Певзнер П.
Компо Ф.

АЛГОРИТМЫ биоинформатики

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Алгоритмы биоинформатики

Перед вами одно из самых популярных за рубежом руководств по биоинформатике. Книга обеспечивает уникальный баланс между практическими задачами современной биологии и фундаментальными алгоритмическими идеями.

Каждая глава начинается с биологического вопроса, например:

- Что такое молекулярные часы ДНК?
- Чем различаются геномы человека и мыши?
- Какое животное заразило нас коронавирусом?
- Как строятся эволюционные деревья?
- Как дрожжи научились делать вино?
- Как обнаруживают локацию болезнетворных мутаций в геноме?
- Почему биологи до сих пор не разработали вакцину от ВИЧ?

А затем неуклонно развивается алгоритмическая сложность, необходимая для ответа на этот вопрос. Сотни упражнений включены непосредственно в текст и помогают разобраться в непростом материале.

Издание предназначено специалистам в области анализа данных, а также будет полезно ученым, инженерам, студентам и аспирантам, работающим на стыке биологии и информатики.

Павел Певзнер — заслуженный профессор компьютерных наук и инженерии Калифорнийского университета в Сан-Диего, где он возглавляет кафедру Рональда Р. Тейлора. В 2006 году Павел был назначен профессором Медицинского института Говарда Хьюза. Является автором и соавтором нескольких книг по биоинформатике.

Филлип Компо — адъюнкт-профессор и заместитель заведующего кафедрой факультета вычислительной биологии Университета Карнеги-Меллона. Руководит программой бакалавриата по вычислительной биологии. Также является соучредителем онлайн-платформы для изучения биоинформатики, которая охватила десятки тысяч учащихся по всему миру.

ISBN 978-5-93700-175-7

Филлип Компо и Павел Певзнер

Алгоритмы биоинформатики

Bioinformatics Algorithms

An Active Learning Approach

PHILLIP COMPEAU AND PAVEL PEVZNER

Алгоритмы биоинформатики

ФИЛЛИП КОМПО И ПАВЕЛ ПЕВЗНЕР

ДМК ПРЕСС
издательство
Москва, 2023

УДК 575.112
ББК 28.071.3
К63

Певзнер П., Компо Ф.
К63 Алгоритмы биоинформатики / пер. с англ. И. Л. Люско. – М.: ДМК Пресс, 2022. – 682 с.: ил.

ISBN 978-5-93700-175-7

Перед вами одно из самых популярных за рубежом руководств по биоинформатике. Книга обеспечивает уникальный баланс между практическими задачами современной биологии и фундаментальными алгоритмическими идеями. Каждая глава начинается с биологического вопроса, а затем неуклонно развивается алгоритмическая сложность, необходимая для ответа на него. Сотни упражнений включены непосредственно в текст и помогают разобраться в непростом материале.

Издание предназначено специалистам в области анализа данных, а также будет полезно ученым, инженерам, студентам и аспирантам, работающим на стыке биологии и информатики.

УДК 575.112
ББК 28.071.3

ISBN 978-5-93700-175-7 (рус.)

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

В каком месте генома начинается репликация ДНК?

Алгоритмическая разминка

Путешествие в тысячу миль...

Репликация генома – одна из важнейших задач, выполняемых в клетке. Прежде чем клетка сможет делиться, она должна сначала реплицировать свой геном, чтобы каждая из двух дочерних клеток наследовала свою собственную копию генома. В 1953 году Джеймс Уотсон и Фрэнсис Крик завершили свою знаменательную статью о двойной спирали ДНК ставшей теперь популярной фразой:

> *От нашего внимания не ускользнуло, что специфическое выстраивание пар азотистых оснований, которое мы постулировали, сразу предполагает возможный механизм копирования генетического материала.*

Они предположили, что две цепи родительской молекулы ДНК раскручиваются во время репликации, и затем каждая родительская цепь действует как матрица для синтеза новой молекулярной цепи. В результате процесс репликации начинается с пары комплементарных цепей ДНК и заканчивается двумя парами комплементарных цепей, как показано на рис. 1.1.

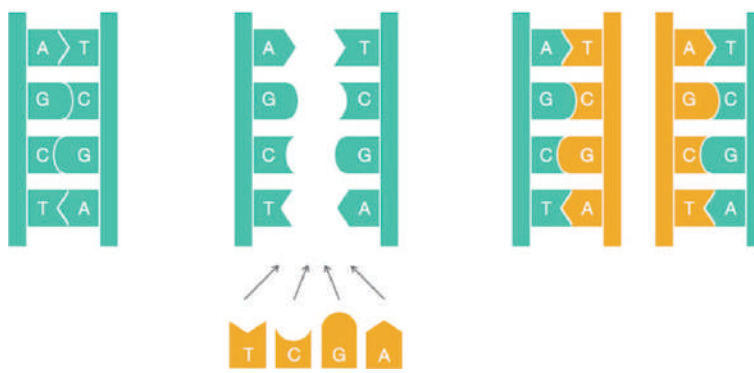


Рис. 1.1 Наивный взгляд на репликацию ДНК. Нуклеотиды аденин (A) и тимин (T) комплементарны друг другу, как и цитозин (C) с гуанином (G). Комплементарные нуклеотиды связываются друг с другом в ДНК

Хотя на рис. 1.1 репликация ДНК представлена на простом уровне, детали репликации оказались гораздо более сложными, чем предполагали Уотсон и Крик; как мы увидим далее, для обеспечения репликации ДНК требуется поразительный механизм молекулярной логистики.

На первый взгляд, ученый-компьютерщик может и не подумать, что эти детали имеют какое-либо значение для вычислений. Чтобы алгоритмически имитировать процесс, показанный на рис. 1.1, нам нужно всего лишь взять строку, представляющую геном, и выдать ее копию! И все же, если мы найдем время для обзора лежащего в основе этого биологического процесса, мы будем вознаграждены новыми алгоритмическими идеями, полученными в анализе процесса репликации.

Репликация начинается в области генома, называемой **точкой начала репликации** (обозначается *ori*), и выполняется молекулярными копировальными машинами, называемыми **ДНК-полимеразами**. Обнаружение *ori* представляет собой важную задачу не только для понимания того, как клетки реплицируются, но и для решения различных биомедицинских задач. Например, в некоторых методах генной терапии используются генетически сконструированные мини-геномы, которые называются **вирусными векторами**, потому что они способны проникать через клеточные стенки (прямо как настоящие вирусы). Вирусные векторы, несущие искусственные гены, использовались в сельском хозяйстве для создания морозоустойчивых томатов и кукурузы, устойчивой к пестицидам. В 1990 году генная терапия была впервые успешно проведена на людях, когда она спасла жизнь четырехлетней девочке, страдающей тяжелым комбинированным иммунодефицитом; девочка была настолько уязвима для инфекций, что была вынуждена жить в стерильной среде.

Идея генной терапии состоит в том, чтобы намеренно заразить пациента, у которого отсутствует важный для жизнедеятельности ген, вирусным вектором, содержащим искусственный ген, кодирующий так называемый терапевтический белок. Оказавшись внутри клетки, вектор реплицируется и в конечном итоге производит множество копий терапевтического белка, который, в свою очередь, лечит болезнь пациента. Чтобы гарантировать, что вектор действительно реплицируется внутри клетки, биологи должны знать, где находится точка начала репликации, *ori,* в геноме вектора, и убедиться, что выполняемые ими генетические манипуляции не влияют на него.

В следующей задаче мы предполагаем, что геном имеет одно начало репликации и представлен в виде **цепи ДНК** или цепи нуклеотидов из четырехбуквенного алфавита {A, C, G, T}.

Задача поиска точки начала репликации: *найти ori в геноме*

Input: ДНК-цепь *Genome*.

Output: локация *ori* в *Genome*.

ОСТАНОВИТЕСЬ и задумайтесь. Является ли эта биологическая задача четко сформулированной вычислительной задачей?

Хотя задача поиска точки начала репликации ставит законный биологический вопрос, она не представляет собой четко сформулированную вычислительную задачу. Действительно, биологи могут немедленно запланировать эксперимент по обнаружению *ori*: например, они могут удалять различные короткие сегменты генома, пытаясь найти сегмент, удаление которого останавливает репликацию. Но специалисты по информатике, с другой стороны,

покачали бы головами и потребовали бы больше информации, прежде чем они смогли бы даже начать думать о задаче.

Почему биологов должно волновать, что думают ученые-компьютерщики? Вычислительные методы в настоящее время являются единственным реальным способом ответить на многие вопросы современной биологии. Во-первых, эти методы намного быстрее, чем экспериментальные методы; во-вторых, результаты многих экспериментов не могут быть интерпретированы без вычислительного анализа. В частности, существующие экспериментальные методы определения локации *ori* требуют много времени. В результате *ori* был экспериментально обнаружен только у нескольких видов. Таким образом, мы хотели бы разработать вычислительный метод поиска *ori*, чтобы биологи могли тратить свое время и деньги на другие задачи.

Скрытые сообщения в точке начала репликации

DnaA-боксы

В оставшейся части этой главы мы сосредоточимся на относительно простом случае обнаружения *ori* в бактериальных геномах, большинство из которых состоит из одной кольцевой хромосомы. Исследования показали, что область бактериального генома, кодирующая *ori*, обычно имеет длину в несколько сотен нуклеотидов. Наш план состоит в том, чтобы начать с бактерии, у которой известны ее *ori*, а затем определить, что делает этот участок генома особенным, чтобы разработать вычислительный метод для нахождения *ori* у других бактерий. Наш пример – *Vibrio cholerae*, бактерия, вызывающая холеру; вот последовательность нуклеотидов, расположенная в ее *ori*:

```
Atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttat
ccacaacctgagtggatgacatcaagataggtcgttgtatctccttcctctcg
tactctcatgaccacggaaagatgatcaagagaggatgatttcttggccatat
cgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgccatatt
gcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgt
tctgtttatcttgttttgactgagacttgttaggatagacggtttttcatcac
tgactagccaaagccttactctgcctgacatcgaccgtaaattgataatgaat
ttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaagatct
tcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcat
gtttccttaaccctctatttttacggaagaatgatcaagctgctgctcttga
tcatcgtttc
```

☞ **Загрузить данные 1.1**

Откуда бактериальная клетка знает, что нужно начинать репликацию именно в этом коротком участке гораздо более длинной хромосомы *Vibrio cholerae*,

состоящей из 1 108 250 нуклеотидов? Должно быть какое-то скрытое сообщение в этой области, приказывающее клетке начать репликацию именно здесь. Действительно, мы знаем, что инициация репликации опосредована ***DnaA***, белком, который связывается с коротким сегментом внутри *ori*, известным как ***DnaA*-бокс**. Вы можете думать о *DnaA*-боксе как о сообщении в цепи ДНК, сообщающем *DnaA* белку: «Присоединяйся сюда!» Вопрос в том, как найти это скрытое сообщение, не зная заранее, как оно выглядит, – сможете ли вы его найти? Другими словами, можете ли вы найти что-то, чем выделяется *ori*? Это рассуждение ставит следующую задачу.

Задача поиска скрытого сообщения: *найти скрытое сообщение в точке начала репликации.*

Input: строка *Text* (представляющая начало репликации генома).

Output: скрытое сообщение в *Text*.

ОСТАНОВИТЕСЬ и задумайтесь. Представляет ли эта задача четко сформулированную вычислительную задачу?

Скрытые сообщения в «Золотом жуке»

Хотя задача скрытого сообщения ставит законный интуитивный вопрос, она все еще не имеет абсолютно никакого смысла для ученого-компьютерщика, поскольку понятие скрытого сообщения точно не определено. Область *ori* холерного вибриона в настоящее время так же загадочна, как пергамент, обнаруженный Уильямом Леграном в рассказе Эдгара Аллана По «Золотой жук». На пергаменте было написано следующее:

```
53++!305))6*;4826)4+.)4+);806*;48!8'60))85;1+(;:+*8!83(88)5*!;46(;88*96*?;
8)*+(;485);5*!2:*+(;4956*2(5*4)8'8*;4069285);)6!8)4++;1(+9;48081;8:8+1;48!
85:4)485!528806*81(+9;48;(88;4(+?34;48)4+;161;:188;+?;
```

Увидев пергамент, рассказчик замечает: «Если бы все драгоценности Голконды ждали меня после решения этой загадки, я совершенно уверен, что не смог бы их заработать». Легран возражает: «Вполне можно сомневаться, способна ли человеческая изобретательность построить такого рода загадку, которую человеческая находчивость при правильном применении не могла бы разрешить». Он обращает внимание, что три последовательных символа «;48» появляются на пергаменте с удивительной частотой:

```
53++!305))6;4826)4+.)4+);806*;48!8'60))85;1+(;:+*8!83(88)5*!;46(;88*96*?;8
)*+(;485);5*!2:*+(;4956*2(5*4)8'8*;4069285);)6!8)4++;1(+9;48081;8:8+1;48!8
5;4)485!528806*81(+9 ;48;(88;4(+?34;48)4+;161;:188;+?;
```

Легран ранее уже сделал вывод, что пираты говорят по-английски; поэтому он предположил, что высокая частота «;48» подразумевает, что она кодирует наиболее часто встречающееся английское слово – артикль «THE». Заменяя каждый символ, Легран получил немного более простой текст для расшифровки, который в конечном итоге привел его к зарытому сокровищу. Можете расшифровать и это сообщение?

```
53++!305))6*THE26)H+.)H+)TE06*THE!E‘60))E5T1+(T:+*E!E3(EE)5*!T
H6(TEE*96*?TE)*+(THE5)T5*!2:*+(TH956*2(5*H)E’E*TH0692E5)T)6!E)
H++T1(+9THE0E1TE:E+1THE!E5TH)HE5!52EE06*E1(+9THET(EETH(+?3HT
HE)H+T161T:1EET+?T
```

Подсчет слов

Опираясь на предположение, что ДНК – это язык сам по себе, давайте воспользуемся методом Леграна и посмотрим, сможем ли мы найти какие-нибудь неожиданно часто встречающиеся «слова» в *ori* холерного вибриона (*Vibrio cholerae*). Имеет смысл искать часто встречающиеся слова в *ori*, потому что для различных биологических процессов определенные цепи нуклеотидов удивительно часто появляются в небольших областях генома. Это связано с тем, что некоторые белки могут связываться с ДНК только в том случае, если присутствует определенная последовательность нуклеотидов, и если существует больше вхождений этой последовательности, то более вероятно, что связывание произойдет успешно. (Также менее вероятно, что мутация нарушит процесс связывания.)

Например, **ACTAT** – удивительно частая подстрока

ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT.

Мы используем термин «***k*-меры**» для обозначения строки длины *k* и определяем *Count*(*Text, Pattern*) как количество раз, когда *k*-мер *Pattern* появляется как подстрока строки *Text*. Следуя приведенному выше примеру,

Count(ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT, **ACTAT**) = 3.

Обратите внимание, что *Count*(CG**ATATA**TCC**ATA**G, **ATA**) равно 3 (а не 2), так как мы должны учитывать перекрывающиеся вхождения *Pattern* в *Text*.

Наш план вычисления *Count*(*Text, Pattern*) состоит в том, чтобы «сдвигать окно» вдоль строки *Text*, проверяя, соответствует ли каждая подстрока *k*-мера текста *Text* образцу *Pattern*. Поэтому мы будем ссылаться на *k*-мер, начинающийся с позиции *i* текста, как на *Text(i, k)*. В этой книге мы часто будем использовать **ноль-индексацию** (нумерацию на основе нуля), что означает, что мы начинаем отсчет с 0, а не с 1. В этом случае *Text* начинается с позиции 0 и заканчивается в позиции |*Text*| – 1 (|*Text*| обозначает количество символов в тексте). Например, если *Text* = GACCATACTG, то *Text*(4, 3) = ATA. Обратите внимание, что последний *k*-мер *Text* начинается с позиции |*Text*| – *k*, например последний

3-мер GACCATACTG начинается с позиции 10 – 3 = 7. Это рассуждение приводит к следующему псевдокоду для вычисления *Count*(*Text, Pattern*).

```
PatternCount(Text, Pattern)
    count ← 0
    for i ← 0 до |Text| – |Pattern|
        if Text(i, |Pattern|) = Pattern
            count ← count + 1
    return count
```

Важное примечание. В этом тексте мы используем термин **псевдокод** для описания алгоритмов, с которыми сталкиваемся при решении задач современной биологии. Псевдокод – это универсальный метод описания алгоритмов, более точный, чем человеческий язык, но не требующий от нас увязнуть в синтаксисе конкретного языка программирования.

Задача поиска часто встречающихся слов

Мы говорим, что *Pattern* является наиболее частым *k*-мером в *Text*, если он максимизирует *Count*(*Text, Pattern*) среди всех *k*-меров. Вы можете видеть, что **ACTAT** является наиболее частым 5-мером для *Text* = ACA**ACTAT**GCAT**ACTAT**CGGGA**ACTAT**CCT, а **ATA** – наиболее частым 3-мером для *Text* = CG**ATATA**TCC**ATA**G.

ОСТАНОВИТЕСЬ и задумайтесь. Может ли строка иметь несколько наиболее часто встречающихся *k*-меров?

Теперь у нас есть строго определенная вычислительная задача.

Задача поиска часто встречающихся слов: *найдите наиболее часто встречающиеся k-меры в строке.*

Input: строка *Text* и целое число *k*.

Output: все наиболее часто встречающиеся *k*-меры в тексте.

Прямой алгоритм нахождения наиболее часто встречающихся *k*-меров в строке *Text* проверяет все *k*-меры, встречающиеся в этой строке (имеется |*Text*| – *k* + 1 таких *k*-меров), а затем вычисляет, сколько раз каждый *k*-мер появляется в *Text*. Для реализации этого алгоритма, называемого **FrequentWords**, нам потребуется сгенерировать массив *Count*, где *Count*(*i*) содержит *Count*(*Text*, *Pattern*) для *Pattern* = *Text*(*i, k*) (рис. 1.2).

Text	**A**	**C**	**T**	G	**A**	**C**	**T**	C	C	C	A	C	C	C	C
Count	**2**	1	1	1	**2**	1	1	3	1	1	1	3	3		

Рис. 1.2 Массив *Count* для *Text* = ACTGACTCCCACCCC и *k* = 3. Например, *Count*(0) = *Count*(4) = 2, потому что ACT (выделен жирным шрифтом) дважды встречается в строке *Text* в позициях 0 и 4

```
FrequentWords(Text, k)
    FrequentPatterns ← пустой набор
    for i ← 0 до |Text| − k
        Pattern ← k-мер Text(i, k)
        Count(i) ← PatternCount(Text, Pattern)
    maxCount ← максимальная величина массива Count
    for i ← 0 до |Text| − k
        if Count(i) = maxCount
            add Text(i, k) до FrequentPatterns
    удалить дубликаты из FrequentPatterns
    return FrequentPatterns
```

ОСТАНОВИТЕСЬ и задумайтесь. Насколько быстро работает **FrequentWords**?

Хотя **FrequentWords** находит наиболее часто встречающиеся *k*-меры, он не очень эффективен. Каждый вызов **PatternCount**(*Text, Pattern*) проверяет, появляется ли *k*-мер *Pattern* в позиции 0 *Text*, позиции 1 *Text* и т. д. Поскольку каждый *k*-мер требует $|Text| - k + 1$ таких проверок, каждая из которых требует *k* сравнений, общее количество шагов **PatternCount**(*Text, Pattern*) равно $(|Text| - k + 1) \cdot k$. Более того, **FrequentWords** должен вызывать **PatternCount** $|Text| - k + 1$ раз (по одному разу для каждого *k*-мера *Text*), так что общее количество шагов равно $(|Text| - k + 1) \cdot (|Text| - k + 1) \cdot k$. Чтобы упростить дело, ученые-компьютерщики часто говорят, что время выполнения **FrequentWords** имеет верхнюю границу $|Text|^2 \cdot k$ шагов, и ссылаются на сложность этого алгоритма как $O(|Text|^2 \cdot k)$ (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Big-*O* («*O* большое»)**).

Если $|Text|$ и *k* малы, как в случае поиска *DnaA*-боксов в типичных бактериальных *ori*, тогда алгоритм со временем работы $O|Текст|^2 \cdot k$ вполне приемлем. Но как только мы найдем новое биологическое приложение, требующее от нас решения задачи часто используемых слов для очень длинного текста, мы быстро столкнемся с проблемами.

Более быстрый подход к задаче частых слов

Если бы вам нужно было решить задачу о часто встречающихся словах вручную для небольшого примера, вы, вероятно, сформировали бы таблицу, подобную

данной таблице ниже, для текста = «ACGTTTCACGTTTTACGG» и *k*, равного 3. Будем сдвигать окно длиной *k*. *Text*, и если текущая *k*-мер подстрока *text* не встречается в таблице, то вы должны создать для нее новую запись. В противном случае вы бы добавили 1 к записи, соответствующей текущей k-мерной подстроке Text. Мы называем эту таблицу **таблицей частот** для *Text* и *k*.

ACG	CGT	GTT	TTT	TTC	TCA	CAC	TTA	TAC	CGG
3	2	2	3	1	1	1	1	1	1

Таблица, соответствующая подсчету количества вхождений каждого 3-мера в Text = «ACGTTTCACGTTTTACGG»

В предыдущем алгоритме **FrequentWords** мы также делаем один проход по тексту, но каждый раз, сталкиваясь в окне с *k*-мером, мы вызываем подпрограмму **PatternCount**, которая требует собственного прохода по всей длине *Text*. Но когда мы строим таблицу частот, то делаем один проход по тексту, и каждый раз, когда мы сталкиваемся с *k*-мером, просто добавляем 1 к счету *k*-мера.

Мы знаем, что массив длины *n* представляет собой упорядоченную таблицу значений, где мы обращаемся к значениям, используя целочисленные индексы от 0 до *n* – 1. Таблица частот представляет собой обобщенную версию массива, называемого **картой (map)** или **словарем (dictionary)**, в котором индексы могут быть произвольными значениями (в данном случае это строки). Точнее, индексы карты называются **ключами (keys)**.

Имея карту *dict*, мы можем получить доступ к значению, связанному с ключом *key*, используя обозначение *dict*[*key*]. В случае таблицы частот с именем *freq* мы можем получить доступ к значению, связанному с некоей ключевой строкой *pattern*, используя обозначение *freq*[*pattern*]. Следующая функция псевдокода принимает строку *text* и целое число *k* в качестве входных данных и выдает таблицу частот в виде сопоставления ключей строки с целочисленными значениями.

```
FrequencyTable(Text, k)
    freqMap ← пустой map
    n ← |Text|
    for i ← 0 to n − k
        Pattern ← Text(i, k)
        if freqMap[Pattern] не существует
            freqMap[Pattern] ← 1
        else
            freqMap[pattern] ← freqMap[pattern] + 1
    return freqMap
```

После того как мы построили таблицу частот для данного *Text* и *k*, можно найти все часто встречающиеся *k*-меры, если определим максимальное значение в таблице, а затем идентифицируем ключи таблицы частот, достигающие этого значения, добавляя каждый из них, который мы нашли в растущем списке. Те-

перь мы готовы написать функцию **BetterFrequentWords** для решения задачи часто встречающихся слов. Эта функция основана на функции **MaxMap**, которая принимает в качестве входных данных карту строк в целых числах и выдает максимальное значение этой карты в качестве выходных данных.

```
BetterFrequentWords(Text, k)
    FrequentPatterns ← массив строк длины 0
    freqMap ← FrequencyTable(Text, k)
    max ← MaxMap(freqMap)
    for всех строк Pattern в freqMap
        if freqMap[pattern] = max
            добавить Pattern к FrequentPatterns
    return FrequentPatterns
```

Упражнение. Напишите функцию **MaxMap**. Убедитесь, что ваша функция будет работать, даже если все значения отрицательные.

