точки зрения, являются произвольными в отношении того, что они на самом деле представляют. Название в конце каждой строки соответствует определенному виду ириса. Все пять строк относятся к одному и тому же виду — данный образец был взят из верхней части файла, и три вида сгруппированы вместе по 50 строк в каждом.

Для того чтобы прочитать с диска CSV-файл, используем несколько функций из стандартной библиотеки Python. Модуль `csv` поможет прочитать данные в структурированном виде. Встроенная функция `open()` создает объект файла, который передается в `csv.reader()`. Помимо этих нескольких строк остальная часть

[1] Этот репозиторий доступен на GitHub по адресу https://github.com/davecom/ClassicComputerScienceProblemsInPython.

листинга 7.14 просто переупорядочивает данные из CSV-файла, чтобы подготовить их к применению нашей сетью для обучения и проверки.

Листинг 7.14. iris_test.py

```
import csv
from typing import List
from util import normalize_by_feature_scaling
from network import Network
from random import shuffle

if __name__ == "__main__":
    iris_parameters: List[List[float]] = []
    iris_classifications: List[List[float]] = []
    iris_species: List[str] = []
    with open('iris.csv', mode='r') as iris_file:
        irises: List = list(csv.reader(iris_file))
        shuffle(irises) # получаем наши строки данных в случайном порядке
        for iris in irises:
            parameters: List[float] = [float(n) for n in iris[0:4]]
            iris_parameters.append(parameters)
            species: str = iris[4]
            if species == "Iris-setosa":
                iris_classifications.append([1.0, 0.0, 0.0])
            elif species == "Iris-versicolor":
                iris_classifications.append([0.0, 1.0, 0.0])
            else:
                iris_classifications.append([0.0, 0.0, 1.0])
            iris_species.append(species)
    normalize_by_feature_scaling(iris_parameters)
```

`iris_parameters` представляет собой коллекцию из четырех атрибутов для каждого образца. Используем эту коллекцию для классификации каждого ириса. `iris_classification` — это фактическая классификация любого образца. В нашей нейронной сети будет три выходных нейрона, каждый из которых соответствует одному из возможных видов. Например, окончательный набор выходных данных `[0,9, 0,3, 0,1]` будет означать классификацию `Iris-setosa`, поскольку этому виду соответствует первый нейрон и он выдал наибольшее число.

Мы уже знаем правильные ответы для обучения, поэтому у каждого ириса есть готовый ответ. Для цветка, который должен иметь тип `Iris-setosa`, запись в `iris_classification` будет иметь вид `[1.0, 0.0, 0.0]`. Эти значения станут использоваться для расчета ошибки после каждого этапа обучения. В `iris_species` хранятся непосредственные соответствия классификаций для каждого цветка на английском языке. Ирис вида iris-setosa будет отмечен в наборе данных как `"Iris-setosa"`.

ВНИМАНИЕ

Отсутствие блока проверки ошибок делает этот код весьма опасным. Он не подходит для практической работы, но вполне пригоден для тестовых целей.

Теперь определим саму нейронную сеть (листинг 7.15).

Листинг 7.15. iris_test.py (продолжение)

```
iris_network: Network = Network([4, 6, 3], 0.3)
```

Аргумент `layer_structure` определяет сеть с тремя слоями (одним входным, одним скрытым и одним выходным) с параметрами `[4, 6, 3]`: входной слой имеет четыре нейрона, скрытый — шесть, а выходной — три. Четыре нейрона входного слоя соответствуют четырем параметрам, используемым для классификации каждого образца. Три нейрона выходного слоя соответствуют непосредственно трем разным видам, к которым мы стремимся отнести каждый входной элемент. Шесть нейронов скрытого слоя являются скорее результатом проб и ошибок, чем некоей формулой. То же самое относится и к `learning_rate`. Эти два значения (количество нейронов в скрытом слое и скорость обучения) могут быть получены экспериментально, если точность сети окажется ниже оптимальной (листинг 7.16).

Листинг 7.16. iris_test.py (продолжение)

```
def iris_interpret_output(output: List[float]) -> str:
    if max(output) == output[0]:
        return "Iris-setosa"
    elif max(output) == output[1]:
        return "Iris-versicolor"
    else:
        return "Iris-virginica"
```

`iris_interpret_output()` — служебная функция, которая передается методу сети `validate()` для определения правильных классификаций.

Наконец сеть готова к обучению (листинг 7.17).

Листинг 7.17. iris_test.py (продолжение)

```
# обучение для первых 140 ирисов из набора данных, и так 50 раз
iris_trainers: List[List[float]] = iris_parameters[0:140]
iris_trainers_corrects: List[List[float]] = iris_classifications[0:140]
for _ in range(50):
    iris_network.train(iris_trainers, iris_trainers_corrects)
```

Обучаем сеть на первых 140 ирисах из набора данных, который состоит из 150 ирисов. Напомню, что строки, прочитанные из CSV-файла, были перетасованы. Это гарантирует, что при каждом запуске программы сеть обучается на разных подмножествах набора данных. Обратите внимание на то, что мы обучаем сеть 50 раз на 140 ирисах. Изменение этого значения сильно влияет на количество времени, которое потребуется на обучение нейронной сети. Как правило, чем дольше обучение, тем точнее будет работать нейронная сеть. Финальным тестом станет проверка правильности классификации последних десяти ирисов из набора данных (листинг 7.18).

Листинг 7.18. iris_test.py (продолжение)

```
# тест на последних десяти ирисах из набора данных
iris_testers: List[List[float]] = iris_parameters[140:150]
iris_testers_corrects: List[str] = iris_species[140:150]
iris_results = iris_network.validate(iris_testers, iris_testers_corrects,
  iris_interpret_output)
print(f"{iris_results[0]} correct of {iris_results[1]} = {iris_results[2] *
  100}%")
```

Вся работа сводится к этому последнему вопросу: сколько из десяти случайно выбранных из набора данных ирисов будут правильно классифицированы нейронной сетью? Поскольку начальные веса каждого нейрона определялись случайным образом, разные прогоны могут давать разные результаты. Можете попробовать настроить скорость обучения, количество скрытых нейронов и количество итераций обучения, чтобы сделать сеть более точной.

В итоге вы должны увидеть такой результат:

```
9 correct of 10 = 90.0%
```

7.5.3. Классификация вина

Мы собираемся протестировать нейронную сеть на другом наборе данных, основанном на химическом анализе сортов итальянских вин[1]. Этот набор данных содержит 178 образцов. Механизм работы с ним будет примерно таким же, что и с набором данных об ирисах, но структура CSV-файла немного иная. Вот его фрагмент:

```
1,14.23,1.71,2.43,15.6,127,2.8,3.06,.28,2.29,5.64,1.04,3.92,1065
1,13.2,1.78,2.14,11.2,100,2.65,2.76,.26,1.28,4.38,1.05,3.4,1050
1,13.16,2.36,2.67,18.6,101,2.8,3.24,.3,2.81,5.68,1.03,3.17,1185
1,14.37,1.95,2.5,16.8,113,3.85,3.49,.24,2.18,7.8,.86,3.45,1480
1,13.24,2.59,2.87,21,118,2.8,2.69,.39,1.82,4.32,1.04,2.93,735
```

Первым значением в каждой строке всегда будет целое число от 1 до 3, представляющее один из трех сортов, к которым может принадлежать образец. Но обратите внимание на то, сколько здесь еще параметров для классификации. В наборе данных об ирисах было только четыре параметра. В наборе данных о винах их 13.

Наша модель нейронной сети отлично масштабируется, нужно только увеличить количество входных нейронов. Файл `wine_test.py` аналогичен `iris_test.py`, но в него внесено несколько незначительных изменений, чтобы учесть различия в их структуре (листинг 7.19).

Листинг 7.19. wine_test.py

```
import csv
from typing import List
```

[1] *Lichman M.* UCI Machine Learning Repository. — Irvine, CA: University of California, School of Information and Computer Science, 2013. http://archive.ics.uci.edu/ml.

```
from util import normalize_by_feature_scaling
from network import Network
from random import shuffle

if __name__ == "__main__":
    wine_parameters: List[List[float]] = []
    wine_classifications: List[List[float]] = []
    wine_species: List[int] = []
    with open('wine.csv', mode='r') as wine_file:
        wines: List = list(csv.reader(wine_file,
          quoting=csv.QUOTE_NONNUMERIC))
        shuffle(wines) # получаем наши строки данных в случайном порядке
        for wine in wines:
            parameters: List[float] = [float(n) for n in wine[1:14]]
            wine_parameters.append(parameters)
            species: int = int(wine[0])
            if species == 1:
                wine_classifications.append([1.0, 0.0, 0.0])
            elif species == 2:
                wine_classifications.append([0.0, 1.0, 0.0])
            else:
                wine_classifications.append([0.0, 0.0, 1.0])
            wine_species.append(species)
    normalize_by_feature_scaling(wine_parameters)
```

Как уже упоминалось, конфигурация слоя в сети для классификации сортов вин требует 13 входных нейронов — по одному для каждого параметра. Кроме того, нужны три выходных нейрона (существует три сорта вина, точно так же, как раньше было три вида ирисов). Интересно, что сеть хорошо работает, когда число нейронов в скрытом слое меньше, чем во входном (листинг 7.20). Одним из возможных интуитивных объяснений этого является то, что некоторые входные параметры на самом деле не помогают выполнить классификацию и их лучше опустить во время обработки. На самом деле меньшее количество нейронов в скрытом слое работает не совсем так, но это интересное предположение.

Листинг 7.20. wine_test.py (продолжение)

```
wine_network: Network = Network([13, 7, 3], 0.9)
```

Подчеркну еще раз: иногда интересно поэкспериментировать с другим количеством скрытых нейронов или с другой скоростью обучения (листинг 7.21).

Листинг 7.21. wine_test.py (продолжение)

```
def wine_interpret_output(output: List[float]) -> int:
    if max(output) == output[0]:
        return 1
    elif max(output) == output[1]:
        return 2
    else:
        return 3
```

Функция `wine_interpret_output()` аналогична `iris_interpret_output()`. Поскольку у нас нет названий сортов вин, просто присваиваем сортам целочисленные номера и используем их в исходном наборе данных.

Листинг 7.22. wine_test.py (продолжение)

```
# обучение на первых 150 образцах вина, повторяется десять раз
wine_trainers: List[List[float]] = wine_parameters[0:150]
wine_trainers_corrects: List[List[float]] = wine_classifications[0:150]
for _ in range(10):
    wine_network.train(wine_trainers, wine_trainers_corrects)
```

Будем обучать сеть на первых 150 образцах из набора данных, оставляя последние 28 для проверки, и повторим обучение десять раз для каждой выборки, что значительно меньше, чем 50 для набора данных об ирисах (листинг 7.22). По какой-то причине (возможно, из-за специфических свойств этого набора данных или настроек параметров, таких как скорость обучения и количество скрытых нейронов) этот набор данных требует более короткого обучения для достижения значительной точности, чем набор данных об ирисах (листинг 7.23).

Листинг 7.23. wine_test.py (продолжение)

```
# тестирование на последних 28 образцах вина в наборе данных
wine_testers: List[List[float]] = wine_parameters[150:178]
wine_testers_corrects: List[int] = wine_species[150:178]
wine_results = wine_network.validate(wine_testers, wine_testers_corrects,
  wine_interpret_output)
print(f"{wine_results[0]} correct of {wine_results[1]} = {wine_results[2] *
  100}%")
```

Если повезет, то ваша нейронная сеть сможет довольно точно классифицировать 28 образцов:

```
27 correct of 28 = 96.42857142857143%
```

7.6. Повышение скорости работы нейронной сети

Нейронные сети требуют множества операций с векторами и матрицами. По сути, это означает, что нужно взять список чисел и выполнить операцию для всех из них одновременно. Поскольку машинное обучение продолжает проникать в наше общество, библиотеки для оптимизированной, эффективной векторной и матричной математики приобретают все большее значение. Во многих библиотеках используются преимущества графических процессоров, поскольку они оптимизированы для этой роли. (На векторах и матрицах базируется компьютерная графика.) Более старая спецификация библиотеки, о которой вы, возможно, слышали, называется BLAS (Basic Linear Algebra Subprograms — подпрограммы базовой линейной алге-

бры). Реализация BLAS лежит в основе популярной числовой библиотеки Python NumPy.

Помимо графических процессоров, можно использовать расширения центрального процессора, позволяющие ускорить обработку векторов и матриц. В состав NumPy входят функции, которые задействуют *SIMD*-инструкции (single instruction, multiple data — одна инструкция, несколько данных). SIMD-инструкции — это специальные инструкции для микропроцессора, которые позволяют обрабатывать несколько фрагментов данных одновременно. Их иногда называют также *векторными инструкциями*.

Разные микропроцессоры содержат разные SIMD-инструкции. Например, SIMD-расширение для G4 (процессор архитектуры PowerPC, который встречается в ранних версиях Mac) называлось AltiVec. Микропроцессоры ARM-архитектуры, а также те, что устанавливаются в iPhone, имеют расширение NEON. Современные микропроцессоры Intel включают в себя SIMD-расширения, известные как MMX, SSE, SSE2 и SSE3. К счастью, вам не нужно знать их все. Библиотека, подобная NumPy, автоматически выберет правильные инструкции для эффективного вычисления на той архитектуре, на которой работает программа.

Поэтому неудивительно, что реальные библиотеки нейронных сетей, в отличие от игрушечной библиотеки, созданной в этой главе, используют в качестве базовой структуры данных массивы NumPy вместо списков стандартных библиотек Python. Но этим они не ограничиваются. Популярные библиотеки нейронных сетей Python, такие как TensorFlow и PyTorch, не только задействуют SIMD-инструкции, но и широко применяют вычисления на GPU. Поскольку графические процессоры специально оптимизированы для быстрых векторных вычислений, это на порядок ускоряет работу нейронных сетей по сравнению с работой только на центральном процессоре.

Подчеркну еще раз: вы вряд ли станете строить нейронную сеть для практической работы, используя только стандартную библиотеку Python, как мы делали в этой главе. Вместо этого следует задействовать хорошо оптимизированную библиотеку с поддержкой SIMD и GPU, такую как TensorFlow. Единственными исключениями будет библиотека нейронных сетей, предназначенная для обучения, или библиотека, которая должна работать на встроенном устройстве, не поддерживающем SIMD-инструкции и не имеющем GPU.

7.7. Проблемы и расширения нейронных сетей

Благодаря достижениям в области глубокого обучения нейронные сети сейчас в моде, но у них есть существенные недостатки. Самая большая проблема заключается в том, что решение задачи в нейронной сети — это что-то вроде черного ящика. Даже когда нейронные сети работают хорошо, они не позволяют пользователю детально разобраться в том, как именно они решают проблему. Например, классификатор набора данных ирисов, над которым мы работали в этой главе,

не особенно ясно показывает, насколько каждый из четырех входных параметров влияет на выходные данные. Что важнее для классификации образца, длина или ширина чашелистика?

Возможно, тщательный анализ окончательных весов в обученной сети может дать некоторое представление об этом, но он нетривиален и не дает такого понимания, которое дает, скажем, линейная регрессия с точки зрения значения каждой переменной в моделируемой функции. Другими словами, нейронная сеть может решить задачу, но не объясняет, как именно это делает.

Другая проблема нейронных сетей заключается в том, что для обеспечения точности им часто нужны очень большие наборы данных. Представьте себе классификатор изображений для пейзажей. Возможно, потребуется классифицировать тысячи различных типов изображений (леса, долины, горы, ручьи, степи и т. п.). Потенциально для этого потребуются миллионы обучающих образов. Такие большие наборы данных не только трудно найти — для некоторых приложений их может вообще не существовать. Как правило, хранилищами данных и техническими средствами для сбора и хранения таких объемных наборов данных располагают крупные корпорации и правительства.

Наконец, нейронные сети дóроги с вычислительной точки зрения. Как вы, наверное, заметили, даже простое обучение на наборе данных об ирисах может вызвать перегрузку интерпретатора Python. Чистый Python (без библиотек с поддержкой C, хотя бы NumPy) — не высокопроизводительная вычислительная среда, но на любой вычислительной платформе, где используются нейронные сети, выполняется огромное количество вычислений для обучения сети — ничто другое не занимает так много времени. Существует множество приемов, позволяющих повысить производительность нейронной сети, например, применение SIMD-инструкций или графических процессоров, но в любом случае обучение нейронной сети требует большого количества операций с плавающей точкой.

Одним из приятных нюансов является то, что в вычислительном отношении обучение обходится намного дороже, чем использование сети. Некоторые приложения не требуют постоянного обучения. В этих случаях обученную сеть можно просто вставить в приложение для решения задачи. Например, первая версия платформы Apple Core ML даже не поддерживает обучение. Она только помогает разработчикам приложений запускать предварительно обученные модели нейронных сетей в своих приложениях. Разработчик, создающий приложение для обработки фотографий, может загрузить модель классификации изображений с открытой лицензией, вставить ее в Core ML и сразу же начать задействовать эффективное машинное обучение в своем приложении.