Часто используемые слова в Vibrio cholerae

На рис. 1.3 показаны наиболее часто встречающиеся *k*-меры в области *ori* бактерии *Vibrio cholerae*.

k	3	4	5	6	7	8	9
count	25	12	8	8	5	4	3
***k*-меры**	tga	atga	gatca tgatc	tgatca	atgatca	atgatcaa	atgatcaag cttgatcat tcttgatca ctcttgatc

Рис. 1.3 Наиболее часто встречающиеся *k*-меры в области *ori Vibrio cholerae* для *k* от 3 до 9, а также количество раз, когда каждый *k*-мер встречается

```
atcaatgatcaacgtaagcttctaagcatgatcaaggtgctcacacagtttat
ccacaacctgagtggatgacatcaagataggtcgttgtatctccttcctctcg
tactctcatgaccacggaaagatgatcaagagaggatgatttcttggccatat
cgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgccatatt
gcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgt
tctgtttatcttgttttgactgagacttgttaggatagacggtttttcatcac
tgactagccaaagccttactctgcctgacatcgaccgtaaattgataatgaat
ttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaagatct
tcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcat
gtttccttaaccctctatttttacggaagaatgatcaagctgctgctcttga
tcatcgtttc
```

ОСТАНОВИТЕСЬ и задумайтесь. Кажутся ли какие-либо цифры на рис. 1.3 неожиданно большими?

Например, 9-мер **ATGATCAAG** трижды появляется в области *ori* генома *Vibrio cholerae* – разве это удивительно?

```
atcaatgatcaacgtaagcttctaagcATGATCAAGgtgctcacacagtttat
ccacaacctgagtggatgacatcaagataggtcgttgtatctccttcctctcg
tactctcatgaccacggaaagATGATCAAGagaggatgatttcttggccatat
cgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgccatatt
gcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgt
tctgtttatcttgttttgactgagacttgttaggatagacggtttttcatcac
tgactagccaaagccttactctgcctgacatcgaccgtaaattgataatgaat
ttacatgcttccgcgacgatttacctcttgatcatcgatccgattgaagatct
tcaattgttaattctcttgcctcgactcatagccatgatgagctcttgatcat
gtttccttaaccctctatttttacggaagaATGATCAAGctgctgctcttga
tcatcgtttc
```

Мы выделяем наиболее часто встречающийся 9-мер вместо того, чтобы использовать какое-либо другое значение *k*, потому что эксперименты показали, что бактериальные *DnaA*-боксы обычно имеют длину девять нуклеотидов.

Вероятность того, что существует 9-мер, появляющийся три или более раз в случайно сгенерированной цепи ДНК длиной 500, составляет приблизительно 1/1300 (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Вероятность паттернов в строке**).

На самом деле в этой области есть четыре разных 9-мера, повторяющихся три или более раз: **ATGATCAAG**, CTTGATCAT, TCTTGATCA и CTCTTGATC. Низкая вероятность обнаружения хотя бы одного повторяющегося 9-мера в области *ori* холерного вибриона приводит нас к рабочей гипотезе о том, что один из этих четырех 9-меров может представлять собой вероятный *DnaA*-бокс, который, появляясь несколько раз в короткой области, запускает процесс репликации. Но какой?

ОСТАНОВИТЕСЬ и задумайтесь. Является ли какой-либо из четырех наиболее часто встречающихся 9-меров в *ori* холерных вибрионов «более примечательным», чем другие?

Некоторые скрытые сообщения более примечательны, чем другие

Напомним, что нуклеотиды **A** и **T** комплементарны друг другу, как и **C** с **G**. Имея одну цепь ДНК и запас «свободно плавающих» нуклеотидов, как показано на

рис. 1.1, можно представить себе синтез **комплементарной цепи** на **матричной цепи** . Эта модель репликации была подтверждена Мезельсоном и Шталем в 1958 году (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Самый красивый эксперимент в биологии**). На рис. 1.4 показаны матричная цепь **TCAGCGTATCA** и комплементарная ей цепь **ACTATGCGACT**.

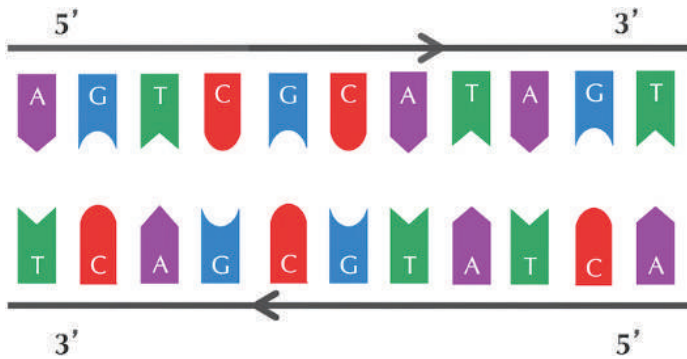


Рис. 1.4 Комплементарные цепи идут в противоположных направлениях. Каждая цепь читается в направлении 5' → 3'

В этот момент вы можете подумать, что допустили ошибку, поскольку комплементарная цепь на рис. 1.4 считывает **TCAGCGTATCA** слева направо, а не **ACTATGCGACT**. Но нет: у каждой цепи ДНК есть направление, а комплементарная цепь идет в направлении, противоположном направлению цепи матрицы, как показано стрелками на рис. 1.4. Каждая цепь читается в направлении 5' → 3' (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Направленность цепей ДНК**, чтобы узнать, почему биологи обозначают начало и конец цепи ДНК как 5' и 3').

Возьмем нуклеотид p и обозначим его комплементарный нуклеотид как p^*. **Обратным комплементом** строки $Pattern = p_1 \ldots p_n$ является строка $Pattern_{rc} = p_n^* \ldots p_1^*$, образованная путем взятия комплемента каждого нуклеотида в $Pattern$ и последующего обращения полученной строки. В этой главе нам понадобится решение следующей задачи.

Задача поиска обратного комплемента: *найдите комплементарное дополнение данной цепи ДНК.*

Input: строка ДНК *Pattern*.

Output: $Pattern_{rc}$, реверсный комплемент *Pattern*.

ОСТАНОВИТЕСЬ и задумайтесь. Еще раз посмотрите на четыре наиболее часто встречающихся 9-мера в области *ori* холерного вибриона с рис. 1.3. Заметили ли вы что-нибудь удивительное теперь?

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagttta
tccacaacctgagtggatgacatcaagataggtcgttgtatctccttcctctcg
tactctcatgaccacggaaag**ATGATCAAG**agaggatgatttcttggccata
tcgcaatgaatacttgtgacttgtgcttccaattgacatcttcagcgccatatt
gcgctggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgtt
ctgtttatcttgttttgactgagacttgttaggatagacggtttttcatcactg
actagccaaagccttactctgcctgacatcgaccgtaaattgataatgaattta
catgcttccgcgacgatttac**CTCTTGATCAT**cgatccgattgaagatcttc
aattgttaattctcttgcctcgactcatagccatgatgag**CTCTTGATCAT**g
tttccttaaccctctatttttacggaaga**ATGATCAAG**ctgctg**CTCTTG**
ATCATcgtttc

Интересно, что среди четырех наиболее часто встречающихся 9-меров в области *ori* холерного вибриона **ATGATCAAG** и **CTTGATCAT** реверсно комплементарны друг другу, что приводит к следующим шести вхождениям этих строк.

atcaatgatcaacgtaagcttctaagc**ATGATCAAG**gtgctcacacagtttatc
cacaacctgagtggatgacatcaagataggtcgttgtatctccttcctctcgta
ctctcatgaccacggaaag**ATGATCAAG**agaggatgatttcttggccatatcgc
aatgaatacttgtgacttgtgcttccaattgacatcttcagcgccatattgcgc
tggccaaggtgacggagcgggattacgaaagcatgatcatggctgttgttctgt
ttatcttgttttgactgagacttgttaggatagacggtttttcatcactgacta
gccaaagccttactctgcctgacatcgaccgtaaattgataatgaatttacatg
cttccgcgacgatttacct**CTTGATCAT**cgatccgattgaagatcttcaattgt
taattctcttgcctcgactcatagccatgatgagct**CTTGATCAT**gtttcctta
accctctatttttacggaaga**ATGATCAAG**ctgctgct**CTTGATCAT**cgtttc

Обнаружение 9-мера, который появляется шесть раз (либо сам по себе, либо как его реверсный комплемент) в цепи ДНК длиной 500 нуклеотидов, гораздо более удивительно, чем обнаружение 9-мера, который появляется три раза (как сам по себе). Это наблюдение приводит нас к рабочей гипотезе, что **ATGATCAAG** и его реверсный комплемент **CTTGATCAT** действительно представляют собой *DnaA*-боксы в *Vibrio cholerae*.

Этот вычислительный вывод имеет биологический смысл, потому что белку *DnaA*, который связывается с *DnaA*-боксом и инициирует репликацию, все равно, с какой из двух цепей он связывается. Таким образом, для наших целей и **ATGATCAAG**, и **CTTGATCAT** представляют собой *DnaA*-боксы.

Однако, прежде чем сделать вывод, что мы нашли *DnaA*-бокс холерного вибриона, внимательный биоинформатик должен проверить, есть ли другие короткие области в геноме холерного вибриона, демонстрирующие множественные точки входа **ATGATCAAG** (или **CTTGATCAT**). В конце концов, возможно, эти цепи повторяются во всем геноме *Vibrio cholerae*, а не только в области *ori*. Для этого нам необходимо решить следующую задачу.

Задача поиска вхождений *Pattern* в строку: *найти все точки вхождения Pattern в строку.*

Input: строки *Pattern* и *Genome*.

Output: все стартовые позиции в *Genome*, где *Pattern* появляется как подстрока.

Упражнение. Выдайте список начальных позиций, разделенных пробелами (в порядке возрастания), где CTTGATCAT появляется как подстрока в геноме холерного вибриона.

☞ **Загрузить данные 1.2**

Примечания.

1. Перерывы на упражнения – это возможность расширить свои знания по теме. Они необязательны, поэтому вы можете свободно продолжать чтение.
2. Мы покажем все стартовые позиции **ATGATCAAG** в этом геноме далее.

После решения задачи сопоставления с паттерном мы обнаруживаем, что **ATGATCAAG** встречается 17 раз в следующих позициях генома холерного вибриона:

116556, 149355, **151913**, **152013**, **152394**, 186189, 194276, 200076, 224527, 307692, 479770, 610980, 653338, 679985, 768828, 878903, 985368

За исключением трех точек входа **ATGATCAAG** в *ori* в начальных положениях **151913**, **152013** и **152394**, никакие другие экземпляры **ATGATCAAG** не образуют **сгустков (clumps)**, т. е. не появляются близко друг к другу в небольшой области генома (Эти сгустки (clumps) не надо путать с DNA clamps, так называемых белков скользящего зажима. – *Прим. ред.*). Вы можете убедиться, что такой же вывод делается и при поиске **CTTGATCAT**. Теперь у нас есть убедительные статистические доказательства того, что **ATGATCAAG**/**CTTGATCAT** может представлять собой скрытое сообщение для *DnaA* о старте репликации.

ОСТАНОВИТЕСЬ и задумайтесь. Можем ли мы заключить, что **ATGATCAAG**/**CTTGATCAT** также представляет собой *DnaA*-бокс в геномах других бактерий?

Взрыв скрытых сообщений

Поиск скрытых сообщений в нескольких геномах

Мы не должны делать поспешных выводов о том, что **ATGATCAAG**/**CTTGATCAT** является скрытым сообщением для всех бактериальных геномов, не проверив предварительно, появляется ли оно вообще в известных областях *ori* других бактерий. В конце концов, возможно, эффект скопления **ATGATCAAG**/**CTTGATCAT** в области *ori* холерного вибриона является просто статистической случайностью, не имеющей ничего общего с репликацией. Или, возможно, разные бактерии имеют разные *DnaA*-боксы…

Давайте проверим предполагаемую область ori *Thermotoga petrophila*, бактерии, которая процветает в чрезвычайно жаркой среде; ее название происходит от места ее обнаружения в воде под нефтяными резервуарами, где температура может превышать 80 °C.

```
aactctatacctccttttttgtcgaatttgtgtgatttatagagaaaatct
tattaactgaaactaaaatggtaggtttggtggtaggttttgtgtacat
tttgtagtatctgatttttaattacataccgtatattgtattaaattga
cgaacaattgcatggaattgaatatatgcaaaacaaacctaccaccaaac
tctgtattgaccattttaggacaacttcagggtggtaggtttctgaagct
ctcatcaatagactattttagtctttacaaacaatattaccgttcagatt
caagattctacaacgctgttttaatgggcgttgcagaaaacttaccaccta
aaatccagtatccaagccgatttcagagaaacctaccacttacctaccact
tacctaccacccgggtggtaagttgcagacattattaaaaacctcatcag-
aagcttgttcaaaaatttcaatactcgaaa  cctaccacctgcgtcccctatt
atttactactactaataatagcagtataattgatctga
```

Эта область не содержит ни одного вхождения **ATGATCAAG** или **CTTGATCAT**! Таким образом, разные бактерии могут использовать разные *DnaA*-боксы в качестве скрытых сообщений для белка *DnaA*.

Применение задачи частых слов к описанной выше области *ori* показывает, что следующие шесть 9-меров появляются в этой области три или более раз:

AACCTACCA AAACCTACC ACCTACCAC
CCTACCACC GGTAGGTTT TGGTAGGTT

Должно происходить что-то необычное, потому что крайне маловероятно, что шесть разных 9-меров будут так часто встречаться в коротком участке случайной строки. Мы немного схитрим и проконсультируемся с **Ori-Finder**, патентованным программным инструментом для поиска точек начала репликации в цепях ДНК. Это программное обеспечение выбирает **CCTACCACC** (вместе с его обратным комплементом **GGTGGTAGG**) в качестве рабочей гипотезы для *DnaA*-бокса у *Thermotoga petrophila*. Вместе эти два комплементарных 9-мера появляются пять раз в точке начала репликации:

```
aactaaaatggtaggtttGGTGGTAGGttttgtgtacattttgtagtatc
tgatttttaattacataccgtatattgtattaaattgacgaacaattgc
atggaattgaatatatgcaaaacaaaCCTACCACCaaactctgtattga
ccattttaggacaacttcagGGTGGTAGGtttctgaagctctcatcaata
gactattttagtctttacaaacaatattaccgttcagattcaagattcta
caacgctgttttaatgggcgttgcagaaaacttaccacctaaaatccagt
atccaagccgatttcagagaaacctaccacttacctaccactaCCTACCA
CCcgggtggtaagttgcagacattattaaaaacctcatcagaagcttgttc
aaaaatttcaatactcgaaaCCTACCACCtgcgtcccctattatttacta-
ctactaataatagcagtataattgatctga
```

Задача поиска сгустков

Теперь представьте, что вы пытаетесь найти *ori* в недавно секвенированном бактериальном геноме. Поиск сгустков ATGATCAAG/CTTGATCAT или CCTACCACC/GGTGGTAGG вряд ли поможет, так как этот новый геном может использовать совершенно другое скрытое сообщение. Прежде чем мы потеряем всякую надежду, давайте изменим наше направление поиска: вместо того, чтобы искать сгустки определенного *k*-мера, давайте попытаемся найти *каждый k*-мер, который образует сгусток в геноме. Будем надеяться, что расположение этих сгустков прольет свет на местонахождение *ori*.

Наш план состоит в том, чтобы перемещать окно фиксированной длины *L* вдоль генома в поисках области, где *k*-мер появляется несколько раз подряд. Значение параметра *L* = 500 отражает типичную длину *ori* в бактериальных геномах.

Мы определили *k*-мер как сгусток (clump), если он появляется много раз в пределах короткого интервала генома. Более формально для заданных чисел *L* и *t* *k*-мер *Pattern* образует **(*L, t*)-clump** внутри (более длинной) строки *Genome*, если существует интервал *Genome* длины *L*, в котором этот *k*-мер появляется по крайней мере *t* раз. (Это определение предполагает, что *k*-мер полностью укладывается в интервал.) Например, **TGCA** образует (25, 3)-clump в следующем *Genome*:

```
gatcagcataagggtccCTGCAATGCATGACAAGCCTGCAGTtgttttac
```

Из наших предыдущих примеров областей *ori* ATGATCAAG образует (500, 3)-clump в геноме *Vibrio cholerae*, а CCTACCACC образует (500, 3)-clump в геноме *Thermotoga petrophila*. Теперь мы готовы сформулировать следующую задачу.

Задача поиска сгустков: *найдите Patterns, образующие сгустки в строке.*

Input: строка *Genome* и целые числа *k*, *L* и *t*.

Output: все различные *k*-меры, образующие (*L, t*)-clump в *Genome*.

ЗАРЯДНАЯ СТАНЦИЯ: Решение задачи поиска сгустков. Вы можете решить задачу поиска сгустков, просто применив свой алгоритм для задачи частых слов к каждому окну длиной L в *Genome*. Однако, если ваш алгоритм решения задачи часто встречающихся слов не очень эффективен, такой подход может оказаться нецелесообразным. Например, вспомните, что **FrequentWords** имеет время выполнения $O(L^2 \cdot k)$. Применение этого алгоритма к каждому окну длины L в *Genome* приведет к продолжительности расчета $O(L^2 \cdot k \cdot |Genome|)$. Более того, даже если мы используем более быстрый алгоритм для поиска часто встречающихся слов, время выполнения останется большим, даже когда мы пытаемся проанализировать небольшой бактериальный геном, не говоря уже о человеческом геноме.

Задача поиска сгустков сложнее, чем та, с которой мы сталкивались до сих пор, и написать функцию, решающую ее с нуля, будет сложно. Однако именно здесь модульность в написании программ так полезна. У нас уже есть функция **FrequencyTable**, которая создаст таблицу частот для данного окна из строки длины L. Если мы применим ее к данному окну, то нам просто нужно проверить, есть ли в таблице какие-либо ключи строк, значения которых по крайней мере равно t. Мы будем добавлять любые такие ключи, которые мы еще не видели в каком-либо другом текстовом окне, к растущему списку строк. В конце концов этот список строк будет содержать (L, t)-сгустки текста. Это обрабатывается следующей функцией **FindClumps**.

```
FindClumps(Text, k, L, t)
    Patterns ← набор строк длиной 0
    n ← |Text|
    for каждого целого i между 0 and n − L
        Window ← Text(i, L)
        freqMap ← FrequencyTable(Window, k)
        for каждого ключа s в freqMap
            if freqMap[s] ⩾ t
                добавить s к Patterns
    удалить дубликаты из Patterns
    return Patterns
```

Давайте поищем сгустки в геноме *Escherichia coli* (*E. coli*), рабочей лошадке бактериальной геномики. Мы находим сотни различных 9-меров, образующих (500, 3)-сгустков в геноме *E. coli*, и совершенно неясно, какой из этих 9-меров может представлять собой *DnaA*-бокс в области *ori* бактерии.

ОСТАНОВИТЕСЬ и задумайтесь. Должны ли мы сдаться? Если нет, что бы вы сделали сейчас?

В этот момент неопытный исследователь мог бы сдаться, так как оказалось, что у нас недостаточно информации, чтобы определить местонахождение *ori* в *E. coli*. Но бесстрашный биоинформатик-ветеран попытался бы узнать больше о деталях репликации в надежде, что они обеспечат новые алгоритмические идеи для поиска *ori*.

Самое простое объяснение процесса репликации ДНК

Теперь мы готовы обсудить процесс репликации более подробно. Как показано на рис. 1.5 (вверху), две комплементарные цепи ДНК, идущие в противоположных направлениях вокруг кольцевой хромосомы, разделяются, начиная с локации *ori*. Когда цепи раскручиваются, они создают две **репликационные вилки**, которые расширяются в обоих направлениях вдоль хромосомы до тех пор, пока цепи полностью не разделятся на **конце репликации** («replication terminus», или «остановка репликации», обозначается *ter*). Конец репликации расположен примерно напротив *ori* в хромосоме.

Важно знать о репликации то, что ДНК-полимераза не ждет, пока две родительские цепи полностью разделятся, прежде чем инициировать репликацию; вместо этого она начинает процесс копирования, *пока* цепи еще распутываются. Таким образом, всего четыре ДНК-полимеразы, каждая из которых отвечает за одну комплементарную часть цепи, могут начинать с *ori* и реплицировать всю хромосому. Чтобы начать репликацию, ДНК-полимеразе нужен **праймер** – короткий комплементарный сегмент (показан красным на рис. 1.5), который связывается с исходной цепью и запускает ДНК-полимеразу. После того как цепи начинают разделяться, каждая из четырех ДНК-полимераз начинает репликацию, добавляя нуклеотиды, начиная с праймера и двигаясь вдоль хромосомы от *ori* к *ter* либо по часовой стрелке, либо против часовой стрелки. Когда все четыре ДНК-полимеразы достигли *ter*, ДНК хромосомы будет полностью реплицирована, что приведет к образованию двух пар комплементарных цепей (рис. 1.5 (внизу)), и клетка готова к делению.

Пока вы читали описание выше, профессора биологии писали петицию, чтобы нас уволили и отправили обратно в «Биологию 101» (Сериал NBC про студентов-биологов. – *Прим. ред.*). И они были бы правы, потому что наше изложение страдает серьезным недостатком; но мы намеренно описали процесс репликации именно таким образом, чтобы вы могли лучше понять то, что мы собираемся до вас донести дальше.

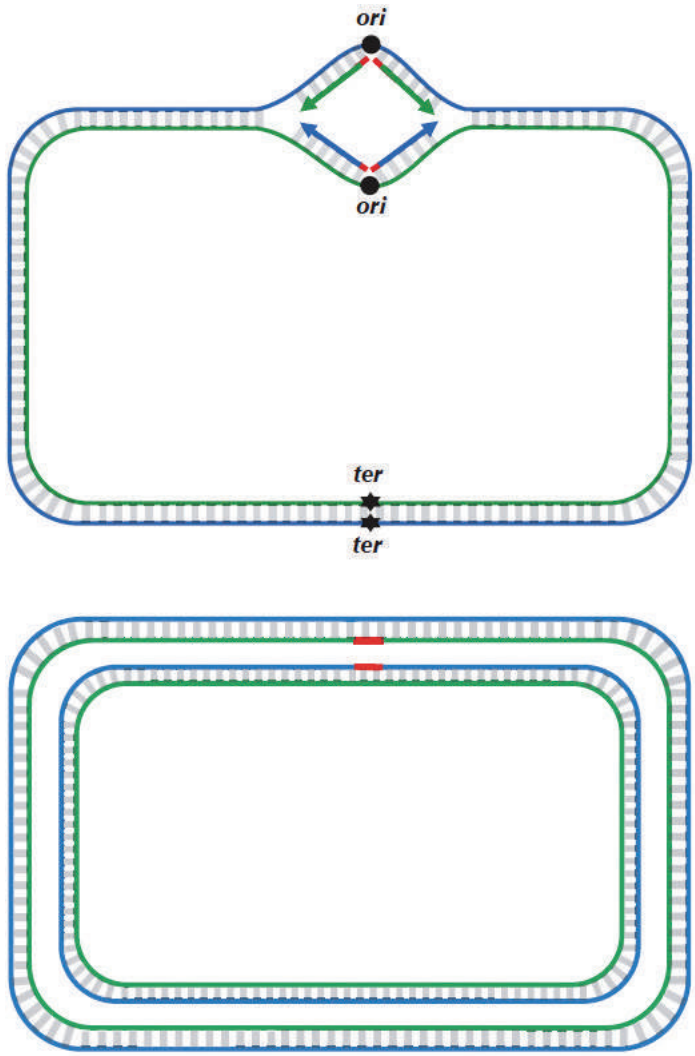


Рис. 1.5 (Вверху) Четыре воображаемых ДНК-полимеразы в процессе репликации хромосомы по мере того, как вилки репликации передвигаются от *ori* до *ter*. Синяя цепь направлена по часовой стрелке, а зеленая – против часовой стрелки. (Внизу) Репликация завершена

Проблема с нашим нынешним описанием заключается в том, что оно предполагает, что ДНК-полимеразы могут копировать ДНК в *любом* направлении вдоль цепи ДНК (т. е. как в направлении 5’ → 3’, так и в направлении 3’ → 5’). Однако природа еще не наделила ДНК-полимеразы такой способностью, поскольку они являются **однонаправленными**, что означает, что они могут передвигаться по

матричной цепи ДНК только в направлении 3’ → 5’. Обратите внимание, что это направление, противоположное направлению 5’ → 3’ цепи ДНК.

ОСТАНОВИТЕСЬ и задумайтесь. Если бы вы использовали однонаправленную ДНК-полимеразу, как бы вы реплицировали ДНК? Сколько ДНК-полимераз потребовалось бы для выполнения такой задачи?

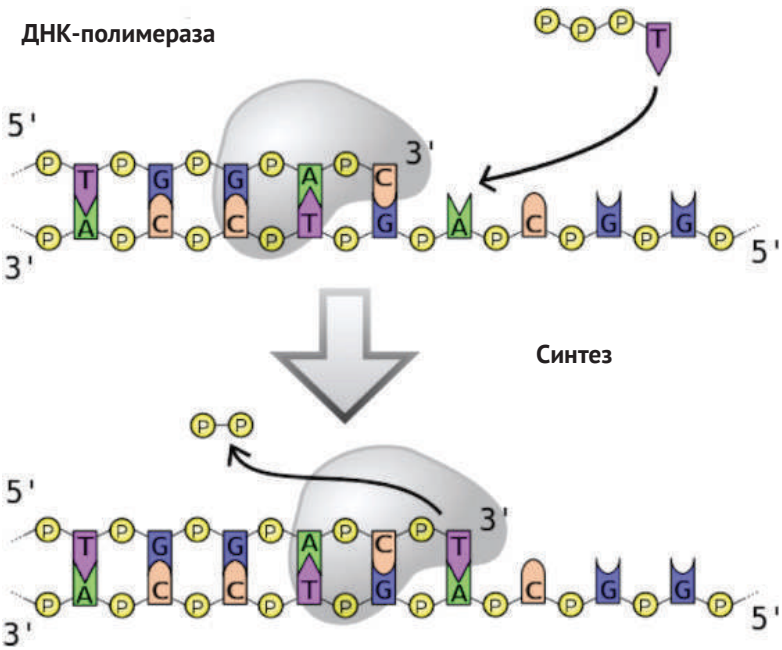


Рис. 1.6 ДНК-полимераза копирует цепь матрицы в направлении 3’ → 5’. Обратите внимание, что она создает дочернюю цепь в направлении 5’ → 3’

Однонаправленность ДНК-полимеразы требует серьезного пересмотра нашей наивной модели репликации. Представьте, что вы решили пройтись по ДНК от *ori* до *ter*. Существуют четыре разные полуцепи родительской ДНК, соединяющие *ori* с *ter*, как показано на рис. 1.7. Две из этих полуцепей проходят от *ori* в направлении 5’ → 3’ и поэтому называются **прямыми полуцепями** (представлены тонкими синими и зелеными линиями на рис. 1.7). Две другие полуцепи проходят от *ori* к *ter* в направлении 3’ → 5’ и поэтому называются **обратными полуцепями** (обозначены толстыми синими и зелеными линиями на рис. 1.7).

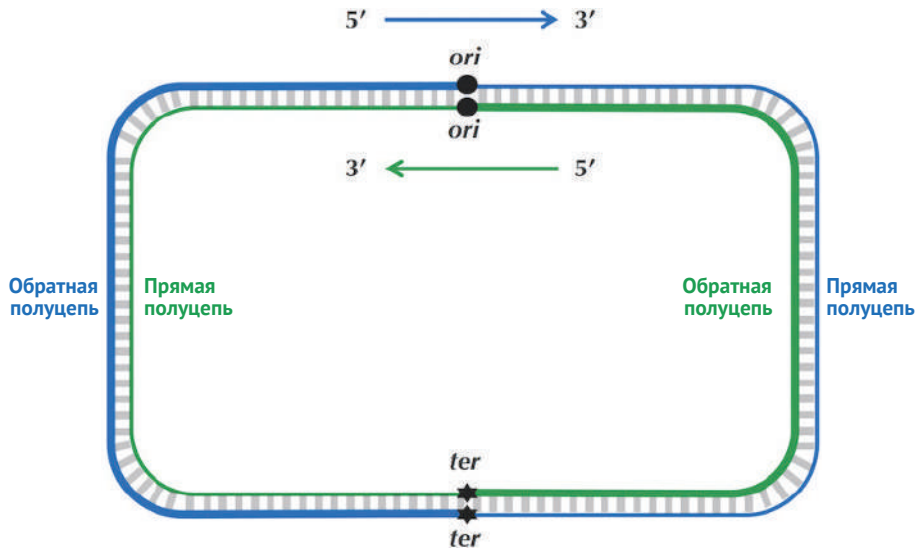


Рис. 1.7 Комплементарные цепи ДНК с прямой и обратной полуцепями, показанными тонкими и толстыми линиями соответственно

Асимметрия репликации

В то время как биологи будут чувствовать себя как дома со следующим описанием репликации ДНК, ученые-компьютерщики могут найти его перегруженным новыми терминами. Если это кажется слишком сложным с биологической точки зрения, не стесняйтесь пролистывать этот раздел, если вы верите нам, что процесс репликации **асимметричен**, т. е. что прямые и обратные полуцепи при репликации имеют очень разные судьбы.

Поскольку ДНК-полимераза может двигаться только в обратном (3’ → 5’) направлении, она может безостановочно копировать нуклеотиды от *ori* до *ter* вдоль обратных полуцепей. Однако репликация на прямых полуцепях сильно отличается, потому что ДНК-полимераза не может двигаться в прямом (5’ → 3’) направлении; на этих полуцепях ДНК-полимераза должна реплицировать в *обратном направлении*, к *ori*. Взгляните на рис. 1.8, чтобы понять, почему это так.

На прямой полуцепи, чтобы реплицировать ДНК, ДНК-полимераза должна ждать, пока репликационная вилка немного откроется (примерно на 2000 нуклеотидов), пока на *конце* репликационной вилки не сформируется новый праймер; после этого ДНК-полимераза начинает реплицировать небольшой фрагмент ДНК, начиная с этого праймера и двигаясь *назад* в направлении к *ori*. Когда две ДНК-полимеразы на передних полуцепях достигают *ori*, возникает ситуация, показанная на рис. 1.9 ниже. Обратите внимание на разницу между этим рисунком и рис. 1.5.

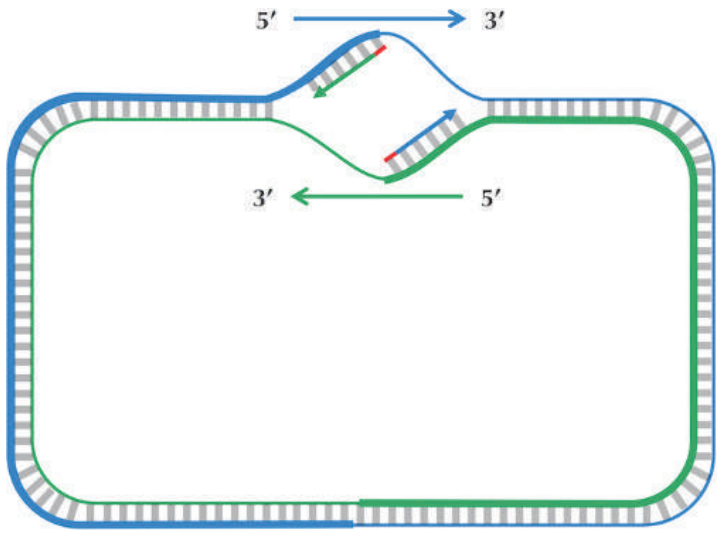


Рис. 1.8 Репликация начинается с *ori* (праймеры показаны красным), с синтеза фрагментов на обратных полуцепях (показаны толстыми линиями). ДНК-полимераза должна дождаться, пока репликационная вилка откроется на некоторое (небольшое) расстояние, прежде чем она начнет копировать передние полуцепи (показаны тонкими линиями) обратно к *ori*

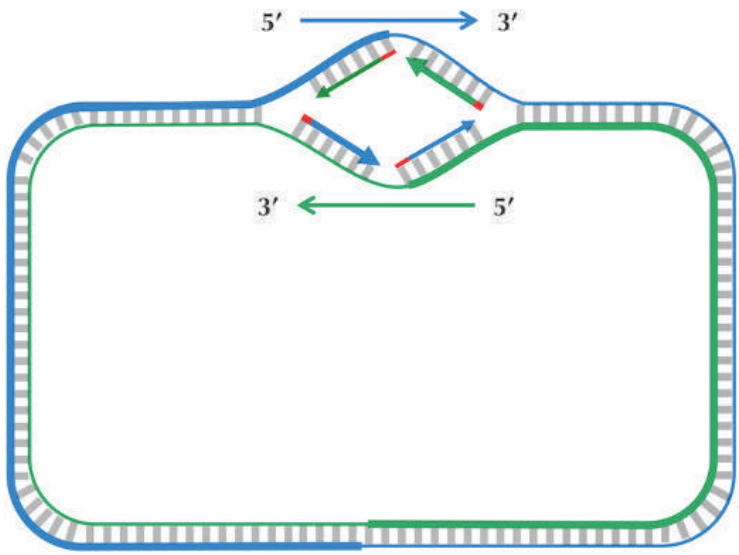


Рис. 1.9 Теперь дочерние фрагменты синтезируются (с некоторой задержкой) на прямых полуцепях (показаны тонкими линиями)

После этого репликация на каждой обратной полуцепи продолжается непрерывно; однако у ДНК-полимеразы на прямой полуцепи нет другого выбора, кроме как снова ждать, пока репликационная вилка не откроет еще 2000 нуклеотидов или около того. Затем требуется новый праймер, чтобы начать синтез другого фрагмента в обратном направлении. В целом репликация на прямой полуцепи требует периодической остановки и перезапуска, что приводит к синтезу коротких **фрагментов Оказаки**, комплементарных интервалам на прямой полуцепи. Вы можете увидеть, как формируются эти фрагменты, на рис. 1.10 (вверху).

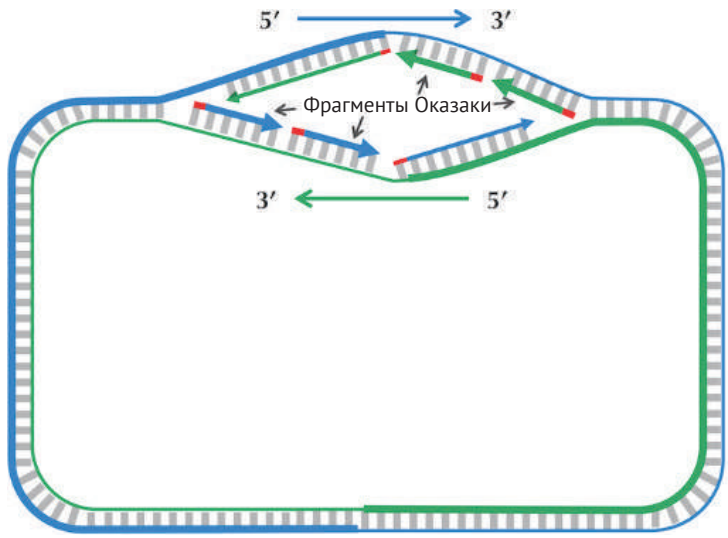


Рис. 1.10 Вилка репликации продолжает расти. Для каждой из обратных полуцепей (показанных жирными линиями) требуется только один праймер, в то время как прямые полуцепи (показаны тонкими линиями) требуют нескольких праймеров для синтеза фрагментов Окадзаки. Два из этих праймеров показаны красным на каждой прямой полуцепи

Когда репликационная вилка достигает *ter*, процесс репликации почти завершен, но между несвязанными фрагментами Окадзаки все еще остаются промежутки, как показано ниже.

Наконец, последовательные фрагменты Окадзаки сшиваются ферментом, называемым **ДНК-лигазой**, в результате чего образуются две интактные дочерние хромосомы, каждая из которых состоит из одной родительской цепи и одной вновь синтезированной дочерней цепи, как показано на рис. 1.12.

На самом деле ДНК-лигаза не ждет, пока все фрагменты Оказаки будут реплицированы, чтобы начать сшивать их вместе.

Биологи называют обратную полуцепь **ведущей**, поскольку всего одна ДНК-полимераза проходит эту полуцепь без остановок, а переднюю полуцепь они

называют **отстающей**, поскольку над ней работают многие ДНК-полимеразы, которые многократно останавливают и вновь начинают репликацию.

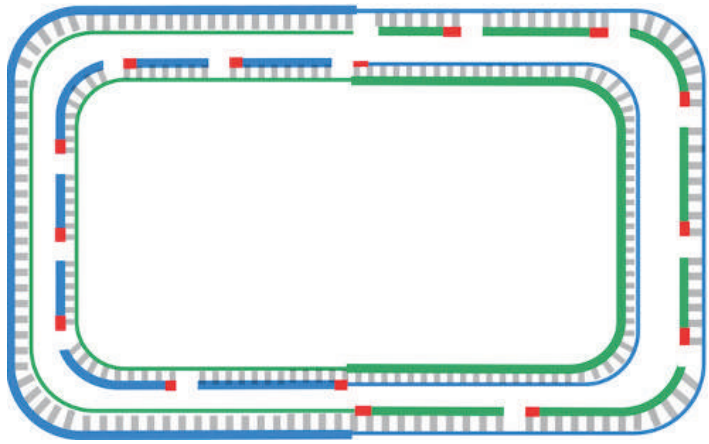


Рис. 1.11 Продолжение процесса репликации

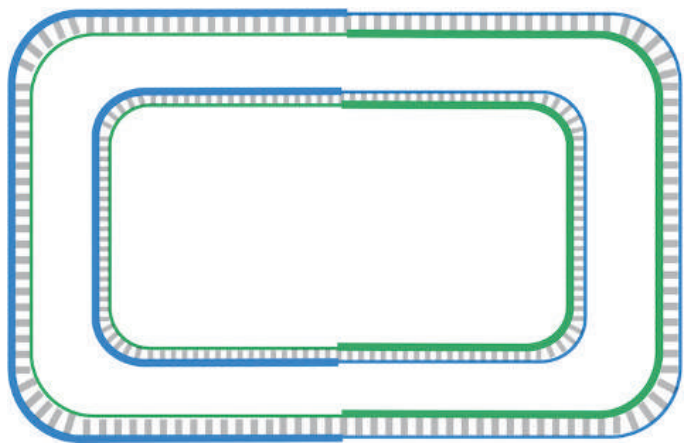


Рис. 1.12 Завершение процесса репликации

Если вы не понимаете различий между ведущими и отстающими полуцепочками, вы не одиноки – мы и легионы студентов-биологов тоже в замешательстве. Путаница усугубляется тем фактом, что в разных учебниках используется разная терминология в зависимости от того, намерены ли авторы ссылаться на ведущую полуцепь матрицы, *с которой синтезируется* дочерняя цепь, или на ту, *которая* синтезируется из (отстающей) полуцепи матрицы. Надеюсь, вы понимаете, почему мы выбрали термины «обратная» и «прямая» полуцепь в попытке смягчить некоторую путаницу.

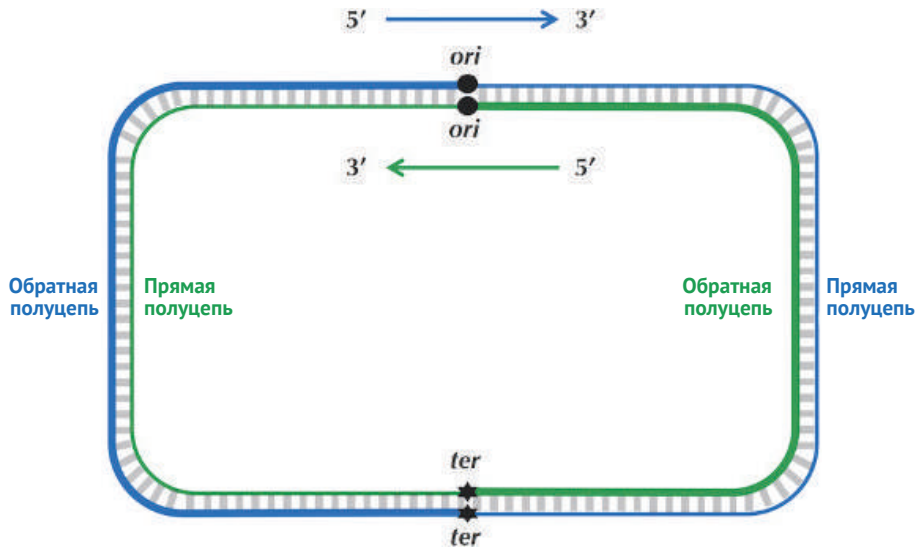


Рис. 1.13 Прямая и обратная полуцепи

Специфическая статистика прямой и обратной полуцепей

Неизвестный биологический феномен или статистическая случайность?

Рисунок 1.14 (вверху слева) показывает удивительную закономерность. Мы разделили геном *E. coli* на 46 фрагментов одинакового размера примерно по 100 000 нуклеотидов, начиная с экспериментально подтвержденного конца репликации, а затем вычислили частоту цитозина в каждом окне. Первые 23 фрагмента (начиная с *ter*) представляют обратную полуцепь, а последние 23 фрагмента (начиная с *ori*) – прямую полуцепь (рис. 1.7). Большинство фрагментов обратной полуцепи имеет высокую частоту цитозина (выше 25 %), тогда как большинство фрагментов прямой полуцепи – низкую цитозиновую частоту (ниже 25 %). Напротив, как показано на рис. 1.14 (вверху справа), большинство фрагментов на обратной полуцепи имеет низкую частоту гуанина (ниже 25 %), тогда как большинство фрагментов на прямой полуцепи – высокую (выше 25 %).

На рис. 1.14 (внизу слева) показаны различия частот G и C в каждом фрагменте генома и представлена еще более поразительная визуализация своеобразной статистики частот нуклеотидов на обратной и прямой полуцепях. Даже если мы предположим, что нам заранее неизвестно местонахождение *ori*, картина все равно проявляется, когда мы начинаем с произвольной позиции генома *E. coli*, как показано на рис. 1.14 (внизу справа).

Если закономерность, которую мы нашли на рис. 1.14, не является статистической случайностью, то мы нашли подсказку о том, как найти *ori*: можно просто пройтись по геному и проверить, где разница между частотой гуанина и цитозина переключается с отрицательной на положительную! Но с какой стати такой простой тест позволяет нам найти источник репликации бактерии?

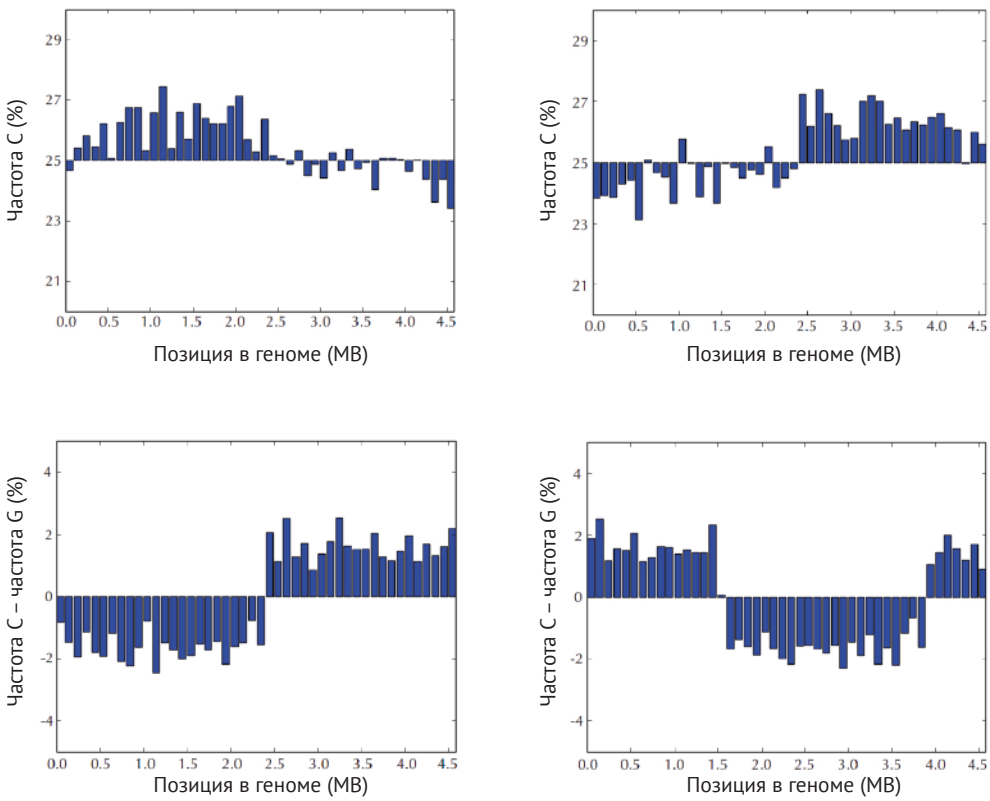


Рис. 1.14 (Вверху слева) Частота цитозина в каждом из 46 непересекающихся фрагментов равной длины (примерно 100 000 нуклеотидов каждый), покрывающих геном *E. coli*. Сайт *ter* находится в положении 0, тогда как сайт *ori* расположен почти напротив *ter* на кольцевой хромосоме *E. coli* примерно в 2,3 млн нуклеотидов от *ter*. Обратная полуцепь охватывает первую половину гистограммы (от 0 до *ori*), тогда как прямая полуцепь – вторую половину гистограммы (начиная с *ori*). (Вверху справа) Частота гуанина в тех же 46 фрагментах генома *E. coli*. (Внизу слева) Разница между частотами встречаемости гуанина и цитозина в 46 фрагментах генома *E. coli* при условии, что геном начинается с экспериментально подтвержденного *ter E. coli*. (Внизу справа) Разница между частотами встречаемости гуанина и цитозина в 46 фрагментах генома *E. coli* при условии, что геном начинается в произвольно выбранном месте

Дезаминирование

В предыдущем разделе мы видели, что по мере расширения репликационной вилки ДНК-полимераза быстро синтезирует ДНК на обратной полуцепи, но испытывает задержки на прямой полуцепи.