В этой главе мы работали с нейронной сетью только одного типа — с прямой связью и обратным распространением. Как уже упоминалось, существует много других видов нейронных сетей. Сверточные нейронные сети также имеют прямую связь, но у них есть несколько типов скрытых слоев, различные механизмы распределения весов и другие интересные свойства, из-за чего они особенно хорошо подходят для классификации изображений. В рекуррентных нейронных сетях

сигналы не просто движутся в одном направлении — такие сети допускают петли обратной связи. Эти сети оказались полезными для приложений непрерывного ввода, распознающих написанное от руки и прочитанное вслух.

Простым расширением нейронной сети, которое могло бы сделать ее более производительной, было бы добавление нейронов смещения. Нейроны смещения подобны фиктивным нейронам в слое, который позволяет представлять выходным данным следующего слоя больше функций, обеспечивая подачу на него постоянного входного сигнала (все еще измененного посредством весов). Даже простые нейронные сети, используемые для реальных задач, обычно содержат нейроны смещения. Если вы добавите такие нейроны в созданную здесь сеть, то, вероятно, обнаружите, что для достижения аналогичного уровня точности потребуется меньше времени на обучение.

7.8. Реальные приложения

Несмотря на то что искусственные нейронные сети появились еще в середине XX века, до последнего десятилетия они не были широко распространены. Массовое применение нейронных сетей сдерживалось отсутствием достаточно производительного оборудования. Сегодня в машинном обучении искусственные нейронные сети стали областью с поистине взрывным ростом, потому что они работают!

За последние десятилетия искусственные нейронные сети позволили создать очень интересные компьютерные приложения, ориентированные на пользователя. К ним относятся практичное (с точки зрения достаточной точности) распознавание голоса, распознавание изображений и рукописных текстов. Распознавание голоса имеется в таких средствах голосового ввода, как Dragon Naturally Speaking, и цифровых помощниках, таких как Siri, Alexa и Cortana. Конкретный пример распознавания изображений — автоматическая отметка людей на фотографиях в Facebook посредством распознавания лиц. В последних версиях iOS можно выполнять поиск в заметках, даже если они написаны от руки, с помощью распознавания рукописного ввода.

Более старая технология распознавания, эффективность которой можно повысить за счет использования нейронных сетей, — это оптическое распознавание символов (optical character recognition, OCR). Технология OCR применяется при сканировании документов и возвращает редактируемый текст вместо изображения. OCR позволяет почтовой службе считывать индексы на конвертах для быстрой сортировки корреспонденции.

В этой главе было показано, что нейронные сети успешно действуют в задачах классификации. Подобные приложения, в которых хорошо работают нейронные сети, — это системы выдачи рекомендаций. Именно так Netflix предлагает фильм, который вас может заинтересовать, а Amazon — книгу, которую вы, вероятно, захотите прочитать. Существуют и другие методы машинного обучения, хорошо подходящие для систем рекомендаций (Amazon и Netflix не обязательно используют

нейронные сети для этих целей, детали реализации таких систем обычно не разглашаются), поэтому нейронные сети следует выбирать только после того, как все параметры изучены.

Нейронные сети могут применяться в любой ситуации, когда необходима аппроксимация неизвестной функции. Это делает их полезными для прогнозирования. Нейронные сети могут использоваться и используются для прогнозирования результатов спортивных событий, выборов или торгов на фондовом рынке. Конечно, их точность зависит от того, насколько хорошо они обучены, то есть от того, насколько велик был набор данных, относящихся к событию с неизвестным результатом, насколько хорошо настроены параметры нейронной сети и сколько итераций обучения выполнено. Как и в большинстве приложений с применением нейронных сетей, одна из самых сложных частей прогнозирования — выбор структуры сети, которая часто определяется методом проб и ошибок.

7.9. Упражнения

1. Примените инфраструктуру нейронной сети, разработанную в этой главе, для классификации элементов из другого набора данных.
2. Создайте обобщенную функцию `parse_CSV()` с параметрами, достаточно гибкими для того, чтобы она могла заменить оба примера синтаксического анализа данных в формате CSV из этой главы.
3. Попробуйте запустить примеры с другой функцией активации (не забудьте также найти ее производную). Как изменение функции активации влияет на точность сети? Требуется ли при этом большее или меньшее обучение?
4. Проработайте заново решения задач из этой главы, используя популярную инфраструктуру нейронных сетей, такую как TensorFlow или PyTorch.
5. Перепишите классы `Network`, `Layer` и `Neuron` с помощью NumPy, чтобы ускорить работу нейронной сети, разработанной в этой главе.

8 Состязательный поиск

Идеальная информационная игра для двух игроков с нулевой суммой — это игра, в которой оба соперника имеют всю доступную им информацию о состоянии игры и все, что является преимуществом для одного, становится потерей преимущества для другого. К таким играм относятся крестики-нолики, Connect Four, шашки и шахматы. В этой главе вы узнаете, как создать сильного искусственного соперника, способного играть в такие игры. В сущности, обсуждаемые методы в сочетании с современными вычислительными возможностями позволяют создавать искусственных соперников, которые прекрасно играют в простые игры этого класса и способны играть в сложные игры, выходящие за пределы возможностей любого соперника-человека.

8.1. Основные компоненты настольной игры

Как и при рассмотрении большинства более сложных задач в предыдущих главах, постараемся сделать решение как можно более обобщенным. В случае состязательного поиска это означает, что наши алгоритмы поиска не должны зависеть от игры. Начнем с определения нескольких простых базовых классов, которые выявляют состояние, необходимое алгоритмам поиска. Затем создадим подклассы базовых классов для конкретных игр (крестики-нолики и Connect Four) и введем эти

подклассы в алгоритмы поиска, чтобы они могли «играть» в эти игры. Вот базовые классы (листинг 8.1).

Листинг 8.1. board.py

```
from __future__ import annotations
from typing import NewType, List
from abc import ABC, abstractmethod

Move = NewType('Move', int)

class Piece:
    @property
    def opposite(self) -> Piece:
        raise NotImplementedError("Should be implemented by subclasses.")

class Board(ABC):
    @property
    @abstractmethod
    def turn(self) -> Piece:
        ...

    @abstractmethod
    def move(self, location: Move) -> Board:
        ...

    @property
    @abstractmethod
    def legal_moves(self) -> List[Move]:
        ...

    @property
    @abstractmethod
    def is_win(self) -> bool:
        ...

    @property
    def is_draw(self) -> bool:
        return (not self.is_win) and (len(self.legal_moves) == 0)

    @abstractmethod
    def evaluate(self, player: Piece) -> float:
        ...
```

Тип `Move` будет представлять ход в игре. По сути, это просто целое число. В таких играх, как крестики-нолики и Connect Four, целое число может описывать ход, определяя квадрат или столбец, в котором должна быть размещена фигура. `Piece` — это базовый класс для фигуры на доске в игре. Его другая роль — индикатор хода. Именно для этого необходимо свойство `opposite`. Нам нужно знать, чей ход следует за текущим ходом.

СОВЕТ

Поскольку в крестиках-ноликах и игре Connect Four существует только один вид фигур, класс Piece в этой главе может использоваться как индикатор хода. В более сложных играх, таких как шахматы, где есть разные виды фигур, ходы могут обозначаться целым числом или логическим значением. В качестве альтернативы можно применять для обозначения хода атрибут color более сложного типа Piece.

Абстрактный базовый класс `Board` является фактическим хранилищем состояния. Для любой конкретной игры, которую будут выполнять алгоритмы поиска, они должны уметь ответить на следующие вопросы.

- Чей сейчас ход?
- Какие ходы можно сделать из текущей позиции согласно правилам?
- Выиграна ли сейчас игра?
- Сыграна ли игра вничью?

Последний вопрос, касающийся ничьих, на самом деле является комбинацией двух предыдущих вопросов для многих игр. Если игра не выиграна, но возможных ходов нет, то это ничья. Вот почему в абстрактном базовом классе `Game` сразу можно создать конкретную реализацию свойства `is_draw`. Кроме того, есть еще несколько действий, которые необходимо реализовать.

- Сделать ход, чтобы перейти из текущей в новую позицию.
- Оценить позицию, чтобы увидеть, какой игрок имеет преимущество.

Каждый метод и свойство класса `Board` являются реализацией одного из предыдущих вопросов или действий. На языке игры класс `Board` можно было бы назвать `Position`, но мы используем это имя для чего-то более конкретного в каждом из подклассов.

8.2. Крестики-нолики

Крестики-нолики — простая игра, но ее можно взять для иллюстрации того же минимаксного алгоритма, который применяется в сложных стратегических играх, таких как Connect Four, шашки и шахматы. Мы построим искусственный интеллект, который прекрасно играет в крестики-нолики с помощью минимаксного алгоритма.

ПРИМЕЧАНИЕ

В этом разделе предполагается, что вы знакомы с игрой в «Крестики-нолики» и ее стандартными правилами. Если нет, то, чуть-чуть поискав в Интернете, вы быстро найдете их.

8.2.1. Управление состоянием игры в крестики-нолики

Давайте разработаем несколько структур, позволяющих отслеживать состояние игры в крестики-нолики по мере ее развития.

Прежде всего нужен способ представления каждой клетки на игровом поле. Будем использовать перечисление `TTTPiece` — подкласс класса `Piece`. Клетка в игре в крестики-нолики может иметь значение `X`, `0` или быть пустой (в перечислении такие обозначаются `E`) (листинг 8.2).

Листинг 8.2. tictactoe.py

```
from __future__ import annotations
from typing import List
from enum import Enum
from board import Piece, Board, Move

class TTTPiece(Piece, Enum):
    X = "X"
    O = "O"
    E = " " # обозначение пустой клетки

    @property
    def opposite(self) -> TTTPiece:
        if self == TTTPiece.X:
            return TTTPiece.O
        elif self == TTTPiece.O:
            return TTTPiece.X
        else:
            return TTTPiece.E

    def __str__(self) -> str:
        return self.value
```

В классе `TTTPiece` есть свойство `opposite`, которое возвращает другой экземпляр `TTTPiece`. Это будет полезно для передачи хода от одного игрока к другому после того, как очередной ход игры завершен. Для представления ходов используем обычное целое число, соответствующее клетке на поле, где был поставлен крестик или нолик. Как вы помните, `Move` был определен в файле `board.py` именно как целое число.

В игре в крестики-нолики существует девять позиций, образующих три ряда по три столбца. Для простоты эти девять позиций можно представить в виде одномерного списка. То, какие клетки получат какие номера (другими словами, индексы в массиве), не имеет значения, но мы будем придерживаться схемы, показанной на рис. 8.1.

0	1	2
3	4	5
6	7	8

Рис. 8.1. Каждой клетке на игровом поле соответствует индекс одномерного списка

Главным хранилищем состояния игры будет класс `TTTBoard`. Он отслеживает два элемента состояния: позицию, представленную вышеупомянутым одномерным списком, и игрока, который сейчас делает ход (листинг 8.3).

Листинг 8.3. tictactoe.py (продолжение)

```
class TTTBoard(Board):
    def __init__(self, position: List[TTTPiece] = [TTTPiece.E] * 9, turn:
       TTTPiece = TTTPiece.X) -> None:
        self.position: List[TTTPiece] = position
        self._turn: TTTPiece = turn

    @property
    def turn(self) -> Piece:
        return self._turn
```

Исходное состояние поля — такое, при котором еще не сделано ни одного хода (пустое поле). Конструктор `Board` имеет параметры по умолчанию, такую позицию, при которой первый игрок готовится поставить `X` (обычный первый ход в игре). Вы можете спросить: зачем нужны переменная экземпляра `_turn` и свойство `turn`? Такой прием позволяет гарантировать, что все подклассы класса `Board` будут отслеживать, чья очередь ходить в данный момент. В Python нет четкого и очевидного способа указать в абстрактном базовом классе, что подклассы должны включать в себя конкретную переменную экземпляра, но для их свойств такой механизм существует.

`TTTBoard` — это по соглашению неизменяемая структура данных: структуры `TTTBoard` не должны изменяться. Вместо этого каждый раз, когда необходимо сделать ход, будет генерироваться новая структура `TTTBoard`, позиция которой изменена с учетом хода. Впоследствии это пригодится в алгоритме поиска. При ветвлении поиска мы не сможем случайно изменить положение на поле, начиная с которого все еще анализируются потенциально возможные ходы (листинг 8.4).

Листинг 8.4. tictactoe.py (продолжение)

```
def move(self, location: Move) -> Board:
    temp_position: List[TTTPiece] = self.position.copy()
    temp_position[location] = self._turn
    return TTTBoard(temp_position, self._turn.opposite)
```

Допустимым ходом в игре является любая пустая клетка. В показанном далее свойстве `legal_moves` используется генератор списков для генерации возможных ходов из данной позиции (листинг 8.5).

Листинг 8.5. tictactoe.py (продолжение)

```
@property
def legal_moves(self) -> List[Move]:
    return [Move(l) for l in range(len(self.position)) if self.position[l] ==
      TTTPiece.E]
```

Индексы, на которые воздействует генератор списков, являются индексами типа `int` в списке позиций. То, что `Move` также определяется как тип `int`, сделано намеренно (и это удобно), так как обеспечивает краткость определения `legal_moves`.

Существует множество способов просмотреть строки, столбцы и диагонали на поле игры, чтобы проверить, завершилась ли она победой. В показанной далее реализации свойства `is_win` это сделано посредством жестко закодированной конструкции из `and`, `or` и `==`, которая кажется бесконечной (листинг 8.6). Это не самый красивый код, но он простой и выполняет свою работу.

Листинг 8.6. tictactoe.py (продолжение)

```
@property
def is_win(self) -> bool:
    # проверяем три строки, три столбца и две диагонали
    return self.position[0] == self.position[1] and self.position[0] ==
      self.position[2] and self.position[0] != TTTPiece.E or \
    self.position[3] == self.position[4] and self.position[3] ==
      self.position[5] and self.position[3] != TTTPiece.E or \
    self.position[6] == self.position[7] and self.position[6] ==
      self.position[8] and self.position[6] != TTTPiece.E or \
    self.position[0] == self.position[3] and self.position[0] ==
      self.position[6] and self.position[0] != TTTPiece.E or \
    self.position[1] == self.position[4] and self.position[1] ==
      self.position[7] and self.position[1] != TTTPiece.E or \
    self.position[2] == self.position[5] and self.position[2] ==
      self.position[8] and self.position[2] != TTTPiece.E or \
    self.position[0] == self.position[4] and self.position[0] ==
      self.position[8] and self.position[0] != TTTPiece.E or \
    self.position[2] == self.position[4] and self.position[2] ==
      self.position[6] and self.position[2] != TTTPiece.E
```

Если все клетки строки, столбца или диагонали не пустые и содержат одинаковые элементы, то игра выиграна.

Игра закончена вничью, если она не выиграна и не осталось допустимых ходов (это свойство уже описано в абстрактном базовом классе `Board`). Наконец, нам нужен способ оценки конкретной позиции и структурного вывода состояния поля (листинг 8.7).

Листинг 8.7. tictactoe.py (продолжение)

```
def evaluate(self, player: Piece) -> float:
    if self.is_win and self.turn == player:
        return -1
    elif self.is_win and self.turn != player:
        return 1
    else:
        return 0

def __repr__(self) -> str:
    return f"""{self.position[0]}|{self.position[1]}|{self.position[2]}

-----
{self.position[3]}|{self.position[4]}|{self.position[5]}
-----
{self.position[6]}|{self.position[7]}|{self.position[8]}"""
```

В большинстве игр приходится вычислять позицию приблизительно, поскольку мы не можем пройти игру до самого конца, чтобы точно определить, кто выиграет или проиграет в зависимости от того, какие ходы будут сделаны. Но в крестиках-ноликах довольно малое пространство поиска, так что мы можем пройти его от любой позиции до конца игры. Следовательно, метод `evaluate()` может просто возвращать некое число, если игрок выиграл, меньшее число — в случае ничьей и совсем малое — в случае проигрыша.

8.2.2. Минимакс

Минимакс — это классический алгоритм для поиска наилучшего хода в игре с двумя игроками, нулевой суммой и отличной информацией, такой как крестики-нолики, шашки или шахматы. Этот алгоритм был расширен и модифицирован для других типов игр. Минимакс обычно реализуется с использованием рекурсивной функции, в которой каждый игрок обозначается как максимизирующий или минимизирующий.

Максимизирующий игрок стремится найти ход, который приведет к максимальному выигрышу. Однако максимизирующий игрок должен учитывать ходы минимизирующего игрока. После каждой попытки максимизирующего игрока максимизировать выигрыш рекурсивно вызывается минимакс, чтобы найти ответ противника, который минимизирует максимизирующий выигрыш игрока. Это продолжается в обоих направлениях (максимизация, минимизация, максимизация и т. д.), пока не будет достигнут базовый случай рекурсивной функции. Базовый случай — это конечная позиция (выигрыш или ничья) либо достижение максимальной глубины поиска.