Каким образом асимметрия репликации может помочь нам обнаружить *ori*? Обратите внимание, что, поскольку репликация на обратной полуцепи происходит быстро, большую часть своей жизни она остается двухцепочечной. И наоборот, передняя полуцепь проводит гораздо большую часть своей жизни в одноцепочечной форме, ожидая использования в качестве матрицы для репликации. Это несовпадение между прямой и обратной полуцепочками важно, потому что одноцепочечная ДНК имеет гораздо более высокую скорость мутаций, чем двухцепочечная ДНК. В частности, если один из четырех нуклеотидов в одноцепочечной ДНК имеет большую склонность к мутациям, чем другие нуклеотиды, то мы должны наблюдать нехватку этого нуклеотида на прямой полуцепи.

Продолжая эту мысль, давайте рассмотрим количество нуклеотидов в обратной и прямой полуцепях. Подсчет нуклеотидов *Thermotoga petrophila* показан на рис. 1.15. Хотя частоты A и T практически идентичны на двух полуцепях, C чаще встречается на обратной полуцепи, чем на прямой, в результате чего разница составляет 219518 – 207901 = +11617. Его комплементарный нуклеотид G реже встречается на обратной полуцепи, чем на прямой полуцепи, в результате чего разница составляет 201634 – 211607 = –9973.

	#C	#G	#A	#T
Вся цепь	427419	413241	491488	491363
Обратная полуцепь	219518	201634	243963	246641
Прямая полуцепь	207901	211607	247525	244722
Разница	+11617	–9973	–3562	+1919

Рис. 1.15 Счет нуклеотидов в геноме *Thermotoga petrophila* на прямой и обратной полуцепях

ОСТАНОВИТЕСЬ и задумайтесь. Заметили ли вы что-нибудь в счете нуклеотидов на этой таблице?

Хотя частоты A и T практически идентичны на обеих полуцепях, C чаще встречается на обратной, чем на прямой полуцепи, в результате чего разница составляет 219518 – 207901 = **+11617**. Его комплементарный нуклеотид G на обратной полуцепи встречается реже, чем на прямой, в результате чего разница составляет 201634 – 211607 = **–9973**.

Оказывается, мы наблюдаем эти несовпадения, потому что цитозин (C) имеет тенденцию мутировать в тимин (T) в процессе, называемом дезаминированием. Скорость дезаминирования увеличивается в 100 раз, когда ДНК является одноцепочечной, это приводит к уменьшению цитозина в передней полуцепи, что приводит к образованию несовпадающих пар оснований T–G. Эти несо-

впадающие пары могут в дальнейшем мутировать в пары T–A, когда связь восстанавливается в следующем раунде репликации, что объясняет наблюдаемое снижение гуанина (G) на обратной полуцепи (напомним, что прямая родительская полуцепь синтезирует обратную дочернюю полуцепь, и наоборот).

ОСТАНОВИТЕСЬ и задумайтесь. Если дезаминирование превращает цитозин в тимин, почему вы думаете, что прямая полуцепь все еще содержит цитозин?

Диаграмма смещения

Давайте посмотрим, сможем ли мы воспользоваться этой специфической статистикой, вызванной дезаминированием, чтобы локализовать *ori* даже более точно, чем при методе, который мы проиллюстрировали на рис. 1.14. Как показано на рис. 1.15, разница между общим количеством гуанина и общим количеством цитозина является отрицательной для обратной полуцепи (201634 – 219518 = –17884) и положительной для передней полуцепи (211607 – 207901 = 3706). Таким образом, наша идея состоит в том, чтобы пройти по геному, сохраняя промежуточную сумму разницы между количеством G и C. Если эта разница начинает *увеличиваться*, то мы предполагаем, что находимся на прямой полуцепи; с другой стороны, если эта разница начинает *уменьшаться*, то мы предполагаем, что находимся на обратной полуцепи (рис. 1.16).

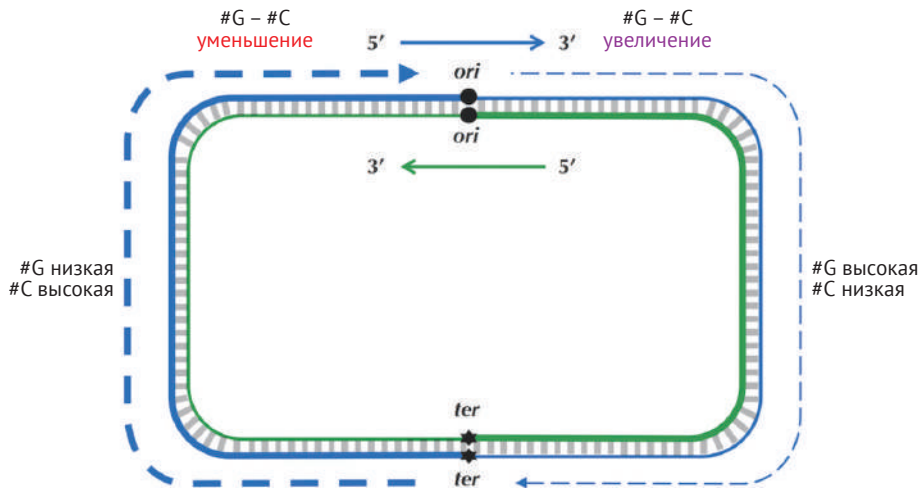


Рис. 1.16 Из-за дезаминирования в каждой прямой полуцепи есть дефицит цитозина по сравнению с гуанином, а в каждой обратной полуцепи дефицит гуанина по сравнению с цитозином. Штриховая синяя линия иллюстрирует воображаемое прохождение по внешней цепи генома со счетом разницы между G и C. Мы предполагаем, что разница между этими значениями положительна на прямой полуцепи и отрицательна на обратной

ОСТАНОВИТЕСЬ и задумайтесь. Представьте, что вы читаете геном (в направлении 5’ → 3’) и замечаете, что разница между количеством гуанина и цитозина только что изменила свое поведение с уменьшения на увеличение. В каком месте генома мы находимся в этот момент?

Поскольку мы не знаем расположение *ori* в кольцевом геноме, давайте линеаризуем его (т. е. выбираем произвольную локацию и делаем вид, что геном начинается здесь), в результате чего получается линейная строка *Genome*. Мы определяем $Skew_i(Genome)$ как разницу между общим количеством местонахождений G и общим количеством местонахождений C в первых i нуклеотидах генома. **Диаграмма смещения** (асимметрии) определяется путем построения $Skew_i(Genome)$, поскольку i находится в диапазоне от 0 до $|Genome|$, где $Skew_0(Genome)$ устанавливается равным нулю. На рис. 1.17 показана диаграмма смещения для короткой цепи ДНК.

Обратите внимание, что мы можем вычислить $Skew_{i+1}(Genome)$ из $Skew_i(Genome)$ в соответствии с нуклеотидом в позиции i генома. Если этот нуклеотид G, то $Skew_{i+1}(Genome) = Skew_i(Genome)+1$; если этот нуклеотид **C**, то $Skew_{i+1}(Genome) = Skew_i(Genome) - 1$; в противном случае $Skew_{i+1}(Genome) = Skew_i(Genome)$.

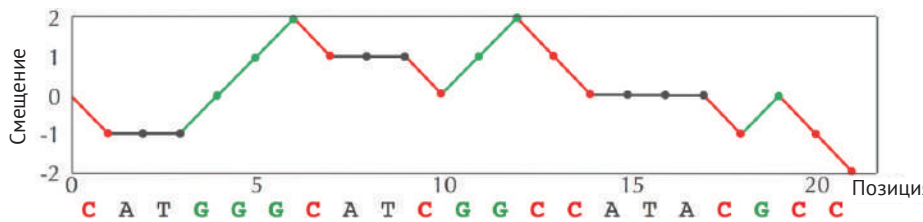


Рис. 1.17 Диаграмма смещения для *Genome* = CATGGGCATCGGCCATACGCC

На рис. 1.18 изображена диаграмма смещения для линеаризованного генома *E. coli*. Обратите внимание на очень четкую последовательность! Оказывается, диаграмма смещения для многих бактериальных геномов имеет сходную характерную форму.

ОСТАНОВИТЕСЬ и задумайтесь. Посмотрев на диаграмму смещения на рис. 1.18, как вы думаете, где, по вашему мнению, находится *ori* в геноме бактерии *E. coli*?

Давайте проследим ДНК в направлении 5’ → 3’ вдоль хромосомы от *ter* к *ori* (по обратной полуцепи), затем продолжим движение от *ori* к *ter* (по прямой

полуцепи). На рис. 1.16 мы видели, что смещение уменьшается по обратной полуцепи и увеличивается по прямой полуцепи. Таким образом, смещение должно составлять минимум в положении, где заканчивается обратная полуцепь и начинается прямая, что и является местом расположения *ori*! Мы только что разработали алгоритм поиска *ori*: его нужно найти там, где смещение достигает минимума.

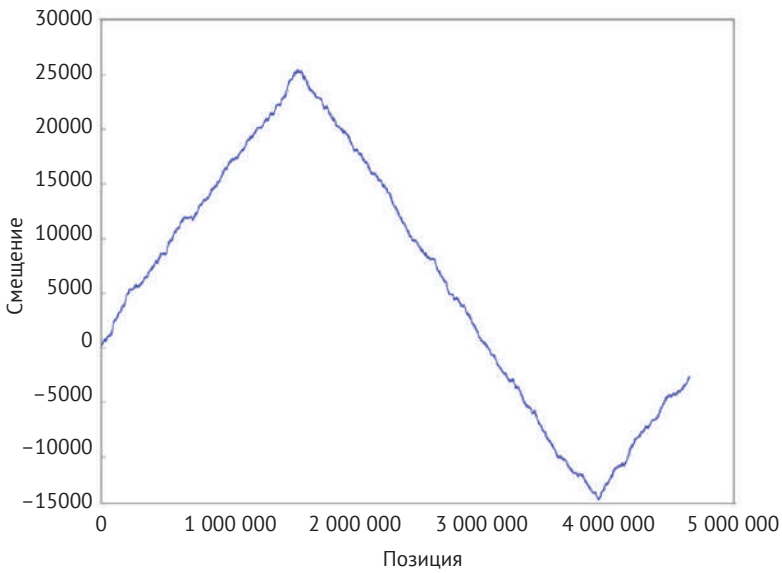


Рис. 1.18 Диаграмма смещения для *E. coli* достигает максимума и минимума в позициях 1550413 и 3923620 соответственно

Задача поиска минимального смещения: *найдите локацию в геноме, где смещение в диаграмме достигает минимума.*

Input: ДНК-цепь *Genome*.

Output: все целые числа *i*, минимизирующие $Skew_i(Genome)$ среди всех значений *i* (от 0 до |*Genome*|).

ОСТАНОВИТЕСЬ и задумайтесь. Обратите внимание, что диаграмма смещения *E. Coli* меняется в зависимости от того, где мы начинаем наше путешествие по кольцевой хромосоме. Считаете ли вы, что ее минимум указывает на одну и ту же позицию *в геноме* независимо от того, где мы начинаем строить эту диаграмму?

Некоторые скрытые сообщения более неуловимы, чем другие

Решение задачи о минимуме смещения дает нам приблизительное местоположение *ori* в позиции 3923620 в *E. coli*. В попытке подтвердить эту гипотезу рассмотрим скрытое сообщение, представляющего потенциальный *DnaA*-бокс рядом с этим местом. Решение задачи часто встречающихся слов в окне длиной 500, начинающемся с позиции 3923620 (показанной ниже), не обнаруживает 9-меров (вместе с их обратными дополнениями), которые появляются три или более раз! Даже если мы обнаружили *ori* в *E. coli*, кажется, что мы все еще не нашли *DnaA*-бокс, который запускает репликацию в этой бактерии...

```
aatgatgatgacgtcaaaaggatccggataaaacatggtgattgcctcgcataa
cgcggtatgaaaatggattgaagcccgggccgtggattctactcaactttgtcg
gcttgagaaagacctgggatcctgggtattaaaaagaagatctatttatttaga
gatctgttctattgtgatctcttattaggatcgcactgccctgtggataacaag
gatccggcttttaagatcaacaacctggaaaggatcattaactgtgaatgatcg
gtgatcctggaccgtataagctgggatcagaatgaggggttatacacaactcaa
aaactgaacaacagttgttctttggataactaccggttgatccaagcttcctga
cagagttatccacagtagatcgcacgatctgtatacttatttgagtaaattaac
ccacgatcccagccattcttctgccggatcttccggaatgtcgtgatcaagaat
gttgatcttcagtg
```

ОСТАНОВИТЕСЬ и задумайтесь. Что бы вы стали делать дальше?

Прежде чем мы сдадимся, давайте еще раз изучим *ori* холерного вибриона, чтобы увидеть, дает ли он нам какое-либо представление о том, как изменить наш алгоритм для поиска *DnaA*-боксов в *E. coli*. Вы, возможно, заметили, что в дополнение к трем вхождениям **ATGATCAAG** и трем вхождениям его обратного комплемента **CTTGATCAT** *Vibrio cholerae ori* содержит дополнительные вхождения **ATGATCAAC** и **CATGATCAT**, которые отличаются от **ATGATCAAG** и **CTTGATCAT** только одним нуклеотидом:

```
atcaATGATCAACgtaagcttctaagcATGATCAAGgtgctcacacagtttatc
cacaacctgagtggatgacatcaagataggtcgttgtatctccttcctctcgta
ctctcatgaccacggaaagATGATCAAGagaggatgatttcttggccatatcgc
aatgaatacttgtgacttgtgcttccaattgacatcttcagcgccatattgcgc
tggccaaggtgacggagcgggattacgaaagCATGATCATggctgttgttctgt
ttatcttgttttgactgagacttgttaggatagacggtttttcatcactgacta
gccaaagccttactctgcctgacatcgaccgtaaattgataatgaatttacatg
cttccgcgacgatttacctCTTGATCATcgatccgattgaagatcttcaattgt
taattctcttgcctcgactcatagccatgatgagctCTTGATCATgtttcctta
accctctatttttacggaagaATGATCAAGctgctgctCTTGATCATcgtttc
```

Обнаружение восьми *приблизительных* вхождений нашего целевого 9-мера и его обратного комплемента в короткой области еще более статистически удивительно, чем обнаружение шести *точных* вхождений **ATGATCAAG** и его обратного комплемента **CTTGATCAT**, на которые мы наткнулись в начале нашего исследования. Кроме того, открытие этих приблизительных 9-меров имеет биологический смысл, поскольку *DnaA* может связываться не только с «идеальными» *DnaA*-боксами, но и с их небольшими вариациями.

Будем говорить, что позиция i в k-мерах $p_1 \ldots p_k$ и $q_1 \ldots q_k$ является **несовпадением**, если $p_i \neq q_i$.

Количество несовпадений между строками p и q называется **расстоянием Хэмминга** между этими строками и обозначается *HammingDistance*(p, q).

Задача определения расстояния Хэмминга: *вычислите расстояние Хэмминга между двумя строками.*

Input: две строки одинаковой длины.

Output: расстояние Хэмминга между этими строками.

Мы говорим, что k-мерный *Pattern* появляется как подстрока текста не более чем с d несовпадениями, если существует некоторый k-мер *Pattern*` подстроки *Text*, имеющий $d$ или меньше несовпадений с *Pattern*, т. е. *HammingDistance*(*Pattern*, *Pattern*`) $\leqslant d$. Наше наблюдение о том, что *DnaA*-бокс может появляться с небольшими вариациями, приводит к следующему обобщению задачи *Pattern*.

Задача поиска приблизительного места вхождений *Pattern* в строку: *найдите все приблизительные вхождения Pattern в строку.*

Input: строки *Pattern* и *Text* вместе с целым числом d.

Output: все начальные позиции, в которых *Pattern* появляется как подстрока *Text* не более чем с d несовпадениями.

Теперь наша цель состоит в том, чтобы модифицировать наш предыдущий алгоритм для задачи о часто встречающихся словах, чтобы найти *DnaA*-боксы путем выявления часто встречающихся k-меров, возможно, с несовпадениями. Имея строки *Text* и *Pattern*, а также целое число d, мы определяем $Count_d$(*Text*, *Pattern*) как количество вхождений *Pattern* в *Text* не более чем с d несовпадениями. Например,

$$Count_1(\text{AACAAGCATAAACATTAAAGAG}, \text{AAAAA})) = 4,$$

потому что **AAAAA** встречается в этой строке четыре раза не более чем с одним несовпадением: **AACAA**, **ATAAA**, **AAACA** и **AAAGA**. Обратите внимание, что два из этих вхождений перекрываются.

Упражнение. Вычислите $Count_2$(AACAAGCATAAACATTAAAGAG, AAAAA).

Вычисление $Count_d$(*Text*, *Pattern*) просто требует, чтобы мы вычислили расстояние Хэмминга между *Pattern* и каждой подстрокой k-мера текста следующим образом.

```
ApproximatePatternCount(Text, Pattern, d)
    count ← 0
    for i ← до |Text| − |Pattern|
        Pattern' ← Text(i, |Pattern|)
        if HammingDistance(Pattern, Pattern') ⩽ d
            count ← count + 1
    return count
```

ОСТАНОВИТЕСЬ и задумайтесь. Каково время работы **ApproximatePatternCount**?

Упражнение. Реализовать **ApproximatePatternCount**. Каково его время расчета?

Наиболее частый *k*-мер с несовпадениями до уровня *d* в *Text* – это просто строка *Pattern*, максимизирующая $Count_d$(*Text*, *Pattern*) среди всех *k*-меров. Обратите внимание, что *Pattern* не обязательно должен фактически отображаться как подстрока *Text*; например, как мы видели выше, **AAAAA** является наиболее частым 5-мером с одним несовпадением в **AA**C**AA**GC**A**T**AAA**C**A**TT**AAA**G**A**G, даже если он не появляется точно в этой строке. Имейте это в виду при решении следующей задачи.

Поиск частых слов с несовпадениями: *Найдите наиболее часто встречающиеся k-меры с несовпадениями в строке.*

Input: Строка Text, а также целые числа *k* и *d*.

Output: Все наиболее часто встречающиеся *k*-меры с несовпадениями в *Text* до *d*.

ЗАРЯДНАЯ СТАНЦИЯ: Решение проблемы часто встречающихся слов с несовпадениями. Один из способов решения задачи поиска часто встречающихся слов с несовпадениями состоит в том, чтобы сгенерировать все 4^k тысяч k-меров *Pattern* и вычислить $Count_d$ (*Text*, *Pattern, d*) для каждого k-мера, а затем вывести k-меры с максимальным числом аппроксимированных вхождений. На практике это неэффективный подход, поскольку многие из 4^k k-меров не должны приниматься во внимание, потому что ни они, ни их мутировавшие версии (с несовпадениями до уровня d) не появляются в *Text*.

Вместо этого следующий псевдокод обобщает функцию **BetterFrequentWords** и использование ею таблицы частот. Он использует единую карту, которая подсчитывает, сколько раз заданная строка имеет приблизительное совпадение в *Text*. Для данной подстроки k-мера *Pattern* строки *Text* нам нужно прибавить 1 к счету каждого k-мера, расстояние Хэмминга в *Pattern* которого не превышает d. Совокупность всех таких k-меров называется ***d*-окрестностью** *Pattern*, обозначаемой **Neighbours**(*Pattern, d*). Ознакомьтесь с разделом **ЗАРЯДНАЯ СТАНЦИЯ: Создайте окрестность строки, чтобы узнать, как реализовать Neighbours**.

```
FrequentWordsWithMismatches(Text, k, d)
    Patterns ← набор строк длины 0
    freqMap ← пустая карта
    n ← |Text|
    for i ← от 0 до n – k
        Pattern ← Text(i, k)
        neighborhood ← Neighbors(Pattern, d)
        for j ← 0 to |neighborhood| – 1
            neighbor ← neighborhood[j]
            if freqMap[neighbor] не существует
                freqMap[neighbor] ← 1
            else
                freqMap[neighbor] ← freqMap[neighbor] + 1
    m ← MaxMap(freqMap)
    for каждого ключа Pattern в freqMap
        if freqMap[Pattern] = m
            добавить Pattern к Patterns
    return Patterns
```

Теперь мы переопределим задачу частых слов, чтобы учесть как несовпадения, так и обратные комплементарности. Напомним, что $Pattern_{rc}$ относится к обратному комплементу *Pattern*.

Задача поиска часто встречающихся слов с несовпадениями и обратными комплементами: *найдите наиболее часто встречающиеся k-меры (с несовпадениями и обратными комплементами) в строке.*

Input: строка ДНК *Text*, а также целые числа *k* и *d*.

Output: все *Pattern* *k*-меров, которые максимизируют сумму $Count_d(Text, Pattern) + Count_d(Text, Pattern_{rc})$ по всем возможным *k*-мерам.

Последняя попытка найти *DnaA*-боксы в *E. Coli*

Теперь мы сделаем последнюю попытку найти *DnaA*-боксы в *E. coli*, находя наиболее часто встречающиеся 9-меры с несовпадениями и обратными комплементами в области, предполагаемой минимальным смещением как *ori*. Хотя минимум диаграммы смещения для *E. coli* находится в позиции 3923620, не следует предполагать, что ее *ori* находится точно в этой позиции из-за случайных флуктуаций смещения. Чтобы решить эту проблему, мы могли бы выбрать больший размер окна (например, $L = 1000$), но расширение окна создает риск того, что мы можем найти другие сгруппированные 9-меры, которые не являются *DnaA*-боксами, но оказываются в этом окне чаще, чем настоящий *DnaA*-бокс. Больший смысл в том, чтобы попробовать маленькое окно, начинающееся, заканчивающееся или центрированное в позиции минимального смещения.

Давайте скрестим пальцы и определим наиболее часто встречающиеся 9-меры (с одним несовпадением и обратными комплементами) в пределах окна длиной 500, начиная с позиции 3923620 генома *E. coli*. Бинго! Экспериментально подтвержденный *DnaA*-бокс в *E. coli* (**TTATCCACA**) представляет собой наиболее часто встречающийся 9-мер с одним несовпадением наряду с его обратным комплементом **TGTGGATAA**:

aatgatgatgacgtcaaaaggatccggataaaacat**ggtgattgcctcgcata**
acgcggtatgaaaatggattgaagcccgggccgtggattctactcaactttgt
cggcttgagaaagacctgggatcctgggtattaaaaagaagatctatttattt
agagatctgttctattgtgatctcttattaggatcgcactgcccTGTGGATAA
caaggatccggcttttaagatcaacaacctggaaaggatcattaactgtgaat
gatcggtgatcctggaccgtataagctgggatcagaatgaggggTTATACACA
actcaaaaactgaacaacagttgttcTTTGGATAActaccggttgatccaagc
ttcctgacagagTTATCCACAgtagatcgcacgatctgtatacttatttgagt
aaattaacccacgatcccagccattcttctgccggatcttccggaatgtcgtg
atcaagaatgttgatcttcagtg

Вы заметите, что мы выделили внутренний интервал этой последовательности более темным текстом. Этот участок является экспериментально подтвержденным *ori E. coli*, который начинается через 37 нуклеотидов после положения 3923620, где смещение достигает своего минимального значения.

Нам повезло, что *DnaA*-боксы *E. coli* были захвачены в выбранном нами окне. Более того, хотя **TTATCCACA** представляет собой наиболее часто встречающийся 9-мер с одним несовпадением и обратной комплементарностью в этом 500-нуклеотидном окне, он не единственный: **GGATCCTGG**, **GATCCCAGC**, **GTTATCCAC**, **AGCTGGGAT** и **CTGGGATCA** также появляются четыре раза с одним несовпадением и обратным комплементом.

ОСТАНОВИТЕСЬ и задумайтесь. В этой главе каждый раз, когда мы находим *ori*, мы, кажется, находим какие-то другие удивительно частые 9-меры. Как вы думаете, почему это так?

Мы не знаем, какой цели – если она вообще существует – эти другие 9-меры служат в геноме *E. coli*, но мы знаем, что в геномах есть *много различных типов* скрытых сообщений; эти скрытые сообщения имеют тенденцию группироваться внутри генома, и большинство из них не имеет ничего общего с репликацией. Одним из примеров являются регуляторные мотивы ДНК, ответственные за экспрессию генов, которые мы будем изучать в главе 2. Важный урок состоит в том, что существующие методы предсказания *ori* остаются несовершенными и иногда неубедительными. Тем не менее даже предоставление биологам небольшой коллекции 9-меров в качестве потенциальных *DnaA*-боксов будет большим подспорьем, если один из этих 9-меров является тем, который мы ищем.

Таким образом, основной смысл этой главы заключается в том, что, хотя компьютерные предсказания могут быть очень действенными, биоинформатики должны сотрудничать с биологами, чтобы проверять свои вычислительные предсказания. Или улучшить эти предсказания: следующий вопрос намекает на то, *как* предсказания *ori* могут быть выполнены с использованием **сравнительной геномики**, метода биоинформации, который использует эволюционное сходство для ответа на трудные вопросы о геномах.

ОСТАНОВИТЕСЬ и задумайтесь. *Salmonella typhimurium* является близким родственником *E. coli*, вызывающей брюшной тиф и болезни пищевого происхождения (рис. 1.19). Узнав, как выглядят *DnaA*-боксы у *E. coli*, как бы вы искали *DnaA*-боксы у *Salmonella enterica*?

У вас будет возможность найти *DnaA*-боксы у *Salmonella enterica* в эпилоге, где будет «Заключительная задача», в которой вас попросят применить то, что вы узнали, к реальному набору данных. В некоторых главах также упоминаются вопросы, оставшиеся без ответа.

Рис. 1.19 *Salmonella typhimurium*

Спасибо, что дочитали до этого места! Мы рассмотрели много материала, но глава еще не закончена. Каждая глава в этой книге сопровождается эпилогом (см. следующий раздел), в котором рассматриваются дополнительные практические соображения или сложности центрального вопроса. Если вам понравился этот материал, пожалуйста, продолжайте! Вам еще многое предстоит узнать об основных алгоритмах, которые помогли превратить биологию в вычислительную дисциплину.

Эпилог. Осложнения в предсказаниях *ori*

В этой главе мы рассмотрели три генома и обнаружили три разных гипотетических 9-мера, кодирующих *DnaA*-боксы: **ATGATCAAG** у *Vibrio cholerae*, **CCTACCACC** у *Thermotoga petrophila* и **TTATCCACA** у *E. coli*. Мы должны предупредить вас, что нахождение *ori* часто оказывается более сложным, чем в трех рассмотренных нами примерах. У некоторых бактерий даже меньше *DnaA*-боксов, чем у *E. coli*, что затрудняет их идентификацию. Локация *ter* часто располагается не прямо напротив *ori*, но может быть значительно смещена, в результате чего прямая и обратная полуцепи имеют существенно разную длину. Положение смещенного минимума часто является лишь грубым индикатором положения *ori*, что вынуждает исследователей расширять окна при поиске *DnaA*-боксов, вводя посторонние повторяющиеся подстроки. Наконец, диаграммы смещения не всегда выглядят так красиво, как у *E. coli*; например, диаграмма смещения для *Thermotoga petrophila* показана на рис. 1.20.

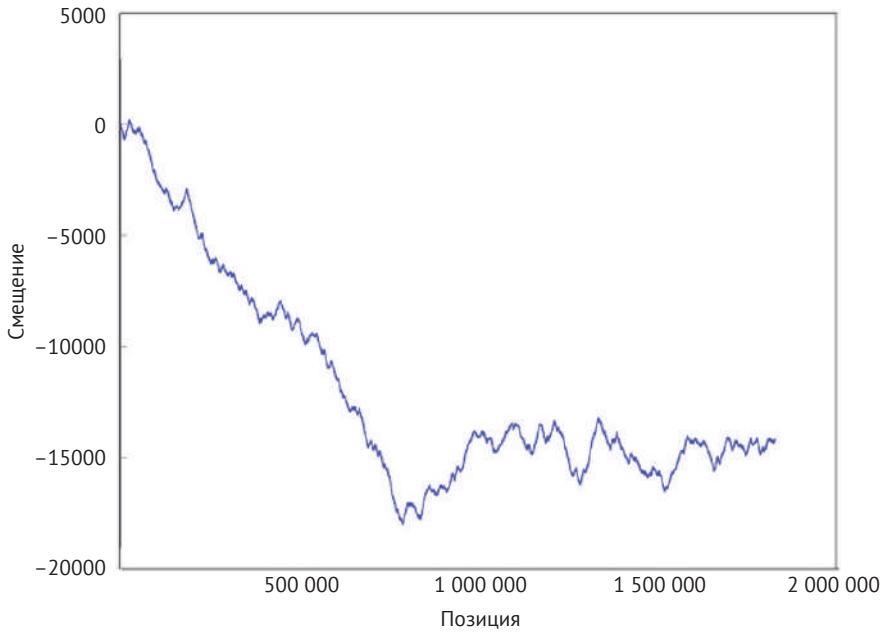


Рис. 1.20 Диаграмма смещения *Thermotoga petrophila* достигает минимума в позиции 787199, но не имеет такой красивой формы, как в диаграмме смещения для *E. coli*

ОСТАНОВИТЕСЬ и задумайтесь. Какой эволюционный процесс мог бы объяснить форму диаграммы смещения для *Thermotoga petrophila*?

Поскольку диаграмма смещения для *Thermotoga petrophila* сложна, а локация *ori* для этого генома даже не подтверждены экспериментально, есть шанс, что область, предсказанная Ori-Finder как область *ori* для *Thermotoga petrophila* (или даже для *Vibrio cholerae*), на самом деле не содержит *ori*!

Теперь у вас уже должно быть хорошее представление о том, как определять местонахождение *ori* и *DnaA*-боксов с помощью вычислений. Далее мы снимем с вас тренировочную защиту и попросим вас решить действительно сложную задачу.

Заключительная задача. Найдите *DnaA*-боксы в *Salmonella enterica*.

☞ **Загрузить данные 1.3**

Примечание Это задание не является обязательным.

Открытые проблемы

Множественные точки начала репликации в бактериальном геноме

Биологи долгое время считали, что каждая бактериальная хромосома имеет только один *ori*. Ванг с сотр. (Wang et al., 2011) генетически модифицировали *E. coli*, вставив синтетический *ori* на миллион нуклеотидов дальше от известного *ori* бактерии. К их удивлению, *E. coli* продолжила работу в обычном режиме, начав репликацию в обоих местах!

После публикации этой статьи сразу же начались поиски встречающихся в природе бактерий с множественными *ori*. В 2012 году Ся (Xia) усомнился в постулате «единого *ori*» и привел примеры бактерий с весьма необычными смещениями. На самом деле наличие более чем одного *ori* имеет смысл в свете эволюции: если геном длинный, а репликация идет медленно, то несколько источников репликации уменьшат количество времени, которое бактерия должна тратить на репликацию своей ДНК.

Например, *Wigglesworthia glossinidia*, симбиотическая бактерия, живущая в кишечнике мухи цеце, имеет нетипичную диаграмму смещения, показанную на рис. 1.21. Поскольку эта диаграмма имеет по крайней мере два ярко выраженных локальных минимума, Ся утверждал, что эта бактерия может иметь две или более областей *ori*.

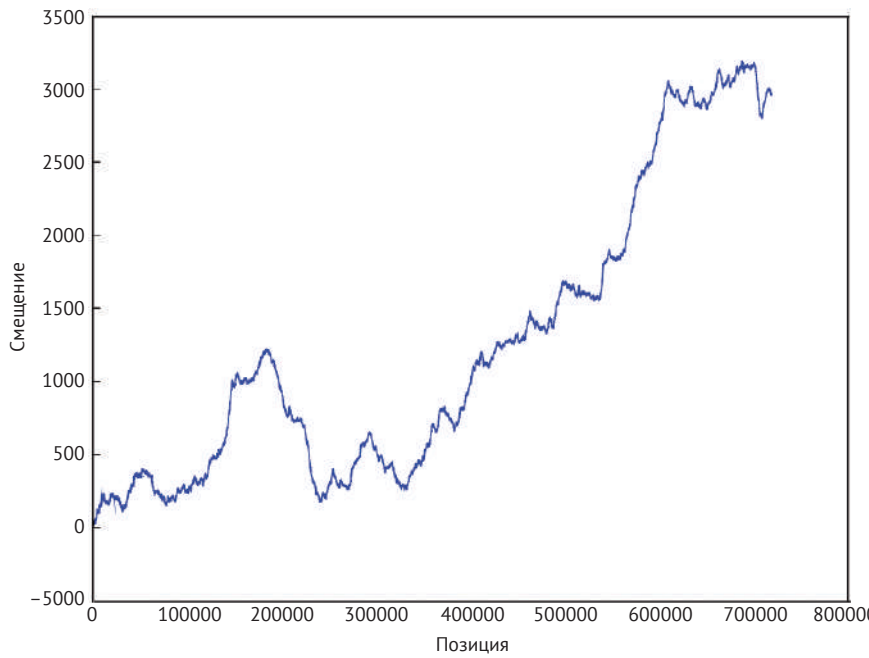


Рис. 1.21 Диаграмма смещения для *Wigglesworthia glossinidia*

Мы должны быть осторожны с гипотезой Ся о том, что эта бактерия имеет две точки начала репликации, поскольку могут быть альтернативные объяснения множественных локальных минимумов в смещении. Например, **рекомбинации генома** (которые мы будем изучать в главе 6) перемещают гены в геноме и часто переносят их с прямой полуцепи на обратную и наоборот, что приводит к аномалиям на диаграмме смещения. Одним из примеров рекомбинации генома является **реверсия**, при которой сегмент хромосомы переворачивается и включается в противоположную цепь; на рис. 1.22 показано, что происходит с диаграммой после реверсии.

Другая трудность связана с тем, что разные виды бактерий могут обмениваться генетическим материалом при **горизонтальном переносе генов**. Если ген из прямой полуцепи одной бактерии перенести на обратную полуцепь другой (или наоборот), то мы будем наблюдать аномалию на диаграмме смещения. В результате остается нерешенным вопрос о количестве *ori* у *Wigglesworthia glossinidia*.

ОСТАНОВИТЕСЬ и задумайтесь. Узнав о сборке генома в главе 3, можете ли вы предложить другое объяснение необычной формы диаграммы смещения *Wigglesworthia glossinidia* на рис. 1.21? (Подсказка: бактериальные геномы иногда собираются неправильно.)

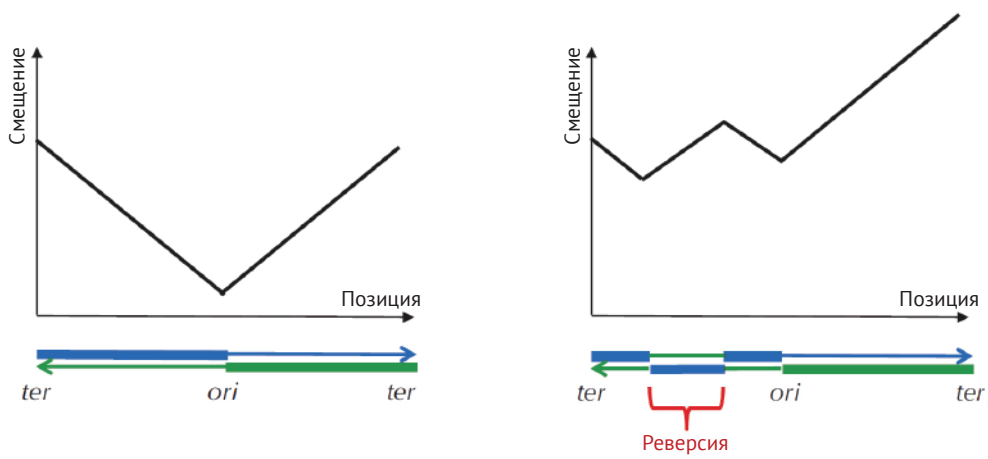


Рис. 1.22 (Слева) «Идеальная» V-образная диаграмма смещения, которая обеспечивает минимальное смещение *ori*. Смещение уменьшается по обратной полуцепи (показана жирной линией) и увеличивается по прямой (показана тонкой линией). Мы предполагаем, что кольцевая хромосома была разрезана на *ter*, в результате чего получилась линейная хромосома, которая начинается и заканчивается на *ter*. (Справа) Диаграмма смещения после реверсии, которая переключает сегменты между обратной и прямой цепочками и изменяет смещение. Как и прежде, смещение уменьшается вдоль сегментов генома, показанных жирными линиями, и увеличивается вдоль сегментов, показанных тонкими

Однако, если бы вы смогли продемонстрировать, что существует два набора идентичных *DnaA*-боксов вблизи двух локальных минимумов на диаграмме смещения *Wigglesworthia glossinidia*, тогда у вас было бы первое убедительное доказательство в пользу множественных точек начала репликации у бактерий. Возможно, просто применив ваше решение задачи часто встречающихся слов с несовпадениями и обратными комплементами, вы обнаружите эти *DnaA*-боксы. Можете ли вы найти другие бактериальные геномы, в которых наличие единственного *ori* вызывает сомнения, и проверить, действительно ли они имеют несколько *ori*?

Поиск источников репликации у архей

Archaea (археи) – одноклеточные организмы, настолько отличные от других форм жизни, что биологи поместили их в особый **домен** одноклеточных организмов, отдельный от бактерий и эукариот. Хотя археи визуально похожи на бактерии, у них есть некоторые геномные особенности, которые тесно связаны с эукариотами. В частности, механизм репликации архей больше похож на механизм эукариот, чем бактерий. Тем не менее археи используют гораздо большее разнообразие источников энергии, чем эукариоты, питаясь аммиаком, металлами и даже газообразным водородом.

На рис. 1.23 показана диаграмма смещения *Sulfolobus solfataricus*, вида архей, обитающих в кислых вулканических источниках при температуре выше 80 °C. На диаграмме можно увидеть по крайней мере три локальных минимума, представленных глубокими спадами, в дополнение к множеству более мелких минимумов.

Лундгрен с сотр. (Lundgren et al., 2004) экспериментально продемонстрировали, что *Sulfolobus solfataricus* действительно имеет три *ori*. С тех пор множественные *ori* были обнаружены у многих других архей. Однако не было разработано точного вычислительного метода для выявления множественных *ori* в недавно секвенированном геноме архей. Например, продуцирующая метан археа *Methanococcus jannaschii* считается рабочей лошадкой геномики архей, но ее происхождение до сих пор остается неустановленным! На ее диаграмме смещения (показанной на рис. 1.24) можно предположить, что у нее может быть несколько точек начала репликации: сможете ли вы их найти?

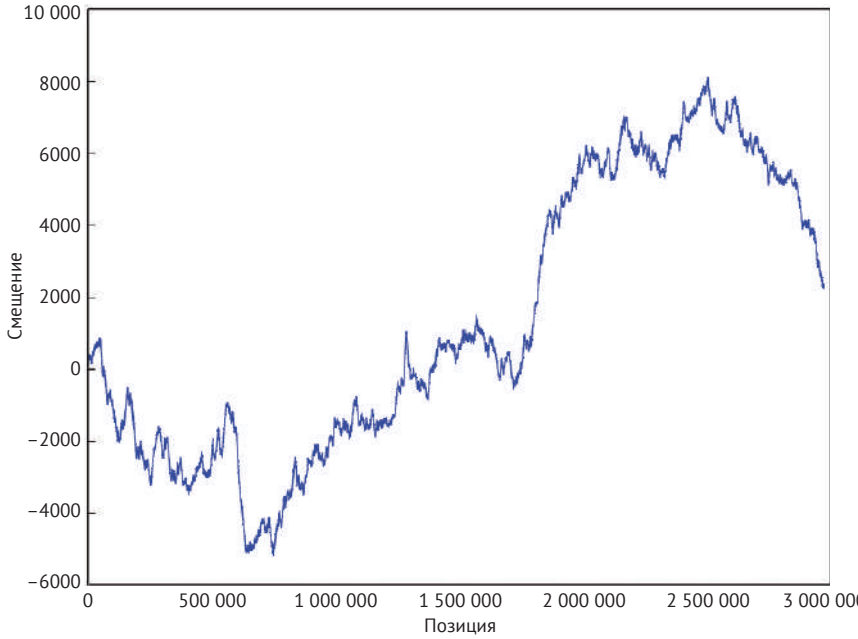


Рис. 1.23 Диаграмма смещения *Sulfolobus solfataricus*

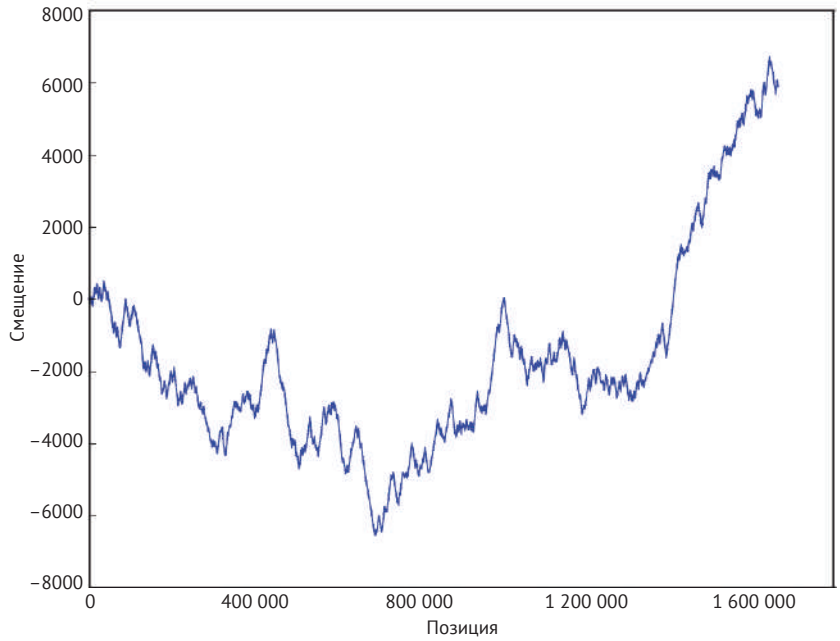


Рис. 1.24 Диаграмма смещения *Methanococcus jannaschii*

Поиск точек начала репликации у дрожжей

Если вы считаете, что найти точки начала репликации у бактерий – сложная задача, подождите, пока вы не проанализируете *ori* в более сложных организмах, таких как дрожжи или человек, у которых сотни точек начала репликации. Среди различных видов дрожжей дрожжи *Saccharomyces cerevisiae* имеют наиболее хорошо описанное начало репликации. Этот вид имеет около 400 различных *ori*, многие из которых могут использоваться во время репликации любой отдельной дрожжевой клетки. Наличие большого количества *ori* приводит к тому, что десятки репликационных вилок устремляются навстречу друг другу из разных мест генома способами, которые еще до конца не изучены. Однако исследователи обнаружили, что источники репликации *S. cerevisiae* имеют общую (несколько изменчивую) последовательность, называемую **ARS-согласованной последовательностью (ACS)**. ACS является сайтом связывания так называемого **Origin Recognition Complex** (Комплекс распознавания точки начала репликации), который инициирует загрузку дополнительных белков, необходимых для запуска репликации в этой точке. Много ACS соответствуют следующей канонической, богатой тимином последовательности длины 11.

TTTAT(G/A)TTT(T/A)(G/T)

Здесь обозначение (X/Y) указывает, что в этом положении может находиться либо нуклеотид X, либо нуклеотид Y.

Однако различные ACS могут отличаться от этой канонической последовательности длиной от 11 до 17 нуклеотидов. Например, 11-нуклеотидная последовательность, показанная выше, часто является частью 17-нуклеотидной последовательности:

(T/A)(T/A)(T/A)(T/A)**TTTAT(G/A)TTT(T/A)(G/T)**(T/G)(T/C)

Недавно был достигнут некоторый прогресс в описании ACS у нескольких других видов дрожжей. У некоторых видов, таких как *S. bayanus*, ACS почти идентична таковому у *S. cerevisiae*, тогда как у других, таких как *K. lactis*, она сильно отличается. Что еще более тревожно, по крайней мере для биоинформатики, что у некоторых видов дрожжей, таких как *S. pombe*, комплекс распознавания точки начала репилкации связывается со слабо определенными AT-богатыми областями, что делает практически невозможным поиск точек начала репликации только на основе анализа последовательности.

Несмотря на недавние усилия, обнаружение точек *ori* в дрожжах остается открытой проблемой, и не существует точного программного обеспечения для определения позиции точки начала репликации на основе анализа последовательности генома дрожжей. Можете ли вы изучить эту проблему и разработать алгоритм для определения позиции *ori* в геноме дрожжей?