Минимакс вычисляет стартовую позицию для максимизирующего игрока. Для метода `evaluate()` класса `TTTBoard`, если наилучшая возможная игра обеих сторон приведет к выигрышу максимизирующего игрока, возвращается 1 балл.

Если наилучшая возможная игра приведет к проигрышу, то возвращается –1. Наконец, если наилучшая игра — это ничья, то возвращается 0.

Эти числа возвращаются при достижении базового случая. Затем они передаются через все рекурсивные вызовы, которые привели к базовому случаю. Чтобы максимизировать каждый рекурсивный вызов, вверх по рекурсии передаются наилучшие вычисленные ходы. Чтобы минимизировать каждый рекурсивный вызов, передаются худшие вычисленные ходы. Таким образом строится дерево решений. На рис. 8.2 показано такое дерево, которое упрощает передачу данных вверх по рекурсивным вызовам для игры, в которой осталось сделать два хода.

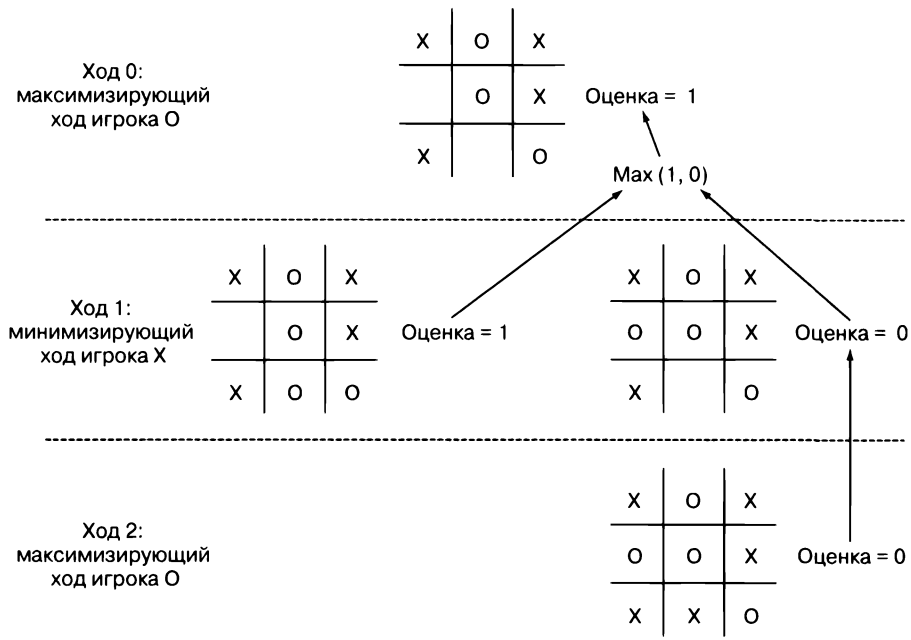


Рис. 8.2. Минимаксное дерево решений для игры в крестики-нолики, в которой осталось сделать два хода. Чтобы максимизировать вероятность выигрыша, первоначальный игрок, 0, должен поставить 0 по центру в нижнем ряду. Стрелки указывают позиции, с которых принимается решение

Для игр, имеющих слишком глубокое пространство поиска, таких как шашки и шахматы, чтобы достичь конечной позиции, минимакс останавливается после достижения определенной глубины (выполнения заданного количества ходов поиска, иногда называемого *ply*). Затем включается функция оценки, использующая эвристику для оценки состояния игры. Чем лучше выглядит игра для первого игрока, тем выше присуждаемая ей оценка. Мы вернемся к этой концепции, когда будем обсуждать игру Connect Four, которая имеет гораздо большее пространство поиска, чем крестики-нолики.

В листинге 8.8 представлена полная реализация функции `minimax()`.

Листинг 8.8. minimax.py

```
from __future__ import annotations
from board import Piece, Board, Move

# Находим наилучший из возможных результатов для первого игрока
def minimax(board: Board, maximizing: bool, original_player: Piece, max_
      depth: int = 8) -> float:
    # Базовый случай – достигнута финальная позиция
    # или максимальная глубина поиска
    if board.is_win or board.is_draw or max_depth == 0:
        return board.evaluate(original_player)

    # Рекурсивный случай – максимизируйте свою выгоду
    # или минимизируйте выгоду противника
    if maximizing:
        best_eval: float = float("-inf")
        # произвольно низкая начальная точка
        for move in board.legal_moves:
            result: float = minimax(board.move(move), False,
              original_player, max_depth - 1)
            best_eval = max(result, best_eval)
        return best_eval
    else: # минимизация
        worst_eval: float = float("inf")
        for move in board.legal_moves:
            result = minimax(board.move(move), True, original_player,
              max_depth - 1)
            worst_eval = min(result, worst_eval)
        return worst_eval
```

При каждом рекурсивном вызове нужно отслеживать позицию на поле независимо от того, максимизируем мы ее или минимизируем и для кого пытаемся оценить позицию (`original_player`). Первые несколько строк функции `minimax()` касаются базового случая — заключительной позиции (выигрыш, проигрыш или ничья) или максимально достижимой глубины. Остальная часть функции обрабатывает рекурсивные случаи.

Один из рекурсивных случаев — это максимизация. В этой ситуации мы ищем ход, который даст максимально возможную оценку. Второй рекурсивный случай — минимизация: ищем ход, который приведет к наименьшей возможной оценке. Как бы то ни было, эти два случая чередуются, пока не будет достигнута финальная позиция или максимальная глубина поиска (базовый случай).

К сожалению, мы не можем использовать нашу реализацию `minimax()`, чтобы найти наилучший ход для данной позиции, так как функция возвращает оценку — значение с плавающей запятой. Оно не сообщает, какой лучший первый ход привел к данной оценке.

Вместо этого создадим вспомогательную функцию `find_best_move()`, которая перебирает вызовы `minimax()` для каждого допустимого хода из данной позиции и позволяет найти ход, который имел бы максимальную оценку. Функцию `find_best_move()` можно представить как первый максимизирующий вызов `minimax()`, но с отслеживанием начальных ходов (листинг 8.9).

Листинг 8.9. minimax.py (продолжение)

```
# Найти наилучший возможный ход из текущей позиции,
# просматривая max_depth ходов вперед
def find_best_move(board: Board, max_depth: int = 8) -> Move:
    best_eval: float = float("-inf")
    best_move: Move = Move(-1)
    for move in board.legal_moves:
        result: float = minimax(board.move(move), False, board.turn,
          max_depth)
        if result > best_eval:
            best_eval = result
            best_move = move
    return best_move
```

Теперь у нас есть все необходимое, чтобы найти наилучший возможный ход для любой позиции при игре в крестики-нолики.

8.2.3. Тестирование минимакса для игры в крестики-нолики

Крестики-нолики — такая простая игра, что даже нам, людям, легко найти в ней правильный ход для текущей позиции. Это позволяет легко разрабатывать модульные тесты.

В следующем фрагменте кода мы испытаем минимаксный алгоритм и попытаемся найти правильный ход из трех различных позиций. Первый тест простой: требуется найти лишь один ход, чтобы одержать победу. Второй тест требует блокировать соперника — искусственный интеллект должен помешать противнику одержать победу. Последний тест немного сложнее и требует от искусственного интеллекта продумать игру на два шага вперед (листинг 8.10).

Листинг 8.10. tictactoe_tests.py

```
import unittest
from typing import List
from minimax import find_best_move
from tictactoe import TTTPiece, TTTBoard
from board import Move

class TTTMinimaxTestCase(unittest.TestCase):
    def test_easy_position(self):
```

```
        # выигрыш за один ход
        to_win_easy_position: List[TTTPiece] = [TTTPiece.X, TTTPiece.O,
          TTTPiece.X, TTTPiece.X, TTTPiece.E,
          TTTPiece.O, TTTPiece.E, TTTPiece.E,
          TTTPiece.O]
        test_board1: TTTBoard = TTTBoard(to_win_easy_position,
          TTTPiece.X)
        answer1: Move = find_best_move(test_board1)
        self.assertEqual(answer1, 6)

    def test_block_position(self):
        # нужно помешать противнику выиграть
        to_block_position: List[TTTPiece] = [TTTPiece.X, TTTPiece.E,
          TTTPiece.E, TTTPiece.E, TTTPiece.E,
          TTTPiece.O, TTTPiece.E, TTTPiece.X,
          TTTPiece.O]
        test_board2: TTTBoard = TTTBoard(to_block_position,
          TTTPiece.X)
        answer2: Move = find_best_move(test_board2)
        self.assertEqual(answer2, 2)

    def test_hard_position(self):
        # найти лучший ход, чтобы выиграть в два хода
        to_win_hard_position: List[TTTPiece] = [TTTPiece.X, TTTPiece.E,
          TTTPiece.E, TTTPiece.E, TTTPiece.E,
          TTTPiece.O, TTTPiece.O, TTTPiece.X,
          TTTPiece.E]
        test_board3: TTTBoard = TTTBoard(to_win_hard_position,
          TTTPiece.X)
        answer3: Move = find_best_move(test_board3)
        self.assertEqual(answer3, 1)

if __name__ == '__main__':
    unittest.main()
```

Для выполнения всех трех тестов запустите файл `tictactoe_tests.py`.

СОВЕТ

Для реализации минимакса не нужно писать много кода: этот алгоритм пригоден для гораздо большего количества игр, чем просто крестики-нолики. Если вы планируете внедрить минимакс для другой игры, важно настроиться на успех, создавая структуры данных, которые хорошо работают в сочетании с минимаксом, такие как класс Board. Распространенной ошибкой для тех, кто изучает минимакс, является использование изменяемой структуры данных, которая изменяется при рекурсивном вызове минимакса и затем не может быть возвращена в исходное состояние для последующих вызовов алгоритма.

8.2.4. Разработка ИИ для игры в крестики-нолики

Теперь, когда у нас есть все необходимые ингредиенты, можно легко сделать следующий шаг — разработать полностью искусственного противника, способного пройти всю игру в крестики-нолики. Вместо того чтобы оценивать тестовую позицию, ИИ будет оценивать только позицию, генерируемую на каждом ходе противника. В следующем фрагменте короткого кода ИИ играет в крестики-нолики против оппонента-человека, который делает первый ход (листинг 8.11).

Листинг 8.11. tictactoe_ai.py

```
from minimax import find_best_move
from tictactoe import TTTBoard
from board import Move, Board

board: Board = TTTBoard()

def get_player_move() -> Move:
    player_move: Move = Move(-1)
    while player_move not in board.legal_moves:
        play: int = int(input("Enter a legal square (0-8):"))
        player_move = Move(play)
    return player_move

if __name__ == "__main__":
    # главный цикл игры
    while True:
        human_move: Move = get_player_move()
        board = board.move(human_move)
        if board.is_win:
            print("Human wins!")
            break
        elif board.is_draw:
            print("Draw!")
            break
        computer_move: Move = find_best_move(board)
        print(f"Computer move is {computer_move}")
        board = board.move(computer_move)
        print(board)
        if board.is_win:
            print("Computer wins!")
            break
        elif board.is_draw:
            print("Draw!")
            break
```

Поскольку по умолчанию значение `max_depth` для `find_best_move()` равно 8, этот ИИ для игры в крестики-нолики всегда будет просматривать ходы до кон-

ца игры. (Максимальное количество ходов в крестиках-ноликах равно девяти, а ИИ ходит вторым.) Поэтому ИИ всегда должен играть идеально. Идеальная игра — это игра, в которой оба противника каждый раз выбирают наилучший ход. Результатом идеальной игры в крестики-нолики является ничья. Учитывая это, вы никогда не сможете выиграть у искусственного интеллекта. Если вы будете играть идеально, то сыграете вничью, если сделаете ошибку, то ИИ победит. Попробуйте сами. Вы не должны победить.

8.3. Connect Four

В игре Connect Four[1] два игрока поочередно бросают разноцветные фишки в вертикальную сетку, состоящую из семи столбцов и шести рядов. Фишки падают вдоль сетки сверху вниз, пока не достигнут дна или не лягут на другую фишку. По сути, единственным выбором игрока на каждом ходе является то, в какой из семи столбцов бросить фишку. Игрок не может поместить фишку в заполненный столбец. Первый игрок, которому удастся выставить четыре фишки своего цвета в строку, столбец или по диагонали без разрывов, выигрывает. Если ни одному из игроков не удалось этого достичь, а сетка заполнена, считается, что игра закончилась вничью.

8.3.1. Подключите четыре игровых автомата

Игра Connect Four во многом похожа на крестики-нолики. Обе они ведутся на поле в виде сетки, и для выигрыша требуется, чтобы игрок расположил фишки в ряд. Но в Connect Four сетка больше, что предполагает гораздо больше способов выиграть, поэтому оценить каждую позицию значительно сложнее.

Представленный далее код (листинг 8.12) будет выглядеть в чем-то очень знакомым, но структуры данных и метод оценки сильно отличаются от использованных для игры в крестики-нолики. Обе игры реализованы как подклассы тех же базовых классов `Piece` и `Board`, с которыми мы познакомились в начале главы, что делает функцию `minimax()` пригодной для обеих игр.

Листинг 8.12. connectfour.py

```
from __future__ import annotations
from typing import List, Optional, Tuple
from enum import Enum
from board import Piece, Board, Move

class C4Piece(Piece, Enum):
    B = "B"
```

[1] Игра Connect Four («Четыре в ряд») является товарным знаком компании Hasbro, Inc. Здесь она используется только в описательной форме.

```
    R = "R"
    E = " " # пустое поле

    @property
    def opposite(self) -> C4Piece:
        if self == C4Piece.B:
            return C4Piece.R
        elif self == C4Piece.R:
            return C4Piece.B
        else:
            return C4Piece.E

    def __str__(self) -> str:
        return self.value
```

Класс `C4Piece` практически идентичен классу `TTTPiece`.

Теперь определим функцию для генерации всех потенциальных выигрышных сегментов в сетке Connect Four определенного размера (листинг 8.13).

Листинг 8.13. connectfour.py (продолжение)

```
def generate_segments(num_columns: int, num_rows: int, segment_length: int)
  -> List[List[Tuple[int, int]]]:
    segments: List[List[Tuple[int, int]]] = []
    # генерируем вертикальные сегменты
    for c in range(num_columns):
        for r in range(num_rows - segment_length + 1):
            segment: List[Tuple[int, int]] = []
            for t in range(segment_length):
                segment.append((c, r + t))
            segments.append(segment)

    # генерируем горизонтальные сегменты
    for c in range(num_columns - segment_length + 1):
        for r in range(num_rows):
            segment = []
            for t in range(segment_length):
                segment.append((c + t, r))
            segments.append(segment)

    # генерируем сегменты диагонали из нижнего левого в верхний правый угол
    for c in range(num_columns - segment_length + 1):
        for r in range(num_rows - segment_length + 1):
            segment = []
            for t in range(segment_length):
                segment.append((c + t, r + t))
            segments.append(segment)

    # генерируем сегменты диагонали из верхнего левого в нижний правый угол
    for c in range(num_columns - segment_length + 1):
        for r in range(segment_length - 1, num_rows):
            segment = []
```

```
            for t in range(segment_length):
                segment.append((c + t, r - t))
            segments.append(segment)
    return segments
```

Эта функция возвращает список списков ячеек сетки (кортежей, состоящих из комбинаций «столбец/строка»). Каждый список в списке содержит четыре ячейки сетки. Мы будем называть списки, состоящие из четырех ячеек сетки, *сегментами*. Если какой-либо сегмент на доске окажется окрашен в один цвет, это будет означать, что данный цвет выиграл.

Возможность быстрого поиска всех сегментов на доске позволяет не только проверить, закончилась ли игра (кто-то выиграл), но и оценить позицию. Поэтому, как вы увидите в следующем фрагменте кода (листинг 8.14), мы кэшируем сегменты для доски заданного размера в виде переменной `SEGMENTS` в классе `C4Board`.

Листинг 8.14. connectfour.py (продолжение)

```
class C4Board(Board):
    NUM_ROWS: int = 6
    NUM_COLUMNS: int = 7
    SEGMENT_LENGTH: int = 4
    SEGMENTS: List[List[Tuple[int, int]]] = generate_segments(NUM_COLUMNS,
      NUM_ROWS, SEGMENT_LENGTH)
```

У класса `C4Board` есть внутренний класс `Column`. Он не абсолютно необходим — мы могли бы использовать для представления сетки одномерный список, как сделали для игры в крестики-нолики, или же двумерный список. Кроме того, в отличие от любого из этих решений, применение класса `Column`, вероятно, немного снизит производительность. Но зато мы получим концептуально мощное представление об игровом поле Connect Four как о группе, состоящей из семи столбцов, что также немного облегчает написание остальной части класса `C4Board` (листинг 8.15).

Листинг 8.15. connectfour.py (продолжение)

```
class Column:
    def __init__(self) -> None:
        self._container: List[C4Piece] = []

    @property
    def full(self) -> bool:
        return len(self._container) == C4Board.NUM_ROWS

    def push(self, item: C4Piece) -> None:
        if self.full:
            raise OverflowError("Trying to push piece to full column")
        self._container.append(item)

    def __getitem__(self, index: int) -> C4Piece:
        if index > len(self._container) - 1:
```

```
            return C4Piece.E
        return self._container[index]

    def __repr__(self) -> str:
        return repr(self._container)

    def copy(self) -> C4Board.Column:
        temp: C4Board.Column = C4Board.Column()
        temp._container = self._container.copy()
        return temp
```

Класс `Column` очень похож на класс `Stack`, который мы использовали в предыдущих главах. Это имеет смысл, поскольку концептуально во время игры столбец Connect Four представляет собой стек, в который можно помещать данные, но никогда не выталкивать их оттуда. Однако, в отличие от созданных ранее стеков, столбец в Connect Four имеет абсолютный предел в шесть элементов. Также интересен специальный метод `__getitem__()`, который позволяет получить экземпляр `Column` по индексу. Это позволяет обрабатывать список столбцов как двумерный список. Обратите внимание: даже если у дублирующего `_container` нет элемента в какой-то конкретной строке, `__getitem__()` все равно вернет пустой элемент.

Следующие четыре метода чем-то похожи на их аналоги для игры в крестики-нолики (листинг 8.16).

Листинг 8.16. connectfour.py (продолжение)

```
def __init__(self, position: Optional[List[C4Board.Column]] = None, turn:
      C4Piece = C4Piece.B) -> None:
    if position is None:
        self.position: List[C4Board.Column] = [C4Board.Column() for _ in
      range(C4Board.NUM_COLUMNS)]
    else:
        self.position = position
    self._turn: C4Piece = turn

@property
def turn(self) -> Piece:
    return self._turn

def move(self, location: Move) -> Board:
    temp_position: List[C4Board.Column] = self.position.copy()
    for c in range(C4Board.NUM_COLUMNS):
        temp_position[c] = self.position[c].copy()
    temp_position[location].push(self._turn)
    return C4Board(temp_position, self._turn.opposite)

@property
def legal_moves(self) -> List[Move]:
    return [Move(c) for c in range(C4Board.NUM_COLUMNS) if not
        self.position[c].full]
```

Вспомогательный метод `_count_segment()` возвращает количество черных и красных фишек в определенном сегменте. Дальше идет метод проверки выигрыша `is_win()`, который просматривает все сегменты на поле и определяет, была ли игра выиграна, используя метод `_count_segment()`, чтобы подсчитать, есть ли в каком-либо сегменте четыре фишки одного цвета.

Листинг 8.17. connectfour.py (продолжение)

```
# Возвращает количество красных и черных фишек в сегменте
def _count_segment(self, segment: List[Tuple[int, int]]) -> Tuple[int, int]:
    black_count: int = 0
    red_count: int = 0
    for column, row in segment:
        if self.position[column][row] == C4Piece.B:
            black_count += 1
        elif self.position[column][row] == C4Piece.R:
            red_count += 1
    return black_count, red_count

@property
def is_win(self) -> bool:
    for segment in C4Board.SEGMENTS:
        black_count, red_count = self._count_segment(segment)
        if black_count == 4 or red_count == 4:
            return True
    return False
```

Как и `TTTBoard`, `C4Board` может использовать свойство `is_draw` абстрактного базового класса `Board` без изменений.

Чтобы оценить позицию, мы по очереди оценим все представляющие ее сегменты, суммируем оценки и вернем результат. Сегмент с красными и черными фишками будет считаться бесполезным. Сегмент, имеющий два поля с фишками одного цвета и два пустых поля, получит 1 балл. Сегмент с тремя фишками одного цвета будет оценен в 100 баллов. Наконец, сегмент с четырьмя фишками одного цвета (победа) набирает 1 000 000 баллов. Если сегмент принадлежит противнику, то его очки вычитаются. Функция `_evaluate_segment()` — вспомогательный метод, который оценивает сегмент с применением предыдущей формулы. Суммарная оценка всех сегментов методом `_evaluate_segment()` выполняется с помощью `evaluate()` (листинг 8.18).

Листинг 8.18. connectfour.py (продолжение)

```
def _evaluate_segment(self, segment: List[Tuple[int, int]], player: Piece) ->
  float:
    black_count, red_count = self._count_segment(segment)
    if red_count > 0 and black_count > 0:
        return 0 # смешанные сегменты нейтральны
    count: int = max(red_count, black_count)
    score: float = 0
```

```
    if count == 2:
        score = 1
    elif count == 3:
        score = 100
    elif count == 4:
        score = 1000000
    color: C4Piece = C4Piece.B
    if red_count > black_count:
        color = C4Piece.R
    if color != player:
        return -score
    return score

def evaluate(self, player: Piece) -> float:
    total: float = 0
    for segment in C4Board.SEGMENTS:
        total += self._evaluate_segment(segment, player)
    return total

def __repr__(self) -> str:
    display: str = ""
    for r in reversed(range(C4Board.NUM_ROWS)):
        display += "|"
        for c in range(C4Board.NUM_COLUMNS):
            display += f"{self.position[c][r]}" + "|"
        display += "\n"
    return display
```

8.3.2. ИИ для Connect Four

Удивительно, но функции `minimax()` и `find_best_move()`, которые мы разработали для игры в крестики-нолики, можно без изменений использовать в реализации Connect Four. В следующем фрагменте кода (листинг 8.19) добавилась лишь пара изменений по сравнению с кодом ИИ для игры в крестики-нолики. Главное различие заключается в том, что значение `max_depth` теперь равно 3. Это обеспечивает разумное время, отводимое компьютеру на обдумывание хода. Другими словами, наш ИИ для Connect Four рассматривает (оценивает) позиции не более чем на три хода вперед.

Листинг 8.19. connectfour_ai.py

```
from minimax import find_best_move
from connectfour import C4Board
from board import Move, Board

board: Board = C4Board()

def get_player_move() -> Move:
    player_move: Move = Move(-1)
```

```
    while player_move not in board.legal_moves:
        play: int = int(input("Enter a legal column (0-6):"))
        player_move = Move(play)
    return player_move

if __name__ == "__main__":
    # главный цикл игры
    while True:
        human_move: Move = get_player_move()
        board = board.move(human_move)
        if board.is_win:
            print("Human wins!")
            break
        elif board.is_draw:
            print("Draw!")
            break
        computer_move: Move = find_best_move(board, 3)
        print(f"Computer move is {computer_move}")
        board = board.move(computer_move)
        print(board)
        if board.is_win:
            print("Computer wins!")
            break
        elif board.is_draw:
            print("Draw!")
            break
```

Попробуйте сыграть в Connect Four с этим ИИ. Вы заметите, что генерация компьютером хода занимает несколько секунд, в отличие от ИИ для игры в крестики-нолики. Компьютер, вероятно, все еще будет выигрывать, и единственный способ победить его — тщательно продумывать свои ходы. Во всяком случае, ИИ не будет делать совершенно очевидных ошибок. Мы можем улучшить его игру, увеличив глубину поиска, но тогда время, затрачиваемое компьютером на каждый ход, будет экспоненциально увеличиваться.

СОВЕТ

Знаете ли вы, что специалисты по информатике решили игру Connect Four? Решить игру — значит найти наилучший ход в любой позиции. Наилучший первый ход в Connect Four — поместить свою фишку в центральный столбец.

8.3.3. Улучшение минимакса с помощью альфа-бета-отсечения

Минимакс работает хорошо, но сейчас нам не нужен глубокий поиск. Существует небольшое расширение минимакса, известное как *альфа-бета-отсечение*, которое позволяет улучшить глубину поиска, исключая те позиции, которые не приведут

к улучшению по сравнению с уже найденными позициями. Магия достигается отслеживанием между рекурсивными вызовами минимакса двух значений, которые получили названия «альфа» и «бета». Альфа представляет собой оценку наилучшего максимизирующего хода, найденного до этого момента в дереве поиска, а бета — оценку наилучшего минимизирующего хода, найденного для противника. Если в какой-то момент бета окажется меньше альфы или равной ей, то исследовать эту ветвь поиска далее нет смысла, поскольку уже найденный ход — лучший или как минимум не хуже тех, что могут быть найдены далее по этой ветке. Такая эвристика значительно сокращает пространство поиска.

Далее показана функция `alphabeta()`, работающая по только что описанному алгоритму (листинг 8.20). Ее следует поместить в уже существующий файл `minimax.py`.

Листинг 8.20. minimax.py (продолжение)

```
def alphabeta(board: Board, maximizing: bool, original_player: Piece,
        max_depth: int = 8, alpha: float = float("-inf"), beta: float =
        float("inf")) -> float:
        # Базовый случай – достигнута финальная позиция
        # или максимальная глубина поиска
    if board.is_win or board.is_draw or max_depth == 0:
        return board.evaluate(original_player)

    # Рекурсивный случай – максимизируйте свою выгоду
    # или минимизируйте выгоду противника
    if maximizing:
        for move in board.legal_moves:
            result: float = alphabeta(board.move(move), False, original_
              player, max_depth - 1, alpha, beta)
            alpha = max(result, alpha)
            if beta <= alpha:
                break
    return alpha
    else: # минимизация
        for move in board.legal_moves:
            result = alphabeta(board.move(move), True, original_player,
              max_ depth - 1, alpha, beta)
            beta = min(result, beta)
            if beta <= alpha:
                break
        return beta
```

Теперь мы можем внести два небольших изменения, чтобы воспользоваться новой функцией. Измените `find_best_move()` в `minimax.py`, задействуя `alphabeta()` вместо `minimax()`, и замените глубину поиска в `connectfour_ai.py` на 5 с 3. Теперь средний игрок в Connect Four больше не сможет выиграть у ИИ. На моем компьютере, используя `minimax()` с глубиной 5, ИИ для Connect Four тратил около трех

минут на ход, в то время как с `alphabeta()` при той же глубине ход занимал около 30 секунд. Одна шестая затраченного времени — невероятное улучшение!

8.4. Другие улучшения минимакса

В этой главе представлены глубоко изученные алгоритмы, и за последние годы для них было найдено много улучшений. Некоторые из этих улучшений относятся к конкретной игре, например «битовая доска» в шахматах, которая позволяет сократить время на генерирование правильных ходов, но большинство являются общими методами, которые можно использовать для любой игры.

Одним из распространенных методов является итеративное углубление. В этом случае функция поиска выполняется сначала с максимальной глубиной, равной 1. Затем она запускается с максимальной глубиной 2, 3 и т. д. По истечении указанного времени поиск прекращается и возвращается результат поиска для последней выбранной глубины.

Примеры, приведенные в этой главе, были жестко закодированы до определенной глубины. Это нормально, если игра идет без применения часов и ограничений по времени или если нам все равно, сколько времени потребуется компьютеру на «подумать». Итеративное углубление позволяет задать для ИИ фиксированное время поиска следующего хода вместо фиксированной глубины поиска с переменным количеством времени на выполнение хода.

Другое потенциальное улучшение — поиск покоя. При использовании этой технологии минимаксное дерево поиска дополнительно расширяется в тех направлениях, которые вызывают наибольшие изменения в положении (например, в шахматах), а не в тех, которые имеют относительно «тихие» позиции. Таким образом, в идеале алгоритм поиска не будет тратить время на вычисления скучных позиций, которые вряд ли принесут игроку значительное преимущество.

Два наилучших способа улучшить минимаксный поиск — это пройти бóльшую глубину за отведенное время или улучшить функцию, применяемую для оценки позиции. Перебор большего количества позиций за неизменное время позволяет тратить меньше времени на каждую позицию. Это может быть связано с повышением эффективности кода или использованием более быстрого аппаратного обеспечения, но может происходить и за счет второго способа — улучшения оценки каждой позиции. Применение большего количества параметров или эвристики для оценки позиции может занять больше времени, но в итоге привести к созданию лучшего механизма, которому требуется меньшая глубина поиска, чтобы найти хороший ход.

Некоторые функции оценки, используемые для поиска минимакса с альфа-бета-отсечением в шахматах, имеют десятки эвристик. Для настройки эвристик даже применялись генетические алгоритмы. Сколько стоит ход со взятием коня в шахматах? Равно ли данное значение стоимости взятия слона? Эти эвристики могут быть тем секретным соусом, который отличает отличный шахматный движок от хорошего.

8.5. Реальные приложения

Минимакс в сочетании с дополнительными расширениями, такими как альфа-бета-отсечение, является основой большинства современных шахматных движков. Этот алгоритм успешно применяется в разнообразных стратегических играх. В сущности, большинство искусственных противников в компьютерных играх основаны на той или иной форме минимакса.

Минимакс (с расширениями, такими как альфа-бета-отсечение) оказался настолько эффективным в шахматах, что привел к громкому поражению чемпиона мира по шахматам среди людей Гарри Каспарова в 1997 году от шахматного компьютера Deep Blue, созданного компанией IBM. Этот долгожданный матч стал событием, которое в корне изменило ситуацию. Прежде шахматы считались самой высокоинтеллектуальной игрой. Тот факт, что компьютер смог превзойти в ней человека, показал, что к искусственному интеллекту следует относиться серьезно.

Спустя два десятилетия подавляющее большинство шахматных движков попрежнему основаны на той или иной версии минимакса. Сегодняшние минимаксные шахматные движки намного превосходят возможности лучших шахматистов мира. Новые методы машинного обучения бросают вызов чисто шахматным движкам, построенным на основе минимакса (с расширениями), но им еще предстоит доказать свое превосходство в шахматах.

Чем выше коэффициент ветвления в игре, тем меньше эффективность минимакса. Коэффициент ветвления — это среднее количество потенциальных ходов в данной позиции игры. Вот почему последние достижения в компьютерной реализации игры го потребовали изучения других областей, таких как машинное обучение. Недавно искусственный интеллект для го, построенный на основе машинного обучения, победил лучшего игрока. Коэффициент ветвления (и, следовательно, пространство поиска) для го является просто непосильным для алгоритмов, основанных на минимаксном алгоритме, которые пытаются генерировать деревья, содержащие будущие позиции игры. Но го является скорее исключением, чем правилом. Большинство традиционных настольных игр — шашки, шахматы, Connect Four, скрэбл и т. п. — имеют довольно малое пространство поиска, так что методы, основанные на минимаксных алгоритмах, в них хорошо работают.

Если вам нужно реализовать нового искусственного противника в настольной игре или даже построить ИИ для пошаговой компьютерной игры, то минимакс, вероятно, первый алгоритм, к которому следует обратиться. Минимакс можно использовать для экономических и политических симуляций, а также для экспериментов в области теории игр. Альфа-бета-отсечение должно работать для любой формы минимакса.

8.6. Упражнения

1. Напишите модульные тесты для игры в крестики-нолики, чтобы убедиться в правильной работе свойств `legal_moves`, `is_win` и `is_draw`.
2. Напишите модульные тесты, проверяющие минимакс в Connect Four.

3. Код в файлах `tictactoe_ai.py` и `connectfour_ai.py` практически идентичен. Разделите его на два метода, которые можно использовать для любой игры.
4. Измените файл `connectfour_ai.py` так, чтобы компьютер играл против самого себя. Какой игрок выиграет — первый или второй? Это каждый раз один и тот же игрок?
5. Можете ли вы путем профилирования существующего кода или иным образом оптимизировать метод оценки в `connectfour.py`, чтобы увеличить глубину поиска за то же время?
6. Используя функцию `alphabeta()`, разработанную в этой главе, в сочетании с библиотекой Python, создайте функции построения корректных шахматных ходов и поддержки состояния шахматной игры для разработки шахматного ИИ.

9 Другие задачи

В этой книге мы рассмотрели множество методов решения проблем, связанных с современными задачами разработки программного обеспечения. Все технологии изучили на примере известных задач информатики. Но не все задачи соответствуют темам предыдущих глав. В данной главе собраны известные задачи, которые не вполне вписываются в другие главы. Можете считать их бонусом: более интересные задачи с меньшим количеством строительных лесов вокруг них.

9.1. Задача о рюкзаке

Это задача оптимизации, которая касается типичной потребности, часто возникающей при вычислениях, — найти наилучший вариант использования ограниченных ресурсов при ограниченном наборе вариантов — и превращает ее в забавную историю. Вор входит в дом с намерением обчистить его. У него есть рюкзак, и он может вынести из дома только то, что туда поместится. Как узнать, что стоит положить в рюкзак? Задача проиллюстрирована на рис. 9.1.