Вычисление вероятностей паттернов в строке

В основном тексте мы говорили вам, что вероятность того, что случайная цепь ДНК длиной 500 содержит 9-мер, встречающийся три или более раз, составляет примерно 1/1300. (**СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Вероятности паттернов в строке.**) Мы описываем метод оценки этой вероятности, но он довольно неточен. Эта открытая проблема нацелена на поиск лучших приближений или даже вывод точных формул для вероятностей *Pattern* в строках. (Термин «паттерн» во многих случаях синонимичен терминам «образец» и «последовательность». – *Прим. ред.*)

Начнем с вопроса: какова вероятность того, что конкретный *k*-мер *Pattern* появится (хотя бы один раз) как подстрока случайной строки длины *N*? Этот вопрос оказался не таким уж простым и впервые был поставлен Соловьевым, 1966 (см. также Sedgewick, Flajolet, 2013).

Первый сюрприз заключается в том, что разные *k*-меры могут иметь разные вероятности появления в случайной строке. Например, вероятность того, что *Pattern* = «01» появится в случайной двоичной строке длины 4, равна 11/16, а вероятность того, что *Pattern* = «11» появится в случайной двоичной строке длины 4, равна 8/16. Это явление называется **парадоксом перекрывающихся слов,** потому что разные вхождения строки *Pattern* могут перекрывать друг друга в одних случаях (например, «11»), но не в других (например, «01»). (**СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Вероятности перекрывающихся слов.**)

Нас интересует вычисление следующих вероятностей для случайной строки из *N* букв в алфавите из *A* букв:

- **Pr**(*N, A, Pattern, t*) – вероятность того, что строка *Pattern* появится не менее *t* раз в случайной строке;
- **Pr***(*N, A, Pattern, t*) – вероятность того, что строка *Pattern* и ее образец с обратным комплементом $Pattern_{rc}$ появятся в случайной строке не менее *t* раз.

Обратите внимание, что две приведенные выше вероятности относительно просто вычислить.

Несколько вариантов решения этих вопросов:

- $\mathbf{Pr}_d$(*N, A, Pattern, t*), приблизительная вероятность того, что строка *Pattern* появится не менее *t* раз в случайной строке (не более чем с *d* несовпадениями);
- **Pr**(*N, A, k, t*), вероятность того, что существует любой *k*-мер, появляющийся не менее *t* раз в случайной строке;
- $\mathbf{Pr}_d$(*N, A, k, t*), вероятность того, что существует любой *k*-мер приблизительно не менее чем с *t* вхождениями в случайную строку (не более чем с *d* несовпадениями).

Зарядные станции

Массив частот

Чтобы сделать **FrequentWords** быстрее, мы в первую очередь подумаем, почему этот алгоритм медленный. Он перемещает окно длины k по тексту, идентифицируя k-мерный *Pattern* текста на каждом шаге. Для каждого такого k-мера он должен сдвигать окно по всей длине *Text*, чтобы вычислить *PatternCount*(*Text, Pattern*). Вместо того чтобы делать все это скольжение, мы стремимся сдвинуть окно по *Text* только один раз. При этом мы будем отслеживать, сколько раз каждый k-мер *Pattern* уже появлялся в тексте, обновляя эти числа по мере продвижения.

Для достижения этой цели мы сначала упорядочим все 4^k k-меров **лексикографически** (т. е. в соответствии с тем, как они появляются в словаре), а затем преобразуем их в 4^k различных целых чисел от 0 до $4^k - 1$. Для заданного целого числа k определим частотный массив строки *Text* как массив длины 4^k, где i-й элемент массива содержит количество раз, которое i-й k-мер (в лексикографическом порядке) встречается в *Text* (рис. 1.25).

k-мер	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
частота	3	0	2	0	1	0	0	0	0	1	3	1	0	0	1	0

Рис. 1.25 Лексикографический порядок 2-меров ДНК (вверху), а также индекс каждого 2-мера в этом порядке (в центре) и массив частот для AAGCAAAGGTGGG (внизу). Например, значение частоты с индексом 10 равно 3, потому что GG, десятый 2-мер ДНК в соответствии с лексикографическим порядком, встречается три раза в AAGCAAAAGGTGGG

Чтобы вычислить массив частот, нам нужно определить, как преобразовать k-мер *Pattern* в целое число, используя функцию *PatternToNumber*(*Pattern*). Мы также должны знать, как обратить этот процесс вспять, преобразовав целое число от 0 до $4^k - 1$ в k-мере с помощью функции *NumberToPattern*(*index, k*). На рис. 1.25 показано, что *PatternToNumber*(GT) = 11 и NumberToPattern(11, 2) = GT.

Упражнение. Вычислите следующее.

1. *PatternToNumber*(ATGCAA).
2. *NumberToPattern*(5437, 7).
3. *NumberToPattern*(5437, 8).

Изучите **ЗАРЯДНАЯ СТАНЦИЯ. Преобразование Patterns в Numbers** и наоборот, чтобы увидеть, как реализовать PatternToNumber и NumberToPattern.

Приведенный ниже псевдокод генерирует массив частот, сначала инициализируя каждый элемент в этом массиве нулем (4^k операций), а затем выполняя

один проход по *Text* (приблизительно $|Text| \cdot k$ операций). Для каждого k-мера *Pattern*, с которым мы сталкиваемся, мы добавляем 1 к значению массива частот, соответствующему *Pattern*. Как и прежде, мы называем k-мер, начинающийся в позиции i *Text*, как *Text* (i, k).

```
ComputingFrequencies(Text, k)
    for i ← 0 до 4^k − 1
        FrequencyArray(i) ← 0
    for i ← 0 to |Text| − k
        Pattern ← Text(i, k)
        j ← PatternToNumber(Pattern)
        FrequencyArray(j) ← FrequencyArray(j) + 1
    return FrequencyArray
```

Теперь у нас есть более быстрый алгоритм для задачи о часто встречающихся словах. После создания массива частот мы можем найти все наиболее часто встречающиеся k-меры, просто найдя все k-меры, соответствующие максимальному элементу (элементам) в массиве частот.

```
FasterFrequentWords(Text , k)
    FrequentPatterns ← пустой набор
    FrequencyArray ← ComputingFrequencies(Text, k)
    maxCount максимальная величина в FrequencyArray
    for i ← 0 до 4k  − 1
        if FrequencyArray(i) = maxCount
            Pattern ← NumberToPattern(i, k)
            добавить Pattern к набору FrequentPatterns
    return FrequentPatterns
```

Хотя алгоритм FasterFrequentWords весьма быстр для малых k (т. е. вы можете использовать его для поиска DnaA-боксов в области ori), он становится неэффективным, когда k велико. Если вы знакомы с алгоритмами сортировки и заинтересованы в более быстром алгоритме, ознакомьтесь с разделом **ЗАРЯДНАЯ СТАНЦИЯ: Поиск часто встречающихся слов путем сортировки**.

Упражнение. Мы утверждаем, что высказывание «**FasterFrequentWords** быстрее, чем **FrequentWords**» корректно только для определенных значений $|Text|$ и k. Оцените время выполнения **FasterFrequentWords** и определите значения $|Text|$ и k, когда **FasterFrequentWords** действительно быстрее, чем **FrequentWords**.

Преобразование Patterns в Numbers и наоборот

Наш подход к вычислению *PatternToNumber*(*Pattern*) основан на простом наблюдении. Если мы удалим последний символ из всех лексикографически упорядоченных *k*-меров, результирующий список по-прежнему будет лексикографически упорядочен (подумайте об удалении последней буквы из каждого слова в словаре). В случае цепей ДНК каждый (*k* – 1)-мер в результирующем списке повторяется четыре раза (рис. 1.26).

AAA **AA**C **AA**G **AA**T **AC**A **AC**C **AC**G **AC**T
AGA **AG**C **AG**G **AG**T **AT**A **AT**C **AT**G **AT**T
CAA **CA**C **CA**G **CA**T **CC**A **CC**C **CC**G **CC**T
CGA **CG**C **CG**G **CG**T **CT**A **CT**C **CT**G **CT**T
GAA **GA**C **GA**G **GA**T **GC**A **GC**C **GC**G **GC**T
GGA **GG**C **GG**G **GG**T **GT**A **GT**C **GT**G **GT**T
TAA **TA**C **TA**G **TA**T **TC**A **TC**C **TC**G **TC**T
TGA **TG**C **TG**G **TG**T **TT**A **TT**C **TT**G **TT**T

Рис. 1.26 Если мы удалим последний символ из всех лексикографически упорядоченных 3-меров ДНК, мы получим лексикографический порядок (красных) 2-меров, где каждый 2-мер повторяется четыре раза

Таким образом, количество 3-меров, встречающихся до **AG**T, равно четырехкратному количеству 2-меров, встречающихся до **AG**, плюс количество 1-меров, встречающихся до T. Следовательно,

$$PatternToNumber(\mathbf{AG}T) = 4 \cdot PatternToNumber(\mathbf{AG}) + SymbolToNumber(T) = 8 + 3 = 11,$$

где *SymbolToNumber*(*symbol*) – это функция, преобразующая символы A, C, G и T в соответствующие целые числа 0, 1, 2 и 3.

Если мы удалим последний символ *Pattern*, который обозначим как *LastSymbol*(*Pattern*), то получим (*k* – 1)-мер, который мы обозначим как *Prefix*(*Pattern*). Таким образом, предыдущее наблюдение обобщается до формулы:

$$PatternToNumber(Pattern) = 4 \cdot PatternToNumber(Prefix(Pattern)) + SymbolToNumber(LastSymbol(Pattern)). \quad (*)$$

Это уравнение приводит к следующему рекурсивному алгоритму, т. е. к программе, которая вызывает сама себя. Если вы хотите узнать больше о рекурсивных алгоритмах, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Ханойские башни.**

```
PatternToNumber(Pattern)
    if Pattern не содержит символов
        return 0
    symbol ← LastSymbol(Pattern)
```

```
    Prefix ← Prefix(Pattern)
    return 4 · PatternToNumber(Prefix) + SymbolToNumber(symbol)
```

Чтобы вычислить значение обратной функции *NumberToPattern*(*index, k*), мы возвращаемся к уравнению (*) выше, что означает, что при делении *index* = *PatternToNumber*(*Pattern*) на 4 остаток будет равен *SymbolToNumber*(*symbol*), а частное будет равно *PatternToNumber*(*Prefix*(*Pattern*)). Таким образом, мы можем использовать этот факт для удаления символов в конце строки по одному, как показано на рис. 1.27.

ОСТАНОВИТЕСЬ и задумайтесь. Как только вы вычислите *NumberToPattern*(9904, 7) на рис. 1.27, как вы будете вычислять *NumberToPattern*(9904, 8)?

n	Quotient(*n*, 4)	Remainder(*n*, 4)	NumberToSymbol
9904	2476	0	A
2476	619	0	A
619	154	3	T
154	38	2	G
38	9	2	G
9	2	1	C
2	0	2	G

Рис. 1.27 При вычислении *Pattern* = NumberToPattern(9904, 7) мы делим 9904 на 4, чтобы получить частное 2476 и остаток 0. Этот остаток представляет собой последний нуклеотид *Pattern*, или NumberToSymbol(0) = A. Затем мы повторяем этот процесс, деля каждое последующее частное на 4, пока мы не получим частное 0. Символы в столбце нуклеотидов, читаемые снизу вверх, дают *Pattern* = GCGGTAA

В приведенном ниже псевдокоде мы обозначаем частное и остаток при делении целого числа *n* на целое число *m* как частное *Quotient*(*n, m*) и остаток *Remainder*(*n, m*) соответственно. Например, *Quotient*(11, 4) = 2 и *Remainder*(11, 4) = 3. Этот псевдокод использует функцию *NumberToSymbol*(*index*), которая является обратной функцией *SymbolToNumber* и преобразует целые числа 0, 1, 2 и 3 в соответствующие символы A, C, G и T.

```
NumberToPattern(index, k)
    if k = 1
        return NumberToSymbol(index)
    prefixIndex ← Quotient(index, 4)
    r ← Remainder(index, 4)
    symbol ← NumberToSymbol(r)
```

PrefixPattern ← **NumberToPattern**(*prefixIndex*, *k* – 1)
return конкатенацию *PrefixPattern* с *symbol*

Упражнение. Примените алгоритм, показанный на рис. 1.27, для вычисления *NumberToPattern*(11, 3). Получился ли ответ таким, как вы ожидали? Объясните, что не так с правилом остановки, описанным в подписи к рис. 1.22, а затем исправьте правило.

Поиск часто встречающихся слов путем сортировки

Чтобы увидеть, как сортировка может помочь нам найти часто встречающиеся *k*-меры, мы рассмотрим мотивирующий пример, когда *k* = 2. Дана строка *Text* = AAGCAAAGGTGGG. Перечислите все ее 2-меры в том порядке, в котором они появляются в *Text*, и преобразуйте каждый 2-мер в целое число, используя *PatternToNumber* для создания массива *Index*, как показано ниже.

2-мер	AA	AG	GC	CA	AA	AA	AG	GG	GT	TG	GG	GG
индекс	0	2	9	4	0	0	2	10	11	14	10	10

Теперь мы отсортируем *Index*, чтобы сгенерировать массив *SortedIndex*, как показано на рис. 1.28.

2-мер	AA	AA	SS	AG	AG	CA	GC	GG	GG	GG	GT	TG
SortedIndex	0	0	0	2	2	4	9	10	10	10	11	14
Count	1	2	3	1	2	1	1	1	2	3	1	1

Рис. 1.28 Лексикографически отсортированные 2-меры в AAGCAAAGGTGG (вверху) вместе с массивами SortedIndex (в центре) и Count (внизу)

ОСТАНОВИТЕСЬ и задумайтесь. Как отсортированный массив на рис. 1.28 может помочь нам найти часто встречающиеся слова?

Поскольку идентичные *k*-меры группируются вместе в отсортированном массиве (например, (0, 0, 0) для AA или (10, 10, 10) для GG на рис. 1.28), часто встречающиеся *k*-меры представляют собой самые длинные цепи целых чисел в *SortedIndex*. Это понимание приводит к **FindingFrequentWordsBySorting**, псевдокод которого показан ниже. Этот алгоритм использует массив *Count*, для которого *Count*(*i*) вычисляет, сколько раз целое число в позиции *i* массива *SortedIndex* появляется в первых *i* элементах этого массива (рис. 1.28 (внизу)).

В псевдокоде для **FindingFrequentWordsBySorting** мы предполагаем, что вы уже знаете, как сортировать массив с помощью алгоритма **Sort**.

```
FindingFrequentWordsBySorting(Text , k)
    FrequentPatterns ← пустой набор
    for i ← 0 до |Text| − k
        Pattern ← Text(i, k)
        Index(i) ← PatternToNumber(Pattern)
        Count(i) ← 1
    SortedIndex ← Sort(Index)
    for i ← 1 до |Text| − k
        if SortedIndex(i) = SortedIndex(i − 1)
            Count(i) = Count(i − 1) + 1
    maxCount ← максимальная величина в массиве Count
    for i ← 0 до |Text| − k
        if Count(i) = maxCount
            Pattern ← NumberToPattern(SortedIndex(i), k)
            добавить Pattern к набору FrequentPatterns
    return FrequentPatterns
```

Решение задачи поиска сгустков

> **Примечание** Следующий текст предполагает, что вы прочитали раздел *Массив частот*.

Приведенный ниже псевдокод перемещает окно длиной L вниз по *Genome*. После вычисления массива частот для текущего окна он идентифицирует сгустки, (L, t)-clumps, просто находя, какие k-меры встречаются в окне не менее t раз. Чтобы отслеживать эти сгустки, наш алгоритм использует массив *Clump* длины 4^k, все значения которого инициализируются нулем. Для каждого значения i от 0 до $4^k - 1$ мы установим *Clump*(i) равным 1, если *NumberToPattern*(i, k) образует (L, t)- clump в *Genome*.

```
ClumpFinding(Genome, k, L, t)
    FrequentPatterns ← пустой набор
    for i ← 0 до 4^k − 1
        Clump(i) ← 0
    for i ← 0 до |Genome| − L
        Text ← строка длины L, начинающаяся с позиции i в Genome
        FrequencyArray ← ComputingFrequencies(Text, k)
        for index ← 0 до 4^k − 1
            if FrequencyArray(index) ⩾ t
                Clump(index) ← 1
```

```
for i ← 0 до 4^k − 1
    if Clump(i) = 1
        Pattern ← NumberToPattern(i, k)
        добавить Pattern к набору FrequentPatterns
return FrequentPatterns
```

Упражнение. Оцените время выполнения **ClumpFindin**.

ClumpFinding делает $|Genome| - L + 1$ итераций, генерирующих массив частот для строки длины L на каждой итерации. Поскольку эта задача занимает примерно $4^k + L \cdot k$ времени, общее время выполнения **ClumpFinding** равно $O|Genome| \cdot (4^k + L \cdot k)$. В результате при поиске *DnaA*-боксов ($k = 9$) в типичном бактериальном геноме ($|Genome| > 1000000$) время выполнения **ClumpFinding** становится слишком долгим.

ОСТАНОВИТЕСЬ и задумайтесь. Можно ли ускорить **ClumpFinding**, избавив от необходимости генерировать новый массив частот на каждой итерации?

Чтобы улучшить **ClumpFinding**, обратим внимание на то, что, когда мы сдвигаем наше окно длиной L на один символ вправо, массив частот не сильно меняется, и поэтому пересчет массива частот с нуля неэффективен (рис. 1.29). Это наблюдение помогает нам изменить **ClumpFinding**, как показано ниже. Обратите внимание, что теперь мы вызываем **ComputingFrequencies** только один раз, обновляя массив частот по ходу выполнения задачи.

```
BetterClumpFinding(Genome, k, t, L)
    FrequentPatterns ← пустой набор
    for i ← 0 до 4^k − 1
        Clump(i) ← 0
    Text ← Genome(0, L)
    FrequencyArray ← ComputingFrequencies(Text, k)
    for i ← 0 до 4^k − 1
        if FrequencyArray(i) ⩾ t
            Clump(i) ← 1
    for i 1 до |Genome| − L
        FirstPattern ← Genome(i − 1, k)
        index ← PatternToNumber(FirstPattern)
        FrequencyArray(index) ← FrequencyArray(index) − 1
        LastPattern ← Genome(i + L − k, k)
        index ← PatternToNumber(LastPattern)
```

```
        FrequencyArray(index) ← FrequencyArray(index) + 1
        if FrequencyArray(index) ⩾ t
            Clump(index) 1
    for i ← 0 до 4^k  − 1
        if Clump(i) = 1
            Pattern ← NumberToPattern(i, k)
            добавить Pattern к набору FrequentPatterns
    return FrequentPatterns
```

k-мер	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Frequency	3	0	2	0	1	0	0	0	0	1	3	1	0	0	1	0
Frequency'	2	0	2	0	1	0	0	0	0	2	3	1	0	0	1	0

Рис. 1.29 Массивы частот для двух последовательных подстрок длины 13, начинающихся с позиций 0 и 1 *Text* = AAGCAAAGGTGGGC, отличаются только двумя элементами, соответствующими первому *k*-меру в *Text*(AA) и последнему *k*-меру в *Text*(GC)

Решение задачи часто встречающихся слов с несовпадениями

Примечание Этот раздел использует некоторые обозначения из раздела *Массив частот*.

Чтобы избежать необходимости генерировать все 4^k *k*-меров для решения задачи *FrequentWords* (часто встречающихся слов) с помощью *Mismatces Problem* (задачи несовпадений), лучше рассматривать только те *k*-меры, которые близки к *k*-меру в *Text*, т. е. для которых расстояние Хэмминга от этого *k*-мера не превышает *d*. Таким образом, для заданного паттерна *k*-меров мы определяем его ***d*-окрестность** *Neighbors*(*Pattern, d*) как множество всех *k*-меров, близких к *Pattern*. Например, Neighbors(ACG, 1) состоит из десяти 3-меров:

ACG CCG GCG TCG AAG AGG ATG ACA ACC ACT

Упражнение. Оцените размер *Neighbors*(*Pattern, d*).

Имея целые числа *k* и *d*, мы обобщаем концепцию массива частот для учета приблизительных совпадений *k*-меров с несовпадениями до *d*. Массив частот не более чем с *d* несовпадениями строки *Text* представляет собой массив длины 4^k, где *i*-й элемент массива содержит количество раз, которое *i*-й *k*-мер (в лексикографическом порядке) появляется в *Text* с частотой до *d* несовпадений (рис. 1.30). Например, хотя CT не встречается в AAGCAAAGGTGGG, оно появляется дважды (как CA и GT), если мы допускаем одно несовпадение.

k-мер	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
частота	6	6	9	6	4	2	7	2	9	5	8	5	5	2	6	2

Рис. 1.30 Лексикографический порядок 2-меров ДНК (вверху) вместе с индексом каждого 2-мера в этом порядке (в центре) и массив частот с одним несовпадением для «генома» длиной 14 нуклеотидов AAGCAAAGGTGGG (внизу)

Приведенный ниже псевдокод вычисляет массив частот с несовпадениями до уровня *d* и отличается от **ComputingFrequencies** только тремя строками, показанными синим цветом.

```
ComputingFrequenciesWithMismatches(Text, k, d)
  for i ← 0 до 4k – 1
    FrequencyArray(i) ← 0
  for i ← 0 до |Text| – k
    Pattern ← Text(i, k)
    Neighborhood ← Neighbors(Pattern, d)
    for каждой строки ApproximatePattern в Neighborhood
      j ← PatternToNumber(ApproximatePattern)
      FrequencyArray(j) ← FrequencyArray(j) + 1
  return FrequencyArray
```

Теперь мы можем обобщить **FasterFrequentWords** до алгоритма поиска частых слов с *d* несовпадениями (который мы называем **FrequentWordsWithMismatches**), заменив вызов **ComputingFrequencies** вызовом нашего нового алгоритма **ComputingFrequenciesWithMismatches**.

ОСТАНОВИТЕСЬ и задумайтесь. Хотя **FrequentWordsWithMismatches** работает быстрее, чем наивный алгоритм, описанный в основном тексте главы, для типичных параметров, используемых в поиске *ori*, он не обязательно быстрее для всех значений параметров. Для каких значений параметров **FrequentWordsWithMismatches** медленнее, чем наивный алгоритм?

Генерация окрестности строки

Наша цель – сгенерировать *d*-окрестность *Neighbors*(*Pattern, d*), множество всех *k*-меров, для которых расстояние Хэмминга от *Pattern* не превышает *d*. Сначала мы сгенерируем 1-окрестность *Pattern*, используя следующий псевдокод.

```
ImmediateNeighbors(Pattern)
    Neighborhood ← набор, состоящий из одной строки Pattern
    for i = 1 до |Pattern|
        symbol ← i-й нуклеотид Pattern
        for каждого нуклеотида x, отличного от symbol
            Neighbor ← Pattern с i-м нуклеотидом, замещенным x
            добавить Neighbor к Neighborhood
    return Neighborhood
```

Наша идея создания *Neighbors*(*Pattern, d*) заключается в следующем. Если мы удалим первый символ *Pattern* (обозначенный *FirstSymbol*(*Pattern*)), то получим (*k* – 1)-мер, который мы обозначим *Suffix*(*Pattern*).

ОСТАНОВИТЕСЬ и задумайтесь. Если мы знаем *Neighbors* (Suffix(*Pattern*), *d*), как это поможет нам построить *Neighbors* (*Pattern*, *d*)?