Если бы вор имел возможность взять любое количество какой-либо вещи, то он мог бы просто разделить ценность каждого предмета на его вес, чтобы определить наиболее ценные из них, способные поместиться в доступную емкость. Но мы сделаем сценарий более реалистичным: допустим, вор не может взять часть предмета, например 2,5 телевизора. Вместо этого придумаем способ решения задачи в так называемом формате 0/1, в котором применяется другое правило: вор может или взять предмет целиком, или же не брать его вовсе.

Рис. 9.1. Взломщику необходимо решить, какие предметы украсть, потому что вместимость рюкзака ограниченна

Прежде всего определим кортеж `NamedTuple` для хранения вещей (листинг 9.1).

Листинг 9.1. knapsack.py

```
from typing import NamedTuple, List

class Item(NamedTuple):
    name: str
    weight: int
    value: float
```

Если бы мы попытались решить эту задачу методом грубой силы, нам пришлось бы рассмотреть каждую комбинацию предметов, которые можно положить в рюкзак. Те, у кого есть склонности к математике, называют это *множеством всех подмножеств*, и множество всех подмножеств для данного множества (в нашем случае множества предметов) состоит из 2^N возможных подмножеств, где N — количество предметов. Поэтому нам необходимо проанализировать 2^N комбинаций ($O(2^N)$). Для небольшого количества предметов это нормально, но для большого — не годится. Следует избегать любого подхода, если при нем задача решается за экспоненциальное число шагов.

Вместо этого будем использовать технологию, известную как *динамическое программирование*, которое по своей концепции похоже на мемоизацию (см. главу 1). Вместо того чтобы решать задачу методом грубой силы, при динамическом программировании решаются подзадачи, которые составляют большую задачу, их результаты сохраняются, а затем применяются для решения более крупной задачи.

Поскольку вместимость рюкзака рассматривается в отдельных шагах, задачу можно решить методом динамического программирования.

Например, чтобы решить задачу для рюкзака вместимостью 3 фунта и трех предметов, мы можем сначала решить задачу для вместимости 1 фунт и одного возможного предмета, потом для вместимости 2 фунта и одного возможного предмета и вместимости 3 фунта и одного возможного предмета.

Затем можем использовать полученные результаты, чтобы решить задачу для вместимости 1 фунт и двух возможных предметов, вместимости 2 фунта и двух возможных предметов, вместимости 3 фунта и двух возможных предметов. И наконец, решить задачу для всех трех возможных предметов.

На протяжении пути мы станем заполнять таблицу, которая будет предлагать нам наилучшее возможное решение для каждой комбинации предметов и вместимости рюкзака. Функция сначала заполнит таблицу, а затем на ее основе выберет решение[1].

Листинг 9.2. knapsack.py (продолжение)

```
def knapsack(items: List[Item], max_capacity: int) -> List[Item]:
# построение таблицы динамического программирования
    table: List[List[float]] = [[0.0 for _ in range(max_capacity + 1)] for _
        in range(len(items) + 1)]
    for i, item in enumerate(items):
        for capacity in range(1, max_capacity + 1):
            previous_items_value: float = table[i][capacity]
            if capacity >= item.weight: # предмет помещается в рюкзак
                value_freeing_weight_for_item: float = table[i][capacity -
                  item.weight]
                # только если этот предмет ценнее предыдущего
                table[i + 1][capacity] = max(value_freeing_weight_for_item +
                  item.value, previous_items_value)
            else: # для этого предмета нет места
                table[i + 1][capacity] = previous_items_value
    # найти решение в таблице
    solution: List[Item] = []
    capacity = max_capacity
    for i in range(len(items), 0, -1): # идем в обратном направлении
        # этот предмет уже выбран?
        if table[i - 1][capacity] != table[i][capacity]:
```

[1] Чтобы создать это решение, я изучил несколько источников, наиболее авторитетным из которых была книга: *Sedgewick R., Wayne K.* Algorithms, 2nd ed. — Addison-Wesley, 1988. Я рассмотрел несколько примеров задачи о рюкзаке типа 0/1 в «Розеттском коде», в первую очередь решение для динамического программирования на Python (http://mng.bz/kx8C), к которому в значительной степени относится данная функция, по сравнению с версией из книги о Swift (код был переведен с Python на Swift и затем снова на Python).

```
            solution.append(items[i - 1])
            # если предмет выбран, то вычитаем его вес
            capacity -= items[i - 1].weight
    return solution
```

Внутренний цикл в первой части этой функции будет выполняться $N \cdot C$ раз, где N — количество предметов, а C — максимальная вместимость рюкзака. Следовательно, алгоритм выполняется за время $O(N \cdot C)$, что для большого количества предметов значительно лучше, чем метод грубой силы. Например, для 11 описанных далее предметов алгоритм грубой силы должен был бы рассмотреть 2^{11}, или 2048, комбинаций. Представленная ранее функция динамического программирования будет выполняться 825 раз, поскольку максимальная вместимость рюкзака составляет в данном случае 75 произвольных единиц ($11 \cdot 75$). Эта разница будет расти экспоненциально по мере увеличения количества предметов.

Давайте посмотрим на решение в действии (листинг 9.3).

Листинг 9.3. knapsack.py (продолжение)

```
if __name__ == "__main__":
    items: List[Item] = [Item("television", 50, 500),
                         Item("candlesticks", 2, 300),
                         Item("stereo", 35, 400),
                         Item("laptop", 3, 1000),
                         Item("food", 15, 50),
                         Item("clothing", 20, 800),
                         Item("jewelry", 1, 4000),
                         Item("books", 100, 300),
                         Item("printer", 18, 30),
                         Item("refrigerator", 200, 700),
                         Item("painting", 10, 1000)]
    print(knapsack(items, 75))
```

Если вы проверите результаты, выведенные в консоль, то увидите, что оптимальными предметами, которые можно взять, являются картина, украшения, одежда, ноутбук, стереосистема и подсвечники. Вот пример выходных данных, показывающий наиболее ценные для вора вещи с учетом того, что вместимость рюкзака ограниченна:

```
[Item(name='painting', weight=10, value=1000), Item(name='jewelry', weight=1,
    value=4000), Item(name='clothing', weight=20, value=800),
    Item(name='laptop', weight=3, value=1000), Item(name='stereo',
    weight=35, value=400), Item(name='candlesticks', weight=2, value=300)]
```

Чтобы лучше понять, как это все работает, рассмотрим некоторые особенности данной функции:

```
for i, item in enumerate(items):
    for capacity in range(1, max_capacity + 1):
```

Для каждого возможного количества предметов мы перебираем в цикле все варианты вместимости рюкзака вплоть до максимальной. Обратите внимание на то, что я говорю «каждое возможное количество предметов», а не «каждый предмет». Когда `i` равно 2, это означает не просто предмет номер 2, а все возможные комбинации из первых двух предметов для каждой исследуемой вместимости. `item` — это следующий из рассматриваемых предметов, который можно украсть:

```
previous_items_value: float = table[i][capacity]
if capacity >= item.weight: # предмет помещается в рюкзак
```

`previous_items_value` — это значение последней комбинации предметов для текущей исследуемой вместимости рюкзака. Для каждой возможной комбинации предметов мы проверяем, можно ли добавить в рюкзак еще один элемент.

Если предмет весит больше, чем позволяет рассматриваемая вместимость рюкзака, то мы просто копируем значение для последней комбинации предметов, которую рассматривали для данной вместимости:

```
else: # для этого предмета нет места
    table[i + 1][capacity] = previous_items_value
```

В противном случае проверяем, приведет ли добавление нового предмета к более высокой стоимости всего украденного, чем последнее сочетание предметов для той вместимости рюкзака, которую мы рассматриваем. Для этого мы прибавляем стоимость предмета к значению, уже вычисленному в таблице для предыдущей комбинации предметов. Если это значение больше, чем значение для последней комбинации предметов для текущей вместимости, то вставляем его в таблицу, если же нет — вставляем последнее значение:

```
value_freeing_weight_for_item: float = table[i][capacity - item.weight]
# только если этот предмет ценнее предыдущего
table[i + 1][capacity] = max(value_freeing_weight_for_item + item.value,
    previous_items_value)
```

На этом составление таблицы завершается. Однако для того, чтобы действительно узнать, какие предметы будут в решении, нужно проделать обратную работу для максимальной вместимости и результирующей исследуемой комбинации предметов:

```
for i in range(len(items), 0, -1): # идем в обратном направлении
    # был ли предмет использован?
    if table[i - 1][capacity] != table[i][capacity]:
```

Мы начинаем с конца и перебираем таблицу справа налево, на каждом этапе проверяя, было ли изменено значение, вставленное в таблицу. Если изменено, это означает, что мы добавили новый предмет, который рассматривался в данной

комбинации, потому что эта комбинация оказалась более ценной, чем предыдущая. Поэтому мы добавляем этот предмет в решение. Кроме того, по мере перемещения по таблице вместимость рюкзака уменьшается на вес предмета:

```
solution.append(items[i - 1])
# если предмет выбран, то вычитаем его вес
capacity -= items[i - 1].weight
```

ПРИМЕЧАНИЕ

И при построении таблицы, и при поиске решения вы могли заметить увеличение и уменьшение итераторов и размера таблицы на 1. Это сделано для удобства с точки зрения программирования. Подумайте, как эта задача строится снизу вверх. В начале решения мы имеем дело с рюкзаком нулевой вместимости. Если вы подниметесь снизу вверх по таблице, то поймете, зачем нужны дополнительные строка и столбец.

Вы все еще в замешательстве? Таблица 9.1 — это то, что строит функция `knapsack()`. Для предыдущей задачи это была бы довольно большая таблица, так что вместо этого рассмотрим таблицу для рюкзака вместимостью 3 фунта и трех предметов: спичек (1 фунт), фонарика (2 фунта) и книги (1 фунт). Предположим, что эти предметы оцениваются в 5, 10 и 15 долларов соответственно.

Таблица 9.1. Пример задачи о рюкзаке для трех предметов

Предметы	0 фунтов	1 фунт	2 фунта	3 фунта
Спички (1 фунт, 5 долларов)	0	5	5	5
Фонарик (2 фунта, 10 долларов)	0	5	10	15
Книга (1 фунт, 15 долларов)	0	15	20	25

Если просматривать таблицу слева направо, то вес предметов, которые вы пытаетесь уместить в рюкзаке, увеличивается. Если просматривать таблицу сверху вниз, то количество предметов, которые вы пытаетесь поместить в рюкзак, увеличивается. В первом ряду вы пробуете поместить в рюкзак только спички. Во втором ряду выбираете наиболее ценную комбинацию из спичек и фонарика, которая может поместиться в рюкзак. В третьем ряду находите самую ценную комбинацию из всех трех предметов.

В качестве упражнения, которое поможет вам лучше понять задачу, попробуйте заполнить пустую версию этой таблицы самостоятельно, используя алгоритм, описанный в функции `knapsack()`, и эти же три предмета. Затем примените алгоритм, представленный в конце функции, чтобы выбрать из таблицы правильные предметы. Эта таблица соответствует переменной `table` в функции.

9.2. Задача коммивояжера

Проблема коммивояжера — классическая и одна из наиболее широко обсуждаемых задач во всей теории вычислений. Коммивояжер должен посетить все города на карте ровно один раз и вернуться в конце путешествия в город, с которого начал маршрут. Между любыми двумя городами существует прямая дорога, и коммивояжер может посещать города в любом порядке. Каков самый короткий маршрут для него?

Эту задачу можно представить как графовую (см. главу 4), в которой города — это вершины, а дороги между ними — ребра. Вашим первым побуждением может быть желание найти минимальное связующее дерево, как описано в главе 4. К сожалению, решить задачу коммивояжера не так просто. Минимальное связующее дерево — это кратчайший путь для соединения всех городов, но оно не обеспечивает кратчайшего пути для посещения всех городов ровно один раз.

Несмотря на то что поставленная задача кажется довольно простой, не существует алгоритма, который мог бы быстро решить ее для произвольного числа городов. Что я подразумеваю, говоря «быстро»? Я имею в виду, что есть проблема: перед нами *НП-трудная задача*. НП-трудная (недетерминированная полиномиальная трудная) задача — это задача, для которой не существует алгоритма полиномиального времени. (Требуемое время является полиномиальной функцией от размера входных данных.) По мере увеличения числа городов, которые должен посетить коммивояжер, сложность задачи растет исключительно быстро. Решить задачу для 20 городов намного труднее, чем для 10. Решить задачу идеально (оптимально) для нескольких миллионов городов, за разумное время невозможно (насколько нам известно).

ПРИМЕЧАНИЕ

Наивный подход к решению задачи коммивояжера имеет сложность O(n!). Почему это так, говорится в подразделе 9.2.2. Перед тем как читать его, рекомендую прочесть подраздел 9.2.1, потому что после попытки реализации наивного решения задачи ее сложность станет очевидной.

9.2.1. Наивный подход

Наивный подход к решению задачи — просто перепробовать все возможные перестановки городов. Попробуем использовать наивный подход, чтобы проиллюстрировать сложность задачи и непригодность такого решения для применения в более крупных масштабах.

Тестовые данные

В нашей версии задачи коммивояжер заинтересован в посещении пяти крупных городов штата Вермонт. Мы не будем указывать начальный (а следовательно,

конечный) город. На рис. 9.2 показаны пять городов и расстояния между ними. Обратите внимание на то, что для построения маршрута указано расстояние между каждой парой городов.

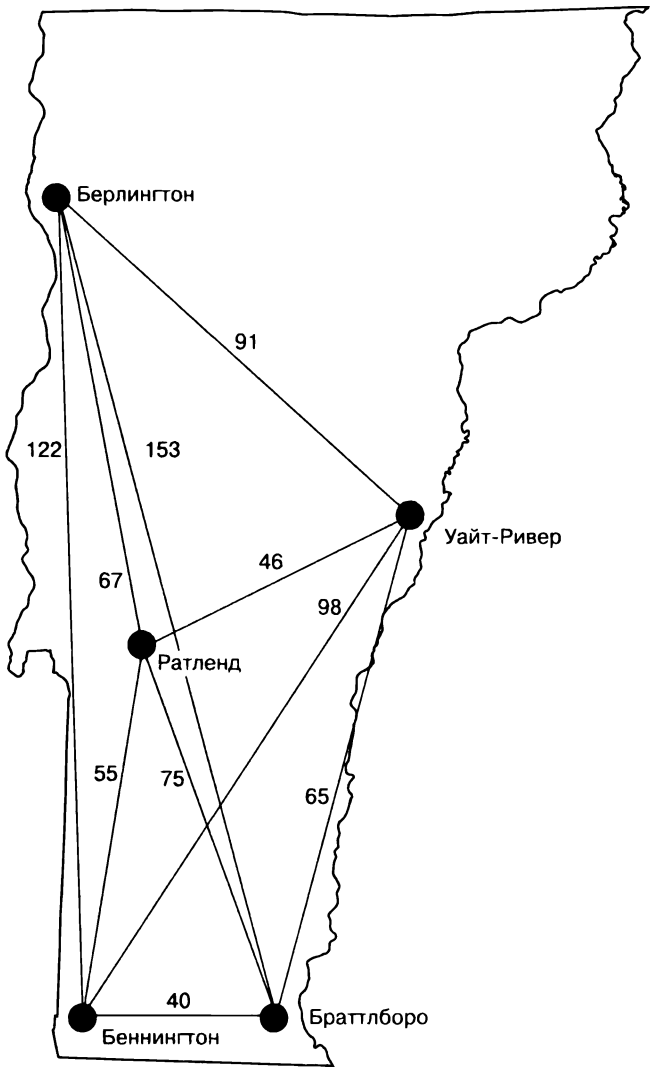


Рис. 9.2. Пять городов штата Вермонт и расстояния между ними

Возможно, вам уже встречались таблицы расстояний. В них можно легко найти расстояние между любыми двумя городами. В табл. 9.2 перечислены расстояния для пяти городов, используемых в задаче.

Таблица 9.2. Расстояния между городами штата Вермонт

Город	Ратленд	Берлингтон	Уайт-Ривер	Беннингтон	Браттлборо
Ратленд	0	67	46	55	75
Берлингтон	67	0	91	122	153
Уайт-Ривер	46	91	0	98	65
Беннингтон	55	122	98	0	40
Браттлборо	75	153	65	40	0

Для решения задачи нужно кодифицировать города и расстояния между ними. Чтобы облегчить поиск расстояний между городами, мы будем использовать словарь словарей с множеством внешних ключей, представляющих первый город пары, и множеством внутренних ключей, представляющих ее второй город. Это будет тип `Dict[str, Dict[str, int]]`, и он будет допускать поиск типа `vt_distances["Rutland"]["Burlington"]`, который должен возвращать `67` (листинг 9.4).

Листинг 9.4. tsp.py

```
from typing import Dict, List, Iterable, Tuple
from itertools import permutations

vt_distances: Dict[str, Dict[str, int]] = {
    "Rutland":
        {"Burlington": 67,
        "White River Junction": 46,
        "Bennington": 55,
        "Brattleboro": 75},
    "Burlington":
        {"Rutland": 67,
        "White River Junction": 91,
        "Bennington": 122,
        "Brattleboro": 153},
    "White River Junction":
        {"Rutland": 46,
        "Burlington": 91,
        "Bennington": 98,
        "Brattleboro": 65},
    "Bennington":
        {"Rutland": 55,
        "Burlington": 122,
        "White River Junction": 98,
        "Brattleboro": 40},
    "Brattleboro":
        {"Rutland": 75,
        "Burlington": 153,
        "White River Junction": 65,
        "Bennington": 40}
```

Перебор всех вариантов

Наивный подход к решению задачи коммивояжера требует сгенерировать все возможные варианты перестановок городов. Существует множество алгоритмов генерации перестановок, они довольно просты, так что вы почти наверняка сможете придумать такой алгоритм самостоятельно.

Один из типичных подходов — поиск с возвратом. Впервые он встретился нам в главе 3 в контексте решения задачи с ограничениями. При решении такой задачи поиск с возвратом используется после того, как найдено частичное решение, которое не удовлетворяет ограничениям задачи. В таком случае вы возвращаетесь к более раннему состоянию и продолжаете поиск по пути, отличному от того, который привел к частично неверному решению.

Найти все перестановки элементов списка (в частности, наших городов), также можно с помощью поиска с возвратом. Чтобы после перестановки элементов перейти к последующим перестановкам, можно вернуться к состоянию, предшествовавшему тому, при котором была выполнена перестановка, и выбрать другую комбинацию для перехода по другому пути.

К счастью, нет необходимости изобретать колесо и писать алгоритм генерации перестановок, потому что в стандартной библиотеке Python в модуле `itertools` есть функция `permutations()`. В следующем фрагменте кода (листинг 9.5) генерируются все перестановки городов штата Вермонт, которые должен будет посетить наш коммивояжер. Поскольку городов всего пять, то это будет 5! (5 факториал), или 120 перестановок.

Листинг 9.5. tsp.py (продолжение)

```
vt_cities: Iterable[str] = vt_distances.keys()
city_permutations: Iterable[Tuple[str, ...]] = permutations(vt_cities)
```

Поиск методом грубой силы

Теперь мы можем сгенерировать все перестановки для списка городов, но это не совсем то же самое, что путь для задачи коммивояжера. Напомню, что в задаче о коммивояжере продавец должен в конце вернуться в тот город, с которого начал. Мы можем легко добавить первый город в перестановке в ее конец, используя генератор списков (листинг 9.6).

Листинг 9.6. tsp.py (продолжение)

```
tsp_paths: List[Tuple[str, ...]] = [c + (c[0],) for c in city_permutations]
```

Теперь мы готовы проверить все сгенерированные пути. Поиск методом грубой силы просматривает каждый путь в списке и, используя расстояние между двумя городами в таблице поиска (`vt_distances`), вычисляет суммарное расстояние для

каждого пути. Функция выводит кратчайший путь и суммарное расстояние для него (листинг 9.7).

Листинг 9.7. tsp.py (продолжение)

```
if __name__ == "__main__":
    best_path: Tuple[str, ...]
    min_distance: int = 99999999999 # произвольное большое число
    for path in tsp_paths:
        distance: int = 0
        last: str = path[0]
        for next in path[1:]:
            distance += vt_distances[last][next]
            last = next
        if distance < min_distance:
            min_distance = distance
            best_path = path
    print(f"The shortest path is {best_path} in {min_distance} miles.")
```

Наконец, мы можем перебрать методом грубой силы все города Вермонта и найти кратчайший путь через заданные пять городов. Результат работы программы должен выглядеть примерно так, а лучший путь показан на рис. 9.3.

```
The shortest path is ('Rutland', 'Burlington', 'White River Junction',
  'Brattleboro', 'Bennington', 'Rutland') in 318 miles.
```

9.2.2. Переходим на следующий уровень

У задачи коммивояжера нет простого решения. По мере увеличения количества городов наивный подход быстро становится нереальным. Количество генерируемых перестановок равно факториалу от n ($n!$), где n — количество городов в задаче. Если бы мы добавили еще один город и их стало бы не пять, а шесть, то число оцененных маршрутов увеличилось бы в шесть раз. А при добавлении еще одного города решить задачу стало бы в семь раз сложнее. Этот подход не масштабируется!

В реальном мире наивный подход к задаче коммивояжера используется крайне редко. Большинство алгоритмов для различных вариаций этой задачи с большим количеством городов являются приближенными. Они пытаются получить решение, близкое к оптимальному. Оно может находиться в пределах небольшой заранее известной полосы, к которой принадлежит идеальное решение (например, возможно, такие решения будут менее чем на 5 % менее эффективными, чем идеальное).

Для решения задачи коммивояжера на больших наборах данных были использованы два метода, уже описанные в этой книге. Первый — динамическое программирование, которое мы применили в этой главе, рассматривая задачу о рюкзаке. Второй — генетические алгоритмы, описанные в главе 5. Было опубликовано множество журнальных статей, в которых генетические алгоритмы называют почти оптимальным решением задачи коммивояжера с большим количеством городов.

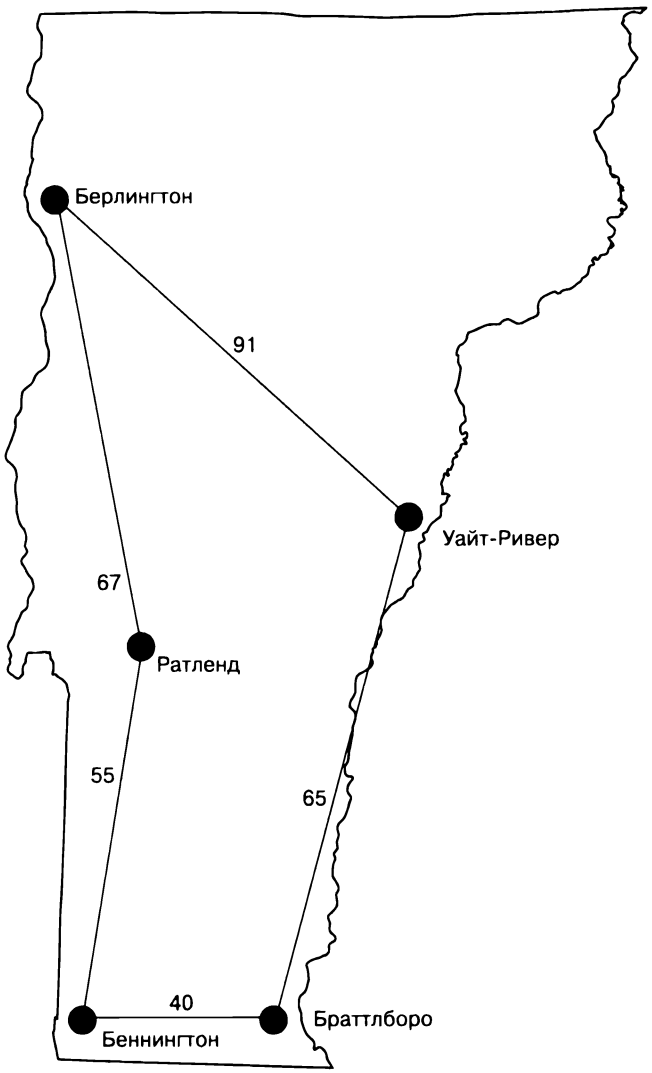


Рис. 9.3. Кратчайший путь для коммивояжера, позволяющий посетить все пять городов в штате Вермонт

9.3. Мнемоника для телефонных номеров

До того как появились смартфоны со встроенной адресной книгой, на каждой кнопке телефона кроме цифр значилось еще несколько букв. Причиной этого было обеспечение возможности выработать простые мнемонические правила для запоминания телефонных номеров. В Соединенных Штатах, как правило, на клавише 1

букв не было, на клавише 2 стояло ABC, на клавише 3 — DEF, на клавише 4 — GHI, на клавише 5 — JKL, на клавише 6 — MNO, на клавише 7 — PQRS, на клавише 8 — TUV, на клавише 9 — WXYZ, на клавише 0 букв также не было. Таким образом, номер 1-800-MY-APPLE соответствует номеру телефона 1-800-69-27753. Такие сочетания все еще встречаются в рекламных объявлениях, поэтому цифры с клавиатуры попали в современные приложения для смартфонов (рис. 9.4).

Рис. 9.4. В приложении Phone для iOS сохранены буквы на клавишах, которые были в старых телефонах

Как придумать новое мнемоническое правило для номера телефона? В 1990-х годах было популярно условно-бесплатное приложение, позволяющее это сделать. Разные части данной программы генерировали все перестановки букв телефонного номера и затем просматривали словарь, чтобы найти слова, которые содержали эти перестановки. Затем программа показывала пользователю перестановки с наиболее полными словами. Мы решим первую половину этой задачи. Поиск в словаре останется в качестве самостоятельного упражнения.

В предыдущей задаче, рассматривая генерацию перестановок, мы использовали функцию `permutations()` для нахождения потенциальных путей решения задачи коммивояжера. Однако, как уже упоминалось, существует много способов генерации перестановок. В частности, в этой задаче для создания новой перестановки мы не будем менять местами две позиции в существующей, а сгенерируем каждую переста-

новку с нуля. Сделаем это, просмотрев потенциальные буквы, которые соответствуют всем цифрам телефонного номера, и, переходя к каждой последующей цифре, станем добавлять новые варианты в конец списка. Это своего рода декартово произведение, и модуль `itertools` стандартной библиотеки Python снова нам поможет.

Сначала определим соответствие чисел и потенциальных букв (листинг 9.8).

Листинг 9.8. tsp.py (продолжение)

```
from typing import Dict, Tuple, Iterable, List
from itertools import product

phone_mapping: Dict[str, Tuple[str, ...]] = {"1": ("1",),
                                             "2": ("a", "b", "c"),
                                             "3": ("d", "e", "f"),
                                             "4": ("g", "h", "i"),
                                             "5": ("j", "k", "l"),
                                             "6": ("m", "n", "o"),
                                             "7": ("p", "q", "r", "s"),
                                             "8": ("t", "u", "v"),
                                             "9": ("w", "x", "y", "z"),
                                             "0": ("0",)}
```

Следующая функция объединяет все эти возможности для каждой цифры в список возможных мнемонических правил для данного телефонного номера. Для этого создается список кортежей потенциальных букв для каждой цифры в телефонном номере. Затем эти кортежи объединяются посредством функции декартова произведения `product()` из модуля `itertools`. Обратите внимание на оператор `unpack(*)` для использования кортежей в `letter_tuples` в качестве аргументов для функции `product()` (листинг 9.9).

Листинг 9.9. tsp.py (продолжение)

```
def possible_mnemonics(phone_number: str) -> Iterable[Tuple[str, ...]]:
    letter_tuples: List[Tuple[str, ...]] = []
    for digit in phone_number:
        letter_tuples.append(phone_mapping.get(digit, (digit,)))
    return product(*letter_tuples)
```

Теперь мы можем найти все возможные мнемонические правила для данного номера телефона (листинг 9.10).

Листинг 9.10. tsp.py (продолжение)

```
if __name__ == "__main__":
    phone_number: str = input("Enter a phone number:")
    print("Here are the potential mnemonics:")
    for mnemonic in possible_mnemonics(phone_number):
        print("".join(mnemonic))
```

Оказывается, номер телефона 144-07-87 можно записать как 1GH0STS, что легче запомнить.

9.4. Реальные приложения

Динамическое программирование, с помощью которого решалась задача о рюкзаке, — это широко используемый метод, который позволяет решить задачи, на первый взгляд кажущиеся нерешаемыми, путем разбиения их на более мелкие задачи и составления решения из этих частей. Сама задача о рюкзаке связана с другими оптимизационными задачами, где конечное количество ресурсов (вместимость рюкзака) должно быть распределено среди конечного, но исчерпывающего набора вариантов (предметов, которые можно украсть). Представьте колледж, который должен освоить свой спортивный бюджет. У него недостаточно денег для финансирования всех команд и есть ожидания относительно того, сколько пожертвований для выпускников внесет каждая команда. Это может привести к возникновению похожей на задачу о рюкзаке проблемы оптимизации распределения бюджета. Подобные задачи часто встречаются в реальном мире.

Задача коммивояжера — типичное явление для таких компаний, как UPS и FedEx. Компании по доставке посылок хотят, чтобы их машины двигались по наикратчайшим маршрутам. Это не только делает работу водителей более приятной, но и экономит топливо и уменьшает расходы на техническое обслуживание. Мы все путешествуем во время работы или для удовольствия, и поиск оптимальных маршрутов при посещении нескольких мест помогает экономить ресурсы. Но задача коммивояжера не сводится к одной лишь маршрутизации путешествий, она встречается практически в любом сценарии маршрутизации, который требует единичных посещений узлов. Несмотря на то что минимальное связующее дерево (см. главу 4) может свести к минимуму длину кабелей, необходимых для соединения соседних узлов, оно не сообщает нам оптимальную длину проводов, если каждый дом должен быть напрямую подключен только к одному другому дому как части гигантской сети, которая замыкается на свою начальную точку. Эту проблему решает задача коммивояжера.

Методы генерации перестановок, подобные используемым при наивном подходе к решению задачи коммивояжера и задачи мнемоники телефонных номеров, полезны для тестирования всевозможных алгоритмов перебора. Например, если вы пытаетесь взломать короткий пароль, то можете сгенерировать все возможные перестановки символов, которые потенциально в нем присутствуют. Специалистам, выполняющим масштабные задачи генерации перестановок, стоит применять очень эффективный алгоритм генерации перестановок, такой как Heap[1].

9.5. Упражнения

1. Перепишите реализацию наивного подхода к задаче коммивояжера, используя графовую структуру из главы 4.
2. Реализуйте генетический алгоритм (см. главу 5) для решения задачи коммивояжера. Начните с простого набора данных о городах штата Вермонт, описанного

[1] *Sedgewick R.* Permutation Generation Methods. — Princeton University. http://mng.bz/87Te.

в этой главе. Сможете ли вы получить генетический алгоритм для достижения оптимального решения в короткие сроки? Затем попытайтесь решить задачу, постепенно увеличивая количество городов. Хорошо ли работает генетический алгоритм? В Интернете вы найдете большое количество наборов данных, специально созданных для решения задачи коммивояжера. Разработайте структуру тестирования для проверки эффективности своего метода.

3. Подключите словарь к программе мнемоники телефонных номеров и возвращайте только те перестановки, которые содержат допустимые слова из словаря.

Приложение А. Глоссарий

В этом приложении даются определения ключевых терминов, использованных в книге.

CSV. Формат обмена текстовыми данными, где строки представляют собой наборы данных, значения которых разделены запятыми, а сами строки обычно разделяются символами новой строки. Аббревиатура CSV расшифровывается как *comma-separated values* — «значения, разделенные запятыми». CSV является распространенным форматом для экспорта данных из электронных таблиц и баз данных (см. главу 7).

NP-трудный. Задача, относящаяся к классу задач, для которых не существует известного алгоритма с полиномиальным временем решения (см. главу 9).

SIMD-инструкции. Инструкции микропроцессора, оптимизированные для выполнения вычислений с использованием векторов, иногда называемые также векторными инструкциями. Аббревиатура *SIMD* расшифровывается как *single instruction, multiple data* — один поток инструкций, много потоков данных (см. главу 7).

XOR. Побитовая логическая операция, которая возвращает `true`, если любой ее операнд имеет значение `True`, но не тогда, когда оба операнда равны `True` или ни один из них не равен `True`. Аббревиатура XOR расшифровывается как *exclusive or* — исключающее ИЛИ. В Python для обозначения операции XOR используется оператор `^` (см. главу 1).

Z-оценка. Количество стандартных отклонений единицы данных от среднего значения набора данных (см. главу 6).

Автомемоизация. Вариант *мемоизации*, реализованный на уровне языка, в котором результаты вызовов функций без побочных эффектов сохраняются для использования при последующих идентичных вызовах (см. главу 1).

Ациклический. *Граф* без циклов (см. главу 4).

Бесконечная рекурсия. Множество рекурсивных вызовов, которые не завершаются, а продолжают выполнять дополнительные рекурсивные вызовы. Аналог *бесконечного цикла*. Обычно вызвана отсутствием базового случая (см. главу 1).

Бесконечный цикл. Цикл, который не заканчивается (см. главу 1).

Битовая строка. Структура данных, которая хранит последовательность единиц и нулей, на каждый из которых отводится 1 бит памяти. Иногда битовые строки называют *битовыми векторами* или *битовыми массивами* (см. главу 1).

Вершина. Отдельный узел *графа* (см. главу 4).

Входной слой. Первый слой *искусственной нейронной сети с прямой связью*, который получает входные данные от какого-либо внешнего объекта (см. главу 7).