Теперь рассмотрим (*k* – 1)-мер *Pattern*', принадлежащий *Neighbors*(Suffix (*Pattern*), *d*). По определению d-окрестности *Neighbors*(*Suffix*(*Pattern*), d) мы знаем, что *HammingDistance*(*Pattern*', *Suffix*(*Pattern*)) либо равно *d*, либо меньше *d*. В первом случае мы можем добавить *FirstSymbol*(*Pattern*) в начало *Pattern*', чтобы получить *k*-мер, принадлежащий *Neighbors*(*Pattern, d*). Во втором случае можем добавить любой символ в начало *Pattern'* и получить *k*-мер, принадлежащий *Neighbors*(*Pattern, d*).

Например, чтобы сгенерировать *Neighbors*(CAA, 1), сначала сформируем *Neighbors*(AA, 1) = {AA, CA, ГА, TA, AC, АГ, AT}. Расстояние Хэмминга между AA и каждым из этих шести соседей равно 1. Во-первых, объединение с каждым из этих паттернов приводит к шести паттернам (CAA, CCA, CGA, CTA, CAC, CAG, CAT), которые принадлежат *Neighbors*(CAA, 1). Во-вторых, объединение любого нуклеотида с AA приводит к четырем паттернам (AAA, CAA, GAA и TAA), принадлежащим *Neighbors*CAA, 1). Таким образом, *Neighbors*(CAA, 1) содержит десять паттернов.

Это пример рекурсивного алгоритма, или алгоритма, который «вызывает сам себя». Если вы раньше не сталкивались с рекурсивными алгоритмами, изучите **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Ханойские башни**.

В следующем псевдокоде **Neighbors** мы используем *symbol* • *Text* для обозначения конкатенации («склеивания») символа *symbol* и строки *Text*, например A • GCATG = AGCATG.

```
Neighbors(Pattern, d)
    if d = 0
        return {Pattern}
```

```
if |Pattern| = 1
    return {A, C, G, T}
Neighborhood ← пустой набор
SuffixNeighbors ← Neighbors (Suffix(Pattern), d)
for каждой строки Text из SuffixNeighbors
    if HammingDistance(Suffix(Pattern), Text) < d
        for каждого нуклеотида x
            добавить x • Text к Neighborhood
    else
        добавить FirstSymbol(Pattern) • Text к Neighborhood
return Neighborhood
```

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующие вопросы.

1. Каково время работы **Neighbors**?
2. **Neighbors** генерирует все *k*-меры на расстоянии Хэмминга не более *d* от *Pattern*. Измените **Neighbors**, чтобы сгенерировать все *k*-меры на расстоянии Хэмминга *точно d* от *Pattern*.

Если вы все еще изучаете, как работают рекурсивные алгоритмы (например, **Neighbors**), можете вместо этого реализовать итеративную версию **Neighbors**, показанную ниже.

```
IterativeNeighbors(Pattern, d)
    Neighborhood ← набор, состоящий из единственной строки Pattern
    for j = 1 до d
        for каждой строки Pattern' в Neighborhood
            добавить ImmediateNeighbors(Pattern`) к Neighborhood
            удалить дубликаты из Neighborhood
    return Neighborhood
```

Поиск часто встречающихся слов с несовпадениями путем сортировки

Примечание: Этот раздел использует некоторые обозначения из раздела *Поиск часто встречающихся слов путем сортировки*.

Следующий псевдокод упрощает задачу частых слов с несовпадениями при сортировке. Сначала он генерирует всех соседей (до *d* несовпадений) для всех *k*-меров в *Text* и объединяет их всех в массив *NeighborhoodArray*. Обратите внимание, что *k*-мер *Pattern* появляется в этом массиве $Count_d(Text, Pattern)$ раз. Осталось только отсортировать этот массив, подсчитать, сколько раз каждый

k-мер встречается в отсортированном массиве, а затем вернуть *k*-меры, встречающиеся максимальное количество раз.

```
FindingFrequentWordsWithMismatchesBySorting(Text, k, d)
    FrequentPatterns ← пустой набор
    Neighborhoods ← пустой список
    for i ← 0 до |Text| – k
        добавить Neighbors(Text(i, k), d) к Neighborhoods
    сформировать массив NeighborhoodArray, содержащий все строки
Neighborhoods
    for i ← 0 до |Neighborhoods| – 1
        Pattern ← NeighborhoodArray(i)
        Index(i) ← PatternToNumber(Pattern)
        Count(i) ← 1
    SortedIndex ← Sort(Index)
    for i ← 0 до |Neighborhoods| – 2
        if SortedIndex(i) = SortedIndex(i + 1)
            Count(i + 1) ← Count(i) + 1
    maxCount ← максимальная величина в массиве Count
    for i ← 0 до |Neighborhoods| – 1
        if Count(i) = maxCount
            Pattern ← PatternToNumber(SortedIndex(i), k)
            добавить Pattern к FrequentPatterns
    return FrequentPatterns
```

Сопутствующие материалы

Оценка «О большого» (Big-O)[1]

Ученые-информатики обычно измеряют эффективность алгоритма с точки зрения времени его выполнения в наихудшем случае из возможных, когда для самых сложных входных данных заданного объема алгоритм потратит наибольшее время. Преимущество рассмотрения времени выполнения работы в наихудшем случае состоит в том, что мы имеем гарантию, что работа нашего алгоритма никогда не займет большее время.

Оценка Big-*O* компактно описывает время работы алгоритма. Например, если ваш алгоритм сортировки массива из n чисел требует порядка n^2 операций для самого сложного набора данных, то мы говорим, что время работы вашего алгоритма равно $O(n^2)$. На самом деле, в зависимости от вашей реализации, может использоваться любое количество операций, например $1{,}5n^2$, $n^2 + n + 2$ или $0{,}5n^2 + 1$; все эти алгоритмы являются $O(n^2)$, потому что оценка big-*O* заботится только о члене, который растет быстрее всего по отношению к раз-

[1] «*O* большое» и «*o* малое» – математические термины для анализа асимптотического поведения функций. – *Прим. ред.*

меру входных данных. Это связано с тем, что при очень большом n разница в поведении двух функций n^2, таких как $999 \cdot n^2$ и $n^2 + 3n + 9999999$, становится незначительной по сравнению с поведением функций из разных классов, скажем $O(n^2)$ и $O(n^6)$. Конечно, мы бы предпочли алгоритм, требующий $1/2 \cdot n^2$ шагов, алгоритму, требующему $1000 \cdot n^2$ шагов.

Когда мы пишем, что время работы алгоритма $O(n^2)$, мы технически имеем в виду, что оно не растет быстрее, чем функция со старшим членом $c \cdot n^2$ для некоторой константы c. Формально функция $f(n)$ является **Big-*O* от функции *g*(*n*)** или $O(g(n))$, когда $f(n) \leqslant c \cdot g(n)$ для некоторой константы c и достаточно большого n.

Вероятности паттернов в строке

Мы упоминали, что вероятность того, что какой-либо 9-мер появится три или более раз в случайной цепи ДНК длиной 500 нуклеотидов, составляет примерно 1/1300. Уверяем вас, что этот расчет появился не на пустом месте. В частности, мы можем сгенерировать **случайную строку**, моделирующую цепь ДНК, выбирая каждый нуклеотид для любой позиции с вероятностью 1/4. Построение случайных строк можно обобщить на произвольный алфавит с A символами, где каждый символ выбирается с вероятностью $1/A$.

Упражнение. Какова вероятность того, что два случайно сгенерированных фрагмента длины n в алфавите из A букв ({A, C, G, T}) будут идентичными?

Теперь зададим простой вопрос: какова вероятность того, что конкретный k-мер *Pattern* появится (хотя бы один раз) как подстрока случайной строки длины N? Например, мы хотим найти вероятность того, что **01** появится в случайной **двоичной строке** ($A = 2$) длины 4. Вот все возможные такие строки.

```
0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111
```

Поскольку **01** является подстрокой 11 из этих 4-меров и поскольку каждый 4-мер может быть сгенерирован с вероятностью 1/16, вероятность того, что *Pattern* = **01** появится в случайном бинарном 4-мере, равна 11/16.

ОСТАНОВИТЕСЬ и задумайтесь. Какова вероятность того, что *Pattern* = **11** появится как подстрока случайного бинарного 4-мера?

Удивительно, но изменение *Pattern* с **01** на **11** изменяет вероятность того, что он появится как подстрока случайной двоичной строки. Действительно, **11** встречается только в 8 бинарных 4-мерах:

```
0000  0001  0010  0011  0100  0101  0110  0111
1000  1001  1010  1011  1100  1101  1110  1111
```

В результате вероятность появления **11** в случайной двоичной (бинарной) строке длины 4 составляет 8/16 = 1/2.

ОСТАНОВИТЕСЬ и задумайтесь. Почему вы полагаете, что **11** может появиться как подстрока случайного бинарного 4-мера с вероятностью, меньшей, чем **01**?

Пусть *Pr*(*N, A, Pattern, t*) обозначает вероятность того, что строка *Pattern* появится *t* или более раз в случайной строке длины *N*, составленной из алфавита из *A* букв. Мы видели, что *Pr*(4, 2, **01**, 1) = 11/16, а *Pr*(4, 2, **11**, 1) = 1/2. Интересно, что, когда мы делаем *t* больше 1, мы видим, что **01** встречается *реже*, чем **11**. Например, вероятность найти **01** дважды или более в случайном бинарном 4-мере определяется как *Pr*(4, 2, **01**, 2) = 1/16, потому что «0101» является единственным бинарным 4-мером, содержащим **01** дважды, и все же Pr(4, 2, **11**, 2) = 3/16, потому что бинарные 4-меры «0111», «1110» и «1111» имеют по крайней мере два вхождения **11**.

Упражнение. Вычислить *Pr*(100, 2, **01**, 1). Дайте ответ не менее чем с 10 знаками после запятой.

Мы видели, что разные *k*-меры имеют разную вероятность многократного появления в качестве подстроки случайной строки. В общем, это явление называется **парадоксом перекрывающихся слов**, потому что разные вхождения подстроки *Pattern* могут перекрываются друг с другом для некоторых вариантов *Pattern*, но не для других (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Парадокс перекрывающихся слов**).

Например, есть два перекрывающихся вхождения **11** в «1110» и три перекрывающихся вхождения **11** в «1111»; однако вхождения **01** никогда не могут перекрываться друг с другом, и поэтому **01** никогда не может встречаться более двух раз в бинарном 4-мере. Парадокс перекрывающихся слов делает вычисление *Pr*(*N, A, Pattern, t*) довольно сложной задачей, поскольку эта вероятность сильно зависит от конкретного выбора *Pattern*. В свете сложностей, связанных с парадоксом перекрывающихся слов, мы попытаемся *аппроксимировать Pr*(*A, N, Pattern, t*), а не вычислять его точно.

Чтобы аппроксимировать *Pr*(*N, A, Pattern, t*), предположим, что *k*-мер *Pattern* не перекрывается. В качестве игрового примера предположим, что мы хотим подсчитать количество **троичных строк** (*A* = 3) длины 7, содержащих **01** по крайней мере дважды. Помимо двух вхождений **01**, в строке осталось три символа. Предположим, что все эти символы равны «2». Два вхождения **01** могут быть вставлены в «222» десятью различными способами для образования 7-мера, как показано ниже.

0101222 **01**20122 **01**22012 **01**22201 2**01**0122
2**01**2012 2**01**2201 22**01**012 22**01**201 222**01**01

Мы вставили эти два вхождения **01** в «222», но мы могли бы вставить их в любой другой троичный 3-мер. Поскольку существует $3^3 = 27$ троичных 3-меров, мы получаем приближение $10 \cdot 27 = 270$ для числа троичных 7-меров, которые содержат два или более экземпляра **01**. Поскольку имеется $3^7 = 2187$ троичных 7-меров, мы оцениваем вероятность *Pr*(7, 3, **01**, 2) как 270/2187.

ОСТАНОВИТЕСЬ и задумайтесь. Является ли 270/2187 хорошим приближением для Pr(7, 3, **01**, 2)? Точная вероятность Pr(7, 3, **01**, 2) – больше или меньше 270/2187?

Чтобы обобщить описанный выше метод для аппроксимации *Pr*(*N*, *A*, *Pattern*, *t*) для произвольных значений параметров, рассмотрим строку *Text* длины *N*, имеющую не менее *t* вхождений *k*-мера *Pattern*. Если мы выберем ровно *t* из этих вхождений, то мы можем думать о *Text* как о строке из $n = N - t \cdot k$ символов, прерванной *t* вставками вхождений *k*-мера *Pattern*. Если мы зафиксируем эти *n* символов, то мы хотим подсчитать количество различных строк *Text*, которые могут быть образованы вставкой *t* вхождений *Pattern* в строку, образованную этими *n* символами.

В качестве примера снова рассмотрим задачу о встраивании двух вхождений **01** в «222» (*n* = 3) и обратим внимание, что мы добавили пять копий заглавной буквы «X» под каждым из этих 7-меров.

0101222 **01**20122 **01**22012 **01**22201 2**01**0122
X **X** XXX **X** X**X** XX **X** XX**X** X **X** XXX**X** X**X** X XX

2**01**2012 2**01**2201 22**01**012 22**01**201 222**01**01
X**X** X**X** X X**X** XXX**X** XX**X** **X** X XX**X** X**X** XXX**X** **X**

Что означает «X»? Вместо того чтобы подсчитывать количество способов вставить два вхождения **01** в «222», мы можем подсчитать количество способов выбрать два из пяти «X» для окрашивания в синий цвет.

XXXXX **X**X**X**XX **X**XX**X**X **X**XXX**X** X**XX**XX
X**XX**X X**X**XX**X** XX**XX**X XX**X**X**X** XXX**XX**

Другими словами, мы подсчитываем количество способов выбрать 2 из 5 объектов, которые можно посчитать с помощью **биномиального коэффициента** $\binom{5}{2} = 10$ (или «сочетания» C_5^2). В более общем смысле биномиальный коэффициент $\binom{m}{k}$ (или C_m^k) представляет собой количество способов выбрать *k* объектов из *m* объектов, что равно $m!/[k!(m-k)!]$

ОСТАНОВИТЕСЬ и задумайтесь. Сколько существует способов внедрить t экземпляров (непересекающихся) k-меров в строку длины n, чтобы получить строку длины $n + t \cdot k$?

Чтобы аппроксимировать $Pr(N, A, Pattern, t)$, мы хотим подсчитать количество способов вставить t экземпляров k-мера *Pattern* в фиксированную строку длины $n = N - t \cdot k$. Таким образом, у нас будет $n + t$ вхождений «X», из которых мы должны выбрать t для размещения *Pattern*, что в сумме дает $\binom{n+t}{t}$. Затем нам нужно умножить $\binom{n+t}{t}$ на количество строк длины n, в которую мы можем вставить t экземпляров *Pattern*, чтобы приблизительно получить общее количество $\binom{n+t}{t} \cdot A^n$ (фактическое число будет меньше из-за переоценки). Разделив на количество строк длины N, мы получим желаемое приближение,

$$\Pr(N, A, Pattern, t) \approx \frac{\binom{n+t}{t} \cdot A^n}{A^N} = \frac{\binom{N-t\cdot k+t}{t} \cdot A^{N-t\cdot k}}{A^N} = \frac{\binom{N-t\cdot(k-t)}{t}}{A^{t\cdot k}}.$$

Теперь мы вычислим вероятность того, что конкретный 5-мер ACTAT встречается не менее $t = 3$ раз в случайной цепи ДНК ($A = 4$) длины $N = 30$. Поскольку $n = N - t \cdot k = 15$, наша расчетная вероятность является

$$\Pr(30, 4, \texttt{ACTAT}, 3) \approx \frac{\binom{30-3\cdot 4}{3}}{4^{15}} = \frac{816}{1073741824} \approx 7.599 \cdot 10^{-7}.$$

Точная вероятность ближе к $7{,}572 \cdot 10^{-7}$, что свидетельствует о том, что наше приближение является относительно точным для неперекрывающихся k-меров *Pattern*. Однако оно становится неточным для перекрывающихся *Pattern*, например $Pr(30, 4, \texttt{AAAAA}, 3) \approx 1{,}148 \cdot 10^{-3}$.

Нас не должно удивлять, что вероятность обнаружения ACTAT в случайной цепи ДНК длиной 30 нуклеотидов так мала. Но помните, что наша первоначальная цель состояла в том, чтобы аппроксимировать вероятность того, что существует *некоторый* 5-мер, появляющийся три или более раз. В общем, вероятность того, что некоторый k-мер появляется t или более раз в случайной строке длины N, составленной из A-буквенного алфавита, обозначается как $Pr(N, A, k, t)$.

Мы аппроксимировали $Pr(N, A, Pattern, t)$ как

$$p = \frac{\binom{N - t \cdot (k-1)}{t}}{A^{t \cdot k}}.$$

Обратите внимание, что приблизительная вероятность того, что *Pattern* не появится t или более раз, равна $1 - p$. Таким образом, вероятность того, что *все* последовательности A^k встречаются менее t раз в случайной строке длины N, может быть аппроксимирована как

$$(1 - p)^{Ak}.$$

Более того, вероятность того, что существует k-мер, появляющийся t или более раз, должна быть равна 1 минус это значение, что дает нам следующее приближение:

$$\Pr(N, A, k, t) \approx 1 - (1 - p)^{Ak}.$$

У вашего калькулятора могут возникнуть трудности с этой формулой, которая требует возведения числа, близкого к 1, в очень большую степень и может привести к ошибкам округления. Во избежание этого если мы предположим, что p примерно одинаково для любого *Pattern*, то мы можем аппроксимировать $Pr(N, A, k, t)$, умножив p на общее количество k-меров A^k,

$$\Pr(N, A, k, t) \approx p \cdot A^k = \frac{\binom{N - t \cdot (k-1)}{t}}{A^{t \cdot k}} \cdot A^k = \frac{\binom{N - t \cdot (k-1)}{t}}{A^{(t-1) \cdot k}}.$$

Мы снова признаем, что это приближение является грубым, чрезмерным упрощением, поскольку вероятность $Pr(N, A, Pattern, t)$ варьируется в зависимости от выбора k-меров и потому что предполагается, что появления различных k-меров являются независимыми событиями. Например, в основном тексте мы хотим аппроксимировать $Pr(500, 4, 9, 3)$, и приведенная выше формула приводит к аппроксимации

$$\Pr(500, 4, 9, 3) \approx \frac{\binom{500 - 3 \cdot 8}{3}}{4^{(3-1) \cdot 9}} = \frac{17861900}{68719476736} \approx \frac{1}{3847}.$$

Из-за перекрывающихся строк это приближение отличается от истинного значения $Pr(500, 4, 9, 3)$, которое ближе к 1/1300. Чтобы узнать, как получить более точные оценки, см. Guibas and Odlyzko, 1981[1].

[1] http://www.sciencedirect.com/science/article/pii/0097316581900054.

Самый красивый эксперимент в биологии

Эксперимент Мезельсона–Шталя, проведенный в 1958 году Мэтью Мезельсоном и Франклином Шталем, иногда называют «самым красивым экспериментом в биологии». В конце 1950-х годов биологи обсуждали три конфликтующие модели репликации ДНК, показанные на рис. 1.31. **Полуконсервативная гипотеза** (вспомните рис. 1.1) предполагала, что каждая родительская цепь ДНК действует как образец для синтеза дочерней цепи. В результате каждая из двух дочерних молекул содержит одну родительскую цепь и одну вновь синтезированную. **Консервативная гипотеза** предполагала, что вся двухцепочечная родительская молекула ДНК служит матрицей для синтеза новой дочерней молекулы, в результате чего остается одна молекула с двумя родительскими цепочками и получается другая, с двумя вновь синтезированными цепочками. **Дисперсная гипотеза** предполагала, что какой-то механизм разрывает остов ДНК на части и сращивает части синтезированной ДНК, так что каждая из дочерних молекул представляет собой лоскутное одеяло из старой и новой двухцепочечной ДНК.

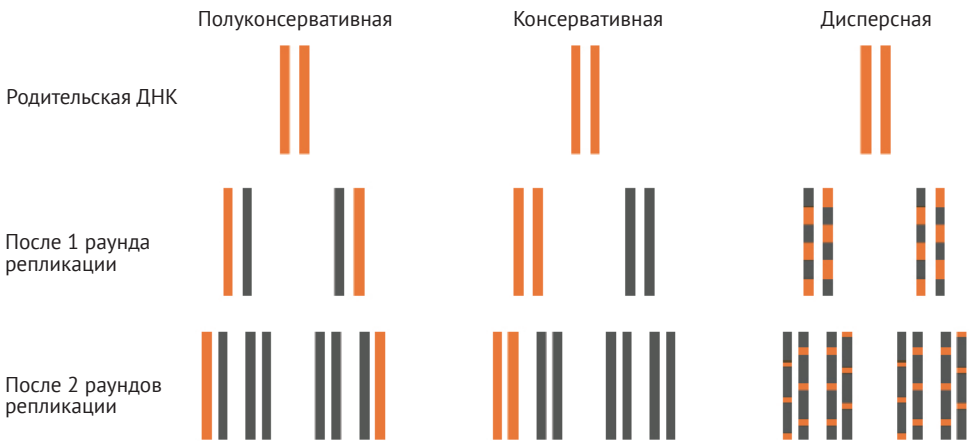


Рис. 1.31 Полуконсервативная, консервативная и дисперсная модели репликации ДНК делают разные предсказания о распределении цепей ДНК после репликации. Желтые цепи содержат изотопы азота ^{15}N («тяжелые» сегменты ДНК), а черные цепи содержат изотопы ^{14}N («легкие» сегменты). Эксперимент Мезельсона–Шталя начался с ДНК, в которой на 100 % содержался изотоп ^{15}N

Мезельсон и Шталь пришли к выводу, что один изотоп азота, **азот-14** (^{14}N), легче и более распространен, чем **азот-15** (^{15}N). Зная, что ДНК естественным образом содержит ^{14}N, Мезельсон и Шталь выращивали *E. coli* для многих циклов репликации в среде ^{15}N, что заставляло бактерии набирать вес по мере того, как они поглощали более тяжелый изотоп для своей ДНК. Когда экспериментаторы убедились, что бактериальная ДНК насыщена ^{15}N, тяжелые клетки *E. coli* переносили на менее плотную среду ^{14}N.

ОСТАНОВИТЕСЬ и задумайтесь. Как вы думаете, что произошло, когда «тяжелая» кишечная палочка размножалась в «легкой» среде ^{14}N?

Гениальность эксперимента Мезельсона–Шталя заключается в том, что вся вновь синтезированная ДНК будет содержать исключительно изотоп ^{14}N, в то время как три существующие гипотезы репликации ДНК предсказывают разные результаты того, как этот изотоп ^{14}N будет включен в ДНК. В частности, после одного цикла репликации консервативная модель предсказала, что половина ДНК *E. coli* по-прежнему будет иметь только ^{15}N и, следовательно, будет тяжелее, тогда как другая половина будет иметь только ^{14}N и будет легче. Тем не менее, когда они попытались разделить ДНК кишечной палочки по весу с помощью центрифуги после одного цикла репликации, вся ДНК имела одинаковую плотность! Вот так они раз и навсегда опровергли консервативную гипотезу.