Выходной слой. Последний слой в искусственной нейронной сети с прямой связью, который используется для определения результата сети для конкретного набора входных данных и конкретной задачи (см. главу 7).

Генетическое программирование. Программы, которые модифицируют сами себя с помощью операторов *отбора*, *кроссинговера* и *мутации*, чтобы найти неочевидные решения задач программирования (см. главу 5).

Глубокое обучение. Еще одно модное слово. Глубоким обучением могут называть любой из нескольких методов, в которых используются передовые алгоритмы машинного обучения для анализа больших данных. Чаще всего глубокое обучение означает применение многослойных *искусственных нейронных сетей* для решения задач с большими наборами данных (см. главу 7).

Градиентный спуск. Метод изменения весов в *искусственной нейронной сети* с использованием *дельт*, рассчитанных во время *обратного распространения*, и *скорости обучения* (см. главу 7).

Граф. Абстрактная математическая конструкция, которая применяется для моделирования реальной задачи путем разделения этой задачи на множество *связных* узлов. Эти узлы называются *вершинами*, а соединения между ними — *ребрами* (см. главу 4).

Декомпрессия. Отмена процесса *компрессии*, возврат данных в исходную форму (см. главу 1).

Дельта. Значение, описывающее разрыв между ожидаемым значением веса в *нейронной сети* и его фактическим значением. Ожидаемое значение определяется с помощью данных *обучения* и *обратного распространения* (см. главу 7).

Дерево. *Граф*, который имеет только один *путь* между любыми двумя вершинами. Дерево является *ацикличным* (см. главу 4).

Диграф. См. *Направленный граф* (см. главу 4).

Динамическое программирование. Вместо решения большой задачи напрямую, методом грубой силы, в динамическом программировании задача разбивается на более мелкие подзадачи, каждая из которых лучше управляема (см. главу 9).

Допустимая эвристика. *Эвристика* алгоритма поиска A*, которая никогда не переоценивает затраты на достижение цели (см. главу 2).

Естественный отбор. Процесс эволюции, благодаря которому хорошо адаптированные организмы процветают, а плохо адаптированные терпят неудачу. Учитывая ограниченный набор ресурсов в окружающей среде, те организмы, которые лучше всего подходят для использования этих ресурсов, будут выживать и размножаться. На протяжении нескольких *поколений* это приводит к тому, что полезные черты распространяются в *популяции*, а следовательно, естественным образом отбираются за счет ограничений окружающей среды (см. главу 5).

Жадный алгоритм. Алгоритм, который в любой момент принятия решения делает наилучший немедленный выбор, рассчитывая, что это приведет к глобально оптимальному решению (см. главу 4).

Исключающее ИЛИ. См. *XOR* (см. главу 1).

Искусственная нейронная сеть. Симуляция биологической *нейронной сети* с использованием вычислительных инструментов для решения задач, которые не удается легко представить в форме, поддающейся традиционным алгоритмическим подходам. Обратите внимание на то, что работа *искусственной нейронной сети* обычно значительно отличается от ее биологического аналога (см. главу 7).

Кластер. См. *Кластеризация* (см. главу 6).

Кластеризация. Методика *неконтролируемого обучения*, при которой набор данных делится на группы связанных точек, известных как *кластеры* (см. главу 6).

Кодон. Комбинация из трех *нуклеотидов*, которые образуют аминокислоту (см. главу 2).

Компрессия. Кодирование (изменение формы) данных, после чего они занимают меньше места (см. главу 1).

Контролируемое обучение. Любая технология машинного обучения, в которой алгоритм каким-либо образом направляется на получение правильных результатов с использованием внешних ресурсов (см. главу 7).

Кроссинговер. В генетическом алгоритме — объединение особей *популяции* для создания потомков, которые являются смесью родителей и будут частью следующего *поколения* (см. главу 5).

Мемоизация. Методика, в которой результаты вычислительных задач сохраняются и впоследствии при необходимости извлекаются из памяти, экономя время вычислений, которое иначе было бы потрачено на повторное вычисление тех же результатов (см. главу 1).

Минимальное связующее дерево. *Связующее дерево*, которое соединяет все вершины при минимальном общем весе *ребер* (см. главу 4).

Мутация. В генетическом алгоритме — случайное изменение некоторого свойства особи перед тем, как включить эту особь в состав следующего *поколения* (см. главу 5).

Направленный граф. Также известный как *дираф* — это *граф*, *ребра* которого можно проходить только в одном направлении (см. главу 4).

Нейрон. Отдельная нервная клетка, такая как клетка человеческого мозга (см. главу 7).

Нейронная сеть. Сеть, состоящая из нескольких *нейронов*, которые действуют согласованно для обработки информации. *Нейроны* часто объединяются в слои (см. главу 7).

Неконтролируемое обучение. Любая технология машинного обучения, при которой не используется прогноз, чтобы сделать выводы, — другими словами, технология, которая не является управляемой, а работает сама по себе (см. главу 6).

Нормализация. Процесс, в результате которого различные типы данных становятся сравнимыми (см. главу 6).

Нуклеотид. Экземпляр одного из четырех оснований ДНК: аденина (A), цитозина (C), гуанина (G) и тимина (T) (см. главу 2).

Область определения. Возможные значения *переменной* в задаче с ограничениями (см. главу 3).

Обратное распространение. Методика, используемая для *обучения нейронной сети*, при котором веса назначаются в соответствии с набором входных данных с известными правильными выходными данными. Для вычисления степени ответственности каждого веса за отклонение реальных результатов от ожидаемых применяются частные производные. Эти *дельты* задействуются для обновления весов при последующих прогонах (см. главу 7).

Обучение. Фаза, в которой веса искусственной нейронной сети корректируются посредством обратного распространения с известными правильными выходными данными для некоторых заданных входных данных (см. главу 7).

Ограничение. Требование, которое должно быть выполнено для решения задачи с ограничениями (см. главу 3).

Отбор. Процесс отбора особей в *поколении* при выполнении генетического алгоритма для размножения с целью создания особей следующего *поколения* (глава 5).

Очередь. Абстрактная структура данных, обеспечивающая упорядочение типа FIFO («первым пришел — первым вышел»). Реализация очереди обеспечивает по крайней мере операции `push` и `pop` для добавления и удаления элементов очереди соответственно (см. главу 2).

Очередь с приоритетом. Структура данных, которая выводит элементы на основе приоритетной последовательности. Например, очередь с приоритетом может использоваться для множества экстренных вызовов, чтобы сначала отвечать на вызовы с самым высоким приоритетом (см. главу 2).

Переменная. В контексте задач с ограничениями переменная — это параметр, который должен быть выдержан в заданных рамках как часть решения задачи. Возможные значения переменной — это ее *область определения*. Требованиями к решению являются одно или несколько *ограничений* (см. главу 3).

Позиция. Ход в игре для двух игроков (см. главу 8).

Поиск с возвратом. Возвращение к более раннему моменту принятия решения в задаче поиска, чтобы пойти в другом направлении, если предыдущий поиск зашел в тупик (см. главу 3).

Поколение. Этап при выполнении генетического алгоритма; также используется для обозначения особей *популяции*, активных на данном этапе (см. главу 5).

Популяция. В генетическом алгоритме популяция — это совокупность особей, каждая из которых представляет собой потенциальное решение задачи, конкурирующих за решение задачи (см. главу 5).

Путь. Набор *ребер*, соединяющих две вершины *графа* (см. главу 4).

Ребро. Связь между двумя *вершинами* (узлами) *графа* (см. главу 4).

Рекурсивная функция. Функция, которая вызывает саму себя (см. главу 1).

Связность. Свойство графа, которое указывает на то, что существует *путь* от любой *вершины* графа к любой другой *вершине* этого графа (см. главу 4).

Связующее дерево. *Дерево*, которое соединяет все вершины *графа* (см. главу 4).

Сигмоидная функция. Одна из множества популярных *функций активации*, используемых в *искусственных нейронных сетях*. Эпонимическая сигмоидная функция всегда возвращает значение в диапазоне 0... 1. Сигмоидная функция полезна также для обеспечения способности сети возвращать не только результаты простых линейных преобразований (см. главу 7).

Синапсы. Разрывы между *нейронами*, в которых высвобождаются нейротрансмиттеры, что позволяет проводить электрический ток. С точки зрения непрофессионала, это связи между нейронами (см. главу 7).

Скорость обучения. Значение, обычно постоянное, которое используется для корректировки скорости изменения весов в *искусственной нейронной сети* на основе вычисленных *дельт* (см. главу 7).

Скрытый слой. Любой слой между входным и выходным слоями в искусственной нейронной сети с прямой связью (см. главу 7).

Стек. Абстрактная структура данных, обеспечивающая упорядочение по принципу «последним пришел — первым вышел» (last-in-first-out, LIFO). Реализация стека обеспечивает как минимум операции `push` и `pop` для добавления и удаления элементов соответственно (см. главу 2).

Упреждение. Тип *нейронной сети*, в которой сигналы распространяются в одном направлении (см. главу 7).

Функция активации. Функция, которая преобразует выходной сигнал *нейрона* в *искусственной нейронной сети*, как правило, для того, чтобы эта сеть могла обрабатывать нелинейные преобразования или ограничить выходное значение некоторым диапазоном (см. главу 7).

Функция жизнеспособности. Функция, которая оценивает эффективность потенциального решения задачи (см. главу 5).

Хромосома. В генетическом алгоритме каждая особь *популяции* называется хромосомой (см. главу 5).

Центроид. Центральная точка кластера. Как правило, каждое измерение этой точки является средним значением данного измерения для остальных точек (см. главу 6).

Цикл. *Путь* в *графе*, который дважды проходит через одну и ту же вершину без *поиска с возвратом* (см. главу 4).

Эвристика. Интуитивная догадка о том, в каком направлении следует двигаться, чтобы решить задачу (см. главу 2).

Приложение Б. Дополнительные ресурсы

Что дальше? Книга охватывает широкий спектр тем, а изучить их глубже вам помогут отличные ресурсы, ссылки на которые содержит это приложение.

Б.1. Python

Как говорилось во введении, предполагается, что вы знаете язык Python хотя бы на среднем уровне. Здесь я привожу две книги по Python, которыми пользовался сам, и рекомендую их, чтобы вы поднялись на следующий уровень знания Python. Они не подходят для тех, кто только начинает изучать Python (в этом случае обратитесь к книге *Ceder N.* The Quick Python Book (Manning, 2018)), но помогут пользователю Python среднего уровня стать опытным пользователем.

- *Ramalho L.* Fluent Python: Clear, Concise and Effective Programming. — O'Reilly, 2015:
  - одна из немногих популярных книг о языке Python, которая адресована не программистам, чей уровень — между начальным и средним или высоким, она предназначена для специалистов среднего и более высокого уровня;
  - охватывает широкий спектр расширенных возможностей Python;
  - позволяет освоить рекомендуемые методы и научит вас писать «питонический» код;
  - содержит множество примеров кода для каждой темы и разъясняет, как работает стандартная библиотека Python;
  - местами несколько многословна, но эти фрагменты вполне можно пропустить.

- *Beazley D., Jones B. K.* Python Cookbook, 3rd ed. — O'Reilly, 2013:
  - обучает решению повседневных задач программирования на примерах;
  - некоторые задачи выходят далеко за рамки задач для начинающих;
  - активно использует стандартную библиотеку Python;
  - вышла пять лет назад и уже несколько устарела (в ней не применяются новейшие стандартные инструменты библиотеки), надеюсь, скоро выйдет четвертое издание.

Б.2. Алгоритмы и структуры данных

Процитирую введение к этой книге: «Это не учебник по структурам данных и алгоритмам». В этой книге редко используется нотация О большого (big-O notation) и нет математических доказательств. Это скорее практическое руководство по важным методикам программирования. Поэтому есть смысл обзавестись и настоящим учебником, который не только даст вам более формальное объяснение того, почему работают те или иные методы, но и послужит полезным справочным пособием. Онлайн-ресурсы хороши, но иногда полезно иметь информацию, тщательно проверенную учеными и издателями.

- *Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К.* Алгоритмы: построение и анализ. 3-е изд. — М.: Вильямс, 2013 (*Cormen T., Leiserson C., Rivest R., Stein C.* Introduction to Algorithms, 3rd ed. — MIT Press, 2009), https://mitpress.mit.edu/books/introduction-algorithms-third-edition:
  - один из самых часто цитируемых текстов в области информатики, настолько исчерпывающий, что на него часто ссылаются по инициалам его авторов — CLRS;
  - комплексное и строгое изложение;
  - стиль преподнесения материала делает его не столь доступным, как другие тексты, тем не менее он является отличным справочным материалом;
  - большинство алгоритмов описано на псевдокоде;
  - сейчас готовится к выходу четвертое издание, и поскольку эта книга стоит дорого, возможно, стоит подождать его выхода.
- *Седжвик Р., Уэйн К.* Алгоритмы на Java. 4-е изд. — М.: Вильямс, 2013 (*Sedgewick R., Wayne K.* Algorithms. 4th ed. — Addison-Wesley Professional, 2011), http://algs4.cs.princeton.edu/home/:
  - доступное и вместе с тем всеобъемлющее введение в алгоритмы и структуры данных;
  - хорошо организованный материал с полными примерами всех алгоритмов на Java;
  - алгоритмы классов, широко используемые в учебных курсах колледжей.

- *Skiena S.* The Algorithm Design Manual, 2nd ed. — Springer, 2011, http://www.algorist.com:
  - своим подходом эта книга отличается от других учебников по данной дисциплине;
  - в книге содержится не очень много кода, акцент сделан на наглядном обсуждении правильного использования каждого алгоритма;
  - читателю советуют, как найти собственный путь в области применения широкого круга алгоритмов.
- *Бхаргава А.* Грокаем алгоритмы: иллюстрированное пособие для программистов и любопытствующих. — СПб.: Питер, 2017 (*Bhargava A.* Grokking Algorithms. — Manning, 2016), https://www.manning.com/books/grokking-algorithms:
  - графический подход к освоению основных алгоритмов, имеются забавные анимационные ролики, которые можно загрузить;
  - это не справочник, а руководство для тех, кто впервые приступает к изучению отдельных важных тем.

Б.3. Искусственный интеллект

Искусственный интеллект меняет наш мир. В этой книге вы познакомились не только с некоторыми традиционными технологиями поиска, применяемыми искусственным интеллектом, такими как A* и минимакс, но и с методиками из столь захватывающей субдисциплины, как машинное обучение, такими как k-средние и нейронные сети. Изучить искусственный интеллект глубже — очень интересно, а еще это позволит подготовиться к следующей волне развития компьютерных технологий.

- *Рассел С., Норвиг П.* Искусственный интеллект: современный подход. 3-е изд. — М.: Вильямс, 2019 (*Russell S., Norvig P.* Artificial Intelligence: A Modern Approach, 3rd ed. — Pearson, 2010), http://aima.cs.berkeley.edu:
  - всеобъемлющий учебник по ИИ, часто используемый в учебных курсах колледжей;
  - широкий охват тем;
  - превосходный репозиторий исходного кода (реализованные версии алгоритмов, описанных в книге на псевдокоде), доступный онлайн.
- *Lucci S., Kopec D.* Artificial Intelligence in the 21st Century, 2nd ed. — Mercury Learning and Information, 2015, http://mng.bz/1N46:
  - изложение доступно для тех, кто ищет более практичное и красочное руководство, чем у Рассела и Норвига;
  - интересные зарисовки для практиков и множество ссылок на реальные приложения.

- *Ng A.* Machine Learning, учебный курс (Стэнфордский университет), https://www.coursera.org/learn/machine-learning/:
  - бесплатный онлайн-курс, который охватывает многие фундаментальные алгоритмы машинного обучения, в изложении всемирно известного эксперта;
  - многие практики считают этот курс отличной отправной точкой в данной области.

Б.4. Функциональное программирование

На Python можно программировать в функциональном стиле, но в общем-то язык для этого не предназначен. Можно углубиться в функциональное программирование на самом Python, но может быть полезно работать и на чисто функциональном языке, а затем перенести некоторые идеи, которые вы извлекли из этого опыта, обратно в Python.

- *Abelson H., Sussman G. J., Sussman J.* Structure and Interpretation of Computer Programs. — MIT Press, 1996, https://mitpress.mit.edu/sicp/:
  - классическое введение в функциональное программирование, часто используемое во вводных курсах по информатике;
  - четко структурированный, простой для освоения, чисто функциональный язык программирования;
  - книга бесплатная, доступна онлайн.
- *Khan A.* Grokking Functional Programming. — Manning, 2018, https://www.manning.com/books/grokking-functional-programming:
  - графически оформленное понятное введение в функциональное программирование.
- *Mertz D.* Functional Programming in Python. — O'Reilly, 2015, https://www.oreilly.com/programming/free/functional-programming-python.csp:
  - книга представляет собой базовое введение в некоторые утилиты функционального программирования из стандартной библиотеки Python;
  - распространяется бесплатно;
  - содержит всего 37 страниц — не очень подробное руководство, но хорошая стартовая точка.

Б.5. Полезные проекты с открытым исходным кодом для машинного обучения

Существует несколько полезных сторонних библиотек Python, оптимизированных для эффективного машинного обучения. Несколько таких проектов упоминались в главе 7. Они предлагают больше возможностей и полезных свойств, чем вы,

вероятно, сможете разработать сами. Именно эти библиотеки (или их эквиваленты) следует использовать для серьезного машинного обучения или приложений по обработке больших данных.

- NumPy, http://www.numpy.org:
  - де-факто стандартная библиотека численных методов для Python;
  - из соображений скорости работы библиотека реализована в основном на C;
  - лежит в основе многих библиотек машинного обучения Python, включая TensorFlow и scikit-learn.
- TensorFlow, https://www.tensorflow.org:
  - одна из самых популярных библиотек Python для работы с нейронными сетями.
- pandas, https://pandas.pydata.org:
  - популярная библиотека для импорта наборов данных в Python и манипулирования ими;
- scikit-learn, http://scikit-learn.org/stable/:
  - хорошо протестированные полнофункциональные версии нескольких алгоритмов машинного обучения, описанных в этой книге.

И многие, многие другие...

Приложение B. Коротко об аннотациях типов

Подсказки типов (или аннотации типов) появились в качестве официальной части языка Python после выхода PEP 484 и Python версии 3.5. С тех пор они получили широкое распространение во многих базах исходного кода Python, а в языке появилась более надежная поддержка для них. Аннотации типов используются во всех листингах исходного кода в данной книге. В этом кратком приложении я намерен представить базовое описание аннотаций типов, объяснить, почему они полезны, описать некоторые из связанных с ними проблем и дать ссылки на более подробные ресурсы об аннотациях типов.

ПРЕДУПРЕЖДЕНИЕ

Это приложение не претендует на полноту. Это лишь краткое введение. За более подробной информацией обращайтесь к документации Python: https://docs.python.org/3/library/typing.html.

B.1. Что такое аннотации типов

Аннотации типов — это способ аннотирования ожидаемых типов переменных, параметров функций и типов возвращаемых функций в Python. Другими словами, это способ, которым программист может указать тип данных, ожидаемый в определенной части программы на Python. Большинство программ на Python написаны без аннотаций типов. На самом деле даже если вы программист на Python среднего уровня, то вполне возможно, никогда прежде не видели программу на Python с аннотациями типов.

Поскольку Python не требует от программиста указывать тип переменной, единственный способ выяснить его без аннотаций типов — это инспекция (буквально чтение исходного кода до данной точки или запуск кода и вывод типа) либо документирование. Это проблематично, поскольку затрудняет чтение кода на Python (некоторые утверждают обратное, и мы вернемся к этому вопросу позже). Другая проблема состоит в следующем: поскольку Python очень гибок, он позволяет программисту использовать одну и ту же переменную для ссылки на несколько типов объектов, что может вызвать появление ошибок. Аннотации типов способны помочь предотвратить такой стиль программирования и устранить эти ошибки.

Теперь, когда в Python есть аннотации типов, мы называем его языком с постепенной типизацией, что означает: при желании вы можете задействовать аннотации типов, но не обязаны это делать. В данном кратком введении я надеюсь убедить вас (возможно, вопреки вашему сопротивлению тому, как сильно аннотации типов меняют внешний вид языка), что наличие доступных аннотаций типов — это хорошо и это именно то, чем стоит пользоваться при написании кода.

B.2. Как выглядят аннотации типа

Аннотации типов добавляются в строку кода, где объявлена переменная или функция. Признаком начала аннотации типа для переменной или параметра функции служит двоеточие (`:`), а для типа, возвращаемого функцией, — стрелка (`->`). Например, рассмотрим следующую строку кода на Python:

```
def repeat(item, times):
```

Можете ли вы сказать, не читая определение функции, что эта функция должна делать? Распечатать строку определенное количество раз? Или что-то еще? Конечно, для того, чтобы выяснить, что должна делать функция, можно прочитать ее определение, но это займет больше времени. Да и автор этой функции, к сожалению, не предоставил никакой документации. Попробуем еще раз, с аннотациями типа:

```
def repeat(item: Any, times: int) -> List[Any]:
```

Так гораздо понятнее. Достаточно просто взглянуть на аннотации типов, и станет ясно, что данная функция принимает элемент любого типа и возвращает список `List`, заполненный этим элементом, заданное число раз. Конечно, документация по-прежнему поможет сделать функцию более понятной, но пользователь библиотеки теперь хотя бы знает, какие значения следует предоставлять этой функции и каких значений можно ожидать на выходе.

Предположим, что библиотека, в которой используется данная функция, работает только с числами с плавающей точкой и эта функция должна была служить для создания списков, которые будут применяться в других функциях. Мы можем легко изменить аннотации типов и ввести ограничение для чисел с плавающей точкой:

```
def repeat(item: float, times: int) -> List[float]:
```

Теперь ясно, что значение `item` должно иметь тип `float` и возвращаемый список должен заполняться числами типа `float`. Ну хорошо, «должен» в данном случае — слишком сильно сказано. Начиная с Python 3.7, аннотации типов не имеют никакого отношения к выполнению программы на Python. Это действительно лишь аннотации, а не требования. Во время выполнения программа на Python может полностью игнорировать аннотации типов и нарушать любые предполагаемые ограничения. Однако инструмент проверки типов может учитывать аннотации типов во время разработки программы и сообщать программисту о наличии некорректных вызовов функций. Вызов `repeat("hello", 30)` может быть замечен прежде, чем попасть в рабочую программу, потому что `"hello"` не относится к типу `float`.

Рассмотрим еще один пример. На этот раз это аннотация типа для объявления переменной:

```
myStrs: List[str] = repeat(4.2, 2)
```

Эта аннотация типа не имеет смысла. Она говорит о том, что переменная `myStrs` должна представлять собой список строк. Но, как мы знаем из предыдущей аннотации типа, `repeat()` возвращает список чисел с плавающей точкой. К тому же, поскольку Python, начиная с версии 3.7, не проверяет аннотации типов на корректность во время выполнения программы, данная ошибочная аннотация типа не повлияет на работу программы. Однако средство проверки типов может обнаружить эту ошибку программиста или неправильное представление о правильном типе, прежде чем они приведут к катастрофическим последствиям.

B.3. Почему полезны аннотации типов

Теперь, когда вы знаете, что такое аннотации типов, у вас может возникнуть вопрос: почему вокруг них так много шума? В конце концов, как вы только что узнали, Python игнорирует аннотации типов во время выполнения программы. Зачем же тратить время на добавление аннотаций типов в код, если интерпретатору Python они безразличны? Как уже отмечалось, аннотации типов хороши по двум основным причинам: они автодокументируют код и позволяют средству проверки типов проверять программу перед ее выполнением.

В большинстве языков программирования со статической типизацией, таких как Java или Haskell, необходимые объявления типов очень четко показывают, каких параметров ожидает функция (или метод) и какой тип она будет возвращать. Это до некоторой степени облегчает бремя написания документации для программиста. Например, совершенно не обязательно указывать, что следующий метод Java ожидает в качестве входных параметров или каков его возвращаемый тип:

```
/ * Функция eat_world принимает данные типа World, возвращает сумму денег,
сгенерированную как отказ * /
public float eatWorld(World w, Software s) { ... }
```

Сравните это с документацией, необходимой для эквивалентного метода, написанного на традиционном Python, без аннотаций типов:

```
# Функция eat_world
# Параметры:
# w — данные типа World, мир, который можно съесть
# s — данные типа Software, программа, позволяющая съесть мир
# Возвращает:
# количество денег, получаемых от поедания мира, в формате float
def eat_world(w, s):
```

Позволяя программисту самостоятельно документировать код, аннотации типов делают документацию Python такой же лаконичной, как и в статически типизированных языках:

```
/ * Функция eat_world принимает данные типа World, возвращает сумму денег,
сгенерированную как отказ * /
def eat_world(w: World, s: Software) -> float:
```

Это, конечно, крайность. Представьте, что вы наследуете базу кода, в которой вообще нет комментариев. Что проще: использовать базу кодов без комментариев с аннотациями типов или без аннотаций типов? Аннотации типов избавят вас от необходимости копаться в реальном коде функции, лишенной комментариев, чтобы понять, какие типы нужно ей передавать в качестве параметров и какой тип ожидать от нее на выходе.

Помните, что аннотация типа — это, по сути, способ указать, какой тип ожидается в программе в какой-то момент. Тем не менее Python ничего не делает, чтобы проверить это ожидание. Именно здесь на помощь приходит средство проверки типов. Оно принимает файл с исходным кодом Python, в котором указаны аннотации типов, и проверяет, на самом ли деле они соответствуют действительности при выполнении программы.

Существует несколько средств проверки типа для аннотаций типов Python. Например, проверка типов встроена в популярную IDE Python PyCharm. При редактировании программы с аннотациями типов в PyCharm среда автоматически указывает на ошибки типов. Это помогает обнаружить их прежде, чем вы закончите писать функцию.

На момент написания книги самой известной программой проверки типов Python была mypy. Проект mypy возглавляет Гвидо ван Россум — тот самый человек, который создал сам Python. Едва ли после этого у вас останутся сомнения в том, что аннотации типов потенциально могут сыграть очень важную роль в будущем Python. С программой mypy работать легко — достаточно запустить команду `mypy example.py`, где `example.py` — имя файла, который нужно проверить. mypy выведет в консоль все ошибки типа, найденные в программе, или не выведет ничего, если ошибок нет.

В будущем могут появиться и другие способы использования аннотаций типов. В настоящее время они не влияют на производительность работающей программы Python. (Еще раз повторяю: они игнорируются на этапе выполнения.) Но, вероятно

в будущих версиях Python информация о типах, полученная из аннотаций типов, будет применяться для выполнения оптимизации. В таком случае, возможно, вы сможете ускорить выполнение программы на Python, просто добавив аннотации типов. Конечно, пока что это только предположения. Мне ничего не известно о планах по реализации оптимизации на основе аннотаций типов в Python.

В.4. Каковы недостатки аннотаций типов

У использования аннотаций типов существует три потенциальных недостатка:

- написание кода с аннотациями типов занимает больше времени, чем без них;
- в некоторых случаях аннотации типов могут ухудшить читаемость кода;
- аннотации типов все еще довольно сырая технология, и реализация некоторых ограничений типов в существующих версиях Python может привести к путанице.

Написание кода с аннотациями типов занимает больше времени по двум причинам: просто требуется вводить больше символов (буквально нажимать больше клавиш) и внимательнее задумываться о коде, который вы пишете.

Размышления о коде — это почти всегда хорошо, но дополнительные размышления замедляют работу. Тем не менее мы надеемся, что потерянное время окупится за счет обнаружения ошибок еще до запуска программы с помощью средства проверки типов. Время, потраченное на отладку ошибок, которые могут быть обнаружены средством проверки типов, наверняка больше, чем время, затрачиваемое на обдумывание типов при компоновке любой сложной кодовой базы.

Некоторые считают, что код на Python с аннотациями типов не так удобен для чтения, как код на Python без них. Вероятно, на то есть две причины: незнание и многословие. Любой синтаксис, с которым вы не знакомы, будет менее читабельным, чем знакомый. Аннотации типов действительно изменяют внешний вид программ на Python, так что он поначалу выглядит непривычно. Чтобы этого избежать, достаточно написать и прочитать больше кода Python с аннотациями типов. Вторая проблема — многословие — более фундаментальна. Python известен своим лаконичным синтаксисом. Часто программа на Python выглядит значительно короче, чем ее эквивалент на другом языке. Код на Python с аннотациями типов менее компактен. Его нельзя столь же быстро просмотреть навскидку: там просто намного больше информации. Компромисс заключается в том, что после первого прочтения код станет более понятным, даже если чтение занимает больше времени. Благодаря аннотациям типов вы сразу видите все ожидаемые типы и можете разобраться в них, не просматривая весь код и не читая документацию.

Наконец, аннотации типов продолжают совершенствоваться. Они определенно улучшились с тех пор, как впервые появились в Python 3.5, но все же до сих пор существуют пограничные случаи, когда аннотации типов не работают должным образом. Пример этого приведен в главе 2. Тип `Protocol`, который обычно является

важной частью системы типов, все еще не включен в модуль типизации стандартной библиотеки Python, поэтому в главе 2 пришлось подключить сторонний модуль `typing_extensions`. Планируется включить тип `Protocol` в будущую версию официальной стандартной библиотеки Python, но то, что он пока туда не входит, свидетельствует: аннотации типов в Python все еще находятся на раннем этапе развития. При написании книги я столкнулся с несколькими пограничными случаями, и было непонятно, как решать эти проблемы, учитывая то, какие примитивы существуют в стандартной библиотеке. Поскольку аннотации типов в Python необязательны, на данном этапе их можно просто игнорировать в тех областях, где они неудобны. Тем не менее даже такое половинчатое использование аннотаций типов способно принести некоторую выгоду.

B.5. Источники дополнительной информации

Во всех главах этой книги приводится множество примеров аннотаций типов, но это не учебник по их применению. Лучший источник, с которого можно начать работу с аннотациями типов, — официальная документация Python для модуля ввода (https://docs.python.org/3/library/typing.html). В ней не только описаны все доступные встроенные типы, но и рассказывается, как их использовать для нескольких расширенных сценариев, выходящих за рамки этого краткого введения.

Другой ресурс для изучения аннотаций типов, к которому действительно стоит обратиться, — проект mypy (http://mypy-lang.org). Это ведущее средство проверки типов в Python. Именно с этим программным обеспечением вы будете работать при фактической проверке правильности аннотаций типов.

Помимо установки и применения mypy, вам следует ознакомиться с документацией mypy (https://mypy.readthedocs.io/). В этой обширной документации объясняется, как использовать аннотации типов в некоторых сценариях, отсутствующих в документации стандартной библиотеки. Например, одна из самых запутанных областей — параметризованный код. Хорошая отправная точка для изучения этого вопроса — документация по параметризованным файлам mypy. Еще одним приятным ресурсом является шпаргалка по аннотациям типов, выпущенная mypy (https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html).

Дэвид Копец

Классические задачи Computer Science на языке Python

Перевела с английского *Е. Полонская*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
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: 09.2019. 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 11.09.19. Формат 70x100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1300. Заказ 7297.

Отпечатано в АО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru
тел: 8(499) 270-73-59

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;*
e-mail: books@piter.com

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

Классические задачи Computer Science на языке Python

Дэвид Копец

Многие задачи в области Computer Science, которые на первый взгляд кажутся новыми или уникальными, на самом деле уходят корнями в классические алгоритмы, методы кодирования и принципы разработки. И устоявшиеся техники по-прежнему остаются лучшим способом решения таких задач! Научитесь писать оптимальный код для веб-разработки, обработки данных, машинного обучения и других актуальных сфер применения Python.

Книга даст вам возможность глубже освоить язык Python, проверить себя на испытанных временем задачах, упражнениях и алгоритмах. Вам предстоит решать десятки заданий по программированию — от самых простых (например, найти элементы списка с помощью двоичной сортировки) до сложных (выполнить кластеризацию данных методом k-средних). Прорабатывая примеры, посвященные поиску, кластеризации, графам и пр., вы вспомните то, о чем успели позабыть, и овладеете классическими приемами решения повседневных задач.

В этой книге

- Алгоритмы поиска.
- Обобщенные технологии для графов.
- Нейронные сети.
- Генетические алгоритмы.
- Состязательный поиск.
- Использование аннотаций типов в описанных задачах.

Для программистов среднего уровня Python.

Дэвид Копец — доцент кафедры компьютерных наук и инноваций колледжа Шамплейн в Берлингтоне, штат Вермонт. Является автором книг Dart for Absolute Beginners (Apress, 2014) и Classic Computer Science Problems in Swift (Manning, 2018).

«Независимо от того, являетесь вы новичком или профессионалом, в этой книге для каждого найдется свой "вау-эффект"».

— Джеймс Уотсон, Adaptive

«Интересный способ получить опыт решения классических задач Computer Science на современном Python».

— Йенс Кристиан Бредал Мэдсен, IT Relation

«Настоятельно рекомендую эту книгу всем, кто хотел бы углубить свое понимание не только языка Python, но и прикладных задач информатики».

— Доктор Даниэл Кенни-Янг, Университет Миннесоты

«Классические задачи всегда были удивительно интересным способом освоения языка, который, кажется, всегда может предложить что-то новое».

— Сэм Зайдель, RackTop Systems

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг и интернет-магазин

instagram.com/piterbooks
youtube.com/ThePiterBooks
vk.com/piterbooks
facebook.com/piterbooks

ISBN: 978-5-4461-1428-3