К сожалению, этот эксперимент не смог исключить ни одну из двух других моделей, поскольку и дисперсная, и полуконсервативная гипотезы предсказывали, что вся ДНК после одного цикла репликации будет иметь одинаковую плотность.

ОСТАНОВИТЕСЬ и задумайтесь. Что предскажут дисперсная и полуконсервативная модели относительно плотности ДНК *E. coli* после двух циклов репликации?

Давайте сначала рассмотрим дисперсную модель, в которой говорится, что каждая дочерняя цепь ДНК образована перемешанными частями – наполовину родительской цепи и наполовину новой ДНК. Если бы эта гипотеза была верна, то после двух циклов репликации любая дочерняя цепь ДНК должна содержать около 25 % ^{15}N и около 75 % ^{14}N. Другими словами, вся ДНК должна иметь одинаковую плотность. И все же, когда Мезельсон и Шталь включили центрифугу после двух циклов репликации *E. coli*, они не наблюдали этого!

Вместо этого они обнаружили, что ДНК разделилась на две разные плотности. Это именно то, что предсказывала полуконсервативная модель: после одного цикла каждая клетка должна иметь одну цепь ^{14}N и одну цепь ^{15}N; после двух циклов половина молекул ДНК должна иметь одну цепь ^{14}N и одну цепь ^{15}N, а другая половина должна иметь две цепи ^{14}N, создавая две измеренные ими разные плотности.

ОСТАНОВИТЕСЬ и задумайтесь. Что предсказывает полуконсервативная модель относительно плотности ДНК *E. coli* после трех раундов репликации?

Мезельсон и Шталь отвергли консервативную и дисперсную гипотезы репликации, и все же они хотели убедиться, что полуконсервативная гипотеза подтверждается дальнейшей репликацией *E. coli*. Эта модель предсказала, что после трех раундов репликации четверть молекул ДНК все еще должна иметь цепь с ^{15}N, в результате чего 25 % ДНК будут иметь промежуточную плотность, тогда как остальные 75 % должны быть легче, имея только ^{14}N. Это действительно то, что Мезельсон и Шталь наблюдали в результате эксперимента, и полуконсервативная гипотеза остается в силе и по сей день.

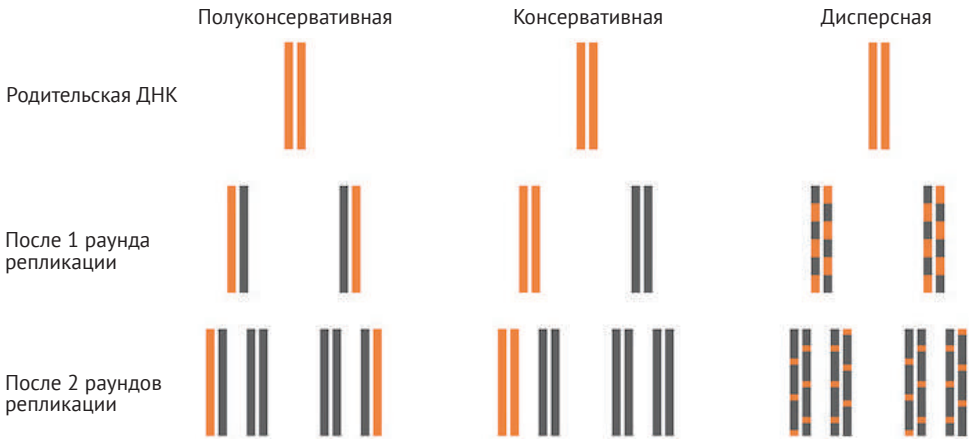


Рис. 1.32 После двух циклов репликации

Направленность цепей ДНК

Сахарный компонент нуклеотида имеет кольцо из пяти атомов углерода, которые обозначены как 1’, 2’, 3’, 4’ и 5’ на рис. 1.33 (слева). 50-й атом присоединяется к фосфатной группе нуклеотида и в конечном итоге к 3’-му концу соседнего нуклеотида. Атом 3’ присоединен к другому соседнему нуклеотиду в цепи нуклеиновой кислоты. В результате мы называем два конца нуклеотида **5’-концом** и **3’-концом**.

Когда мы уменьшим масштаб до уровня двойной спирали, то увидим на рис. 1.33 (справа), что любой фрагмент ДНК ориентирован с 3’-атомами на одном конце и 5’-атомами на другом конце. Как правило, цепь ДНК всегда читается в направлении 5’ → 3’. Обратите внимание, что комплементарные цепи ориентированы противоположно друг другу.

Ханойские башни

Головоломка **«Ханойские башни»** представляет из себя три деревянных вертикальных стержня и несколько деревянных дисков разного размера, каждый из которых имеет отверстие в центре, соответствующее толщине стержней. Диски изначально укладываются на левый стержень (peg 1) таким образом, что диски увеличиваются в диаметре сверху вниз (рис. 1.34). В головоломке нужно перемещать по одному диску между стержнями с целью переместить все диски с левого стержня (peg 1) на правый стержень (peg 3). Однако вам не разрешается размещать больший диск поверх меньшего диска.

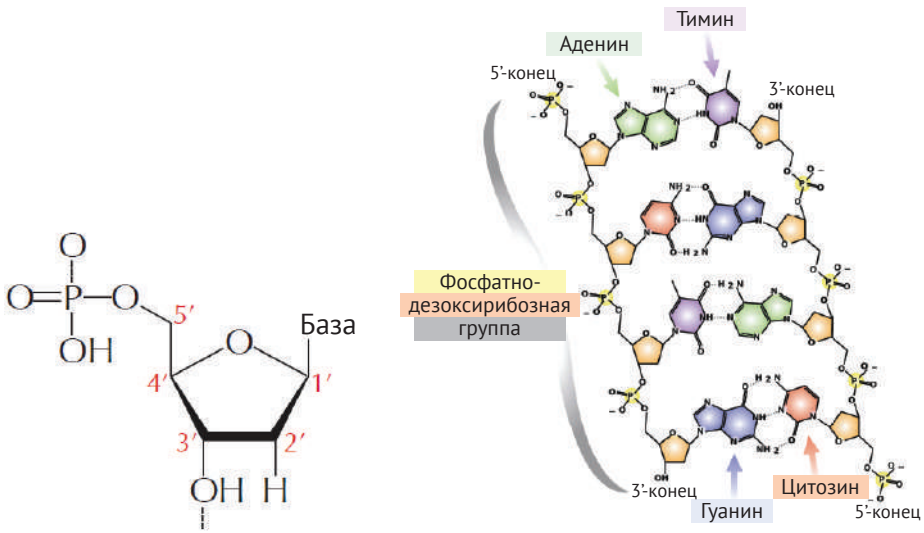


Рис. 1.33 Нуклеотид с атомами углерода сахарного кольца, помеченными 1', 2', 3', 4' и 5'

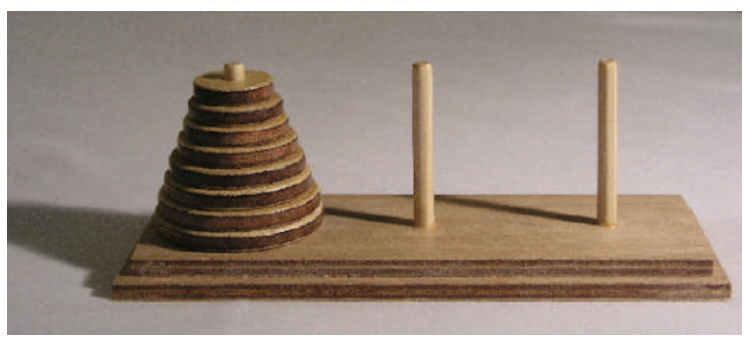


Рис. 1.34 Головоломка «Ханойские башни»

Задача ханойских башен: *решите головоломку «Ханойские башни».*

Input: целое число n.

Output: последовательность ходов, позволяющая решить головоломку «Ханойские башни» с n дисками.

ОСТАНОВИТЕСЬ и задумайтесь. Какое минимальное количество шагов необходимо для решения задачи о ханойских башнях для трех дисков?

Посмотрим, сколько шагов потребуется, чтобы решить задачу о ханойских башнях для четырех дисков. Первое важное замечание заключается в том, что рано или поздно вам придется переместить самый большой диск на правый стержень. Однако, чтобы переместить самый большой диск, нам сначала нужно снять все три самых маленьких диска с первого стержня. Кроме того, все эти три самых маленьких диска должны быть на одном стержне, потому что самый большой диск не может быть помещен поверх другого диска. Таким образом, мы должны сначала переместить верхние три диска на средний стержень (семь ходов), затем передвинуть самый большой диск на правый стержень (один ход), затем снова переместить три самых маленьких диска со среднего стержня поверх самого большого диска на правом стержне (еще семь ходов), всего 15 ходов.

В более общем смысле пусть $T(n)$ обозначает минимальное количество шагов, необходимых для решения головоломки «Ханойские башни» с n дисками. Чтобы переместить n дисков с левого стержня на правый, сначала нужно переместить $n - 1$ самых маленьких дисков с левого стержня на средний стержень ($T(n - 1)$ шагов), затем переместить самый большой диск на правый стержень (1 шаг) и, наконец, переместить $n - 1$ самых маленьких дисков со среднего на правый стержень ($T(n - 1)$ шагов). Этот дает рекуррентное соотношение

$$T(n) = 2T(n - 1) + 1.$$

ОСТАНОВИТЕСЬ и задумайтесь. Используя приведенное выше рекуррентное соотношение, можете ли вы вывести формулу для $T(n)$, не использующую рекурсию?

Теперь у нас есть рекурсивный алгоритм для перемещения n дисков с левого стержня на правый стержень. Мы будем использовать три переменные (каждая из которых принимает различные значения от 1, 2 до 3) для обозначения трех стержней: *startPeg*, *destinationPeg* и *transportPeg*. Эти три переменные всегда

представляют разные стержни, поэтому *startPeg* + *destinationPeg* + *transportPeg* всегда равно 1 + 2 + 3 = 6. **HanoiTowers**(*n, startPeg, destinationPeg*) перемещает *n* дисков со *startPeg* на *destinationPeg* (используя *transportPeg* в качестве временного места размещения).

```
HanoiTowers(n, startPeg, destinationPeg)
    if n = 1
        Переместите верхний диск со startPeg на destinationPeg
        return
    transitPeg = 6 – startPeg – destinationPeg
    HanoiTowers(n – 1, startPeg, transitPeg)
    Переместите верхний диск со startPeg на destinationPeg
    HanoiTowers(n – 1, transitPeg, destinationPeg)
    Return
```

Хотя этот алгоритм может показаться простым, для перемещения башни из 100 дисков потребуется больше шагов, чем количество атомов во Вселенной! Быстрый рост количества ходов, требуемых для решения **HanoiTowers**, объясняется тем, что каждый раз **HanoiTowers** вызывается для *n* дисков, дважды вызывает себя для *n* – 1, что, в свою очередь, инициирует четыре вызова для *n* – 2 и т. д. Например, вызов **HanoiTowers** (4, 1, 3) приводит к вызовам **HanoiTowers** (3, 1, 2) и **HanoiTowers** (3, 2, 3); эти вызовы, в свою очередь, вызывают **HanoiTowers** (2, 1, 3), **HanoiTowers** (2, 3, 2), **HanoiTowers** (2, 2, 1) и **HanoiTowers** (2, 1, 3).

Парадокс перекрывающихся слов

Мы иллюстрируем парадокс перекрывающихся слов на примере игры для двух игроков под названием «Лучшая ставка для простаков». Игрок 1 выбирает двоичный *k*-мер *A*, а Игрок 2, зная, что такое *A*, выбирает другой двоичный *k*-мер *B*. Затем два игрока подбрасывают монету несколько раз, причем подбрасывание монеты представлено строками «1» (орел) и «0» (решка); игра заканчивается, когда *A* или *B* появляется как блок из *k* последовательных подбрасываний монеты.

ОСТАНОВИТЕСЬ и задумайтесь. Всегда ли у двух игроков одинаковые шансы на победу?

На первый взгляд можно предположить, что у всех *k*-меров равные шансы на победу. Однако предположим, что Игрок 1 выбирает «00», а Игрок 2 выбирает «10». После двух бросков либо игрок 1 побеждает («00»), либо побеждает игрок 2 («10»), либо игра продолжается («01» или «11»). Если игра продолжится, то Игрок 1 должен сдаться, поскольку Игрок 2 выиграет, как только выпадет решка («0»). Таким образом, у Игрока 2 в три раза больше шансов на победу!

Может показаться, что Игрок 1 должен иметь преимущество, просто выбрав «самый сильный» k-мер. Однако интригующая особенность «Лучшей ставки для простаков» состоит в том, что если $k > 2$, то Игрок 2 всегда может выбрать k-мер B, который лучше A, независимо от выбора Игроком 1 A. Еще одним сюрпризом является то, что «Лучшая ставка для простаков» – это **нетранзитивная игра**: если A побеждает B, а B побеждает C, то мы не можем автоматически заключить, что A побеждает C (как в игре «Камень, ножницы, бумага»).

Чтобы проанализировать лучший вариант для простаков, мы говорим, что для B ***i*-перекрытие** с A означает то, что последние i цифр A совпадают с первыми i цифрами B. Например, «110110» имеет 1-перекрытие, 2-перекрытие и 5-перекрытие с «011011», как показано на рис. 1.29.

Для двух k-меров A и B **корреляция** A и B, обозначаемая $Corr(A, B) = (c_0,\ldots,c_{k-1})$, представляет собой k-буквенное двоичное слово такое, что $c_i = 1$, если B имеет $(k - i)$-перекрытие с A и 0 в противном случае. **Частота корреляции** A и B определяется как

$$K_{A,B} = c_0 + c_1 \cdot \frac{1}{2} + c_2 \cdot \left(\frac{1}{2}\right)^2 + \cdots + c_{k-1} \cdot \left(\frac{1}{2}\right)^{k-1}.$$

Для строк A и B на рис. 1.35 корреляция составляет «010011», а частота корреляции равна $K_{A,B} = (1/2) + (1/2)^4 + (1/2)^5 = 19/32$.

	Corr(*A*, *B*)
B = 110110	**0**
B = **11011**0	**1**
B = **1**10110	**0**
B = 110110	**0**
B = **11**0110	**1**
B = **1**10110	**1**
A = **011011**	

Рис. 1.35 Корреляция k-меров A = «011011» и B = «110110» представляет собой строку «010011»

Для двух k-меров A и B **корреляция** A и B, обозначаемая $Corr(AB) = (c_0, \ldots, c_{k-1})$, представляет собой k-буквенное двоичное слово такое что $c_i = 1$, если $B(k - i)$-перекрывается с A и 0 в противном случае. **Частота корреляции** A и B определяется как

$$\frac{K_{A,A} - K_{A,B}}{K_{B,B} - K_{B,A}}.$$

Для строк *A* и *B* на приведенном выше рисунке их корреляция равна «010011», а их частота корреляции равна

$$K_{A,B} = (1/2) + (1/2)^4 + (1/2)^5 = 19/32.$$

Математик Джон Конвей предложил следующую обманчиво простую формулу для вычисления вероятности того, что *B* победит *A*:

$$(K_{A,A} - K_{A,B})/(K_{B,B} - K_{B,A}).$$

Конвей никогда не публиковал доказательство этой формулы, а Мартин Гарднер, ведущий популярный писатель-математик, сказал об этой формуле следующее:

> *«Я понятия не имею, почему она работает. Она просто выдает ответ как по волшебству, как и многие другие алгоритмы Конвея».*

Библиографические примечания

Использование смещения для поиска точек начала репликации было впервые предложено Lobry, 1996[1], а также описано у Grigoriev, 1998[2]. Grigoriev, 2011[3], представляет собой отличное введение в метод смещения, а Sernova and Gelfand, 2008[4], дали обзор алгоритмов и программных средств для поиска начала репликации у бактерий. Lundgren et al., 2004[5], продемонстрировали, что археи могут иметь множество точек начала репликации. Wang et al, 2011[6], вставили искусственный *ori* в геном *E. coli* и показали, что он запускает репликацию. Xia, 2012[7], был первым, кто предположил, что бактерии могут иметь множество начал репликации. Gao and Zhang, 2008[8], разработали программу Ori-Finder для поиска начальных точек репликации у бактерий.

---

1 https://www.ncbi.nlm.nih.gov/pubmed/8676740.
2 https://www.ncbi.nlm.nih.gov/pubmed/9580676.
3 https://www.cambridge.org/core/books/bioinformatics-for-biologists/how-do-replication-and-transcription-change-genomes/032192B2F465D63B9CC9BD5941CD7800.
4 https://www.ncbi.nlm.nih.gov/pubmed/18660512.
5 https://www.ncbi.nlm.nih.gov/pubmed/15107501.
6 https://www.pnas.org/content/108/26/E243.
7 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3269012/.
8 https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-79.

Liachko et al., 2013[1], представили наиболее полное описание происхождения репликации дрожжей. Solov'ev, 1966[2], впервые вывел точные формулы для аппроксимации вероятностей сегментов в строке. Gardner, 1974[3], написал прекрасную вступительную статью о парадоксе «Лучший выбор для простаков». Guibas and Odlyzko, 1981[4], предоставили превосходное освещение парадокса перекрывающихся слов, который иллюстрирует сложность вычисления вероятностей сегментов в случайном тексте. Они также получили довольно сложное доказательство формулы Конвея для лучшего «выбора простаков». Sedgewick and Flajolet, 2013[5], представили обзор различных методов вычисления вероятностей паттернов в строке.

[1] https://www.ncbi.nlm.nih.gov/pubmed/23241746.
[2] https://epubs.siam.org/doi/10.1137/1111022.
[3] https://ansible.uk/misc/mgardner.html.
[4] https://www.sciencedirect.com/science/article/pii/0097316581900054.
[5] https://aofa.cs.princeton.edu/home/.

Глава 2

Какие сегменты ДНК играют роль молекулярных часов?

Рандомизированные алгоритмы

Есть ли у нас «часовой ген»?

Распорядок дня животных, растений и даже бактерий контролируется внутренним хронометром, называемым **циркадными часами**. (Другие названия – «циркадный осциллятор» или «эндогенные часы». – *Прим. ред.*) Любой, кто испытал страдания от смены часовых поясов, знает, что эти часы никогда не перестают тикать. Как лабораторные крысы, так и добровольцы-исследователи, будучи помещенными в бункер, естественным образом поддерживают примерно 24-часовой цикл активности и отдыха в полной темноте. И, как любые часы, циркадные часы могут дать сбой, что является следствием генетического заболевания, известного как **синдром задержки фазы сна (DSPS)**. («Синдром позднего засыпания». – *Прим. ред.*)

Циркадные часы, очевидно, должны иметь некоторую основу на молекулярном уровне, что вызывает много вопросов. Как *отдельные клетки* животных и растений (не говоря уже о бактериях) узнают, когда им следует замедлить или увеличить выработку определенных белков? Существует ли «часовой ген»? Можем ли мы объяснить, почему сердечные приступы чаще случаются утром, а приступы астмы – ночью? И можем ли мы идентифицировать гены, ответственные за «сбои» циркадных часов, вызывающие DSPS?

В начале 1970-х годов Рон Конопка и Сеймур Бензер выявили мух-мутантов с аномальными циркадными ритмами и проследили мутации мух до конкретного гена. Биологам понадобилось еще два десятилетия, чтобы обнаружить аналогичный часовой ген у млекопитающих, что было лишь первой частью пазла. Сегодня обнаружено гораздо больше циркадных генов; это гены, имеющие такие названия, как *вневременные гены*, *гены циркадных ритмов* или *циклические гены*; они управляют поведением сотен других генов и демонстрируют высокую степень эволюционной консервативности у разных видов.

Сначала мы сосредоточимся на растениях, поскольку поддержание циркадных ритмов у растений – это вопрос жизни и смерти. Подумайте, сколько генов растений должно обращать внимание на время восхода и захода солнца; действительно, по оценкам биологов, циркадными являются более тысячи генов растений, включая гены, связанные с фотосинтезом, фоторецепцией и цветением. Эти гены должны каким-то образом знать, который сейчас час, чтобы изменять выработку своих генных транскриптов или **экспрессию генов** в течение дня (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Экспрессия генов**).

Оказывается, каждая растительная клетка отслеживает день и ночь независимо от других клеток, и всего три растительных гена, называемые LHY, CCA1 и TOC1, являются главными хранителями времени. Такие регуляторные гены и **регуляторные белки**, которые они кодируют, часто контролируются внешними факторами (например, доступностью питательных веществ или солнечным светом), что позволяет организмам регулировать экспрессию своих генов.

Например, регуляторные белки, контролирующие циркадные ритмы у растений, координируют циркадную активность следующим образом. TOC1 способствует экспрессии LHY и CCA1, тогда как LHY и CCA1 подавляют экспрессию TOC1, что приводит к **отрицательной обратной связи**. Утром солнечный свет

активирует транскрипцию LHY и CCA1, запуская репрессию транскрипции TOC1. По мере уменьшения света уменьшается и производство LHY и CCA1, которые, в свою очередь, больше не подавляют TOC1. Транскрипция TOC1 достигает пика ночью и начинает способствовать транскрипции LHY и CCA1, которые, в свою очередь, подавляют транскрипцию TOC1, и цикл начинается снова.

LHY, CCA1 и TOC1 способны контролировать транскрипцию других генов, потому что регуляторные белки, которые они кодируют, являются **транскрипционными факторами** или главными регуляторными белками, которые включают и выключают другие гены. Транскрипционный фактор регулирует ген, связываясь со специфическим коротким интервалом ДНК, называемым **регуляторным мотивом** или **сайтом связывания фактора транскрипции**, в **восходящей области** гена, области длиной 600–1000 нуклеотидов, предшествующей началу гена. Например, CCA1 связывается с AAAAAATCT в восходящей области многих генов, регулируемых CCA1.

Жизнь биоинформатика была бы легкой, если бы регуляторные мотивы были полностью фиксированы, но реальность более сложна, они могут различаться в некоторых позициях, например CCA1 может альтернативно связываться с **AAGAACTCT**. Но как мы можем определить местонахождение этих регуляторных мотивов, не зная заранее, как они выглядят? Нам нужно разработать алгоритмы для **поиска мотивов**, задачи обнаружения скрытого сообщения, общего для набора строк.

Найти мотив сложнее, чем вы думаете

Идентификация вечернего элемента

В 2000 году Стив Кей использовал **ДНК-чипы** (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: ДНК-чипы**), чтобы определить, какие гены растения *Arabidopsis thaliana* активируются в разное время дня. Затем он извлек восходящие области почти 500 генов, демонстрирующих циркадное поведение, и искал часто появляющиеся последовательности в их восходящих областях. Если вы соедините эти восходящие области в одну строку, вы обнаружите, что слово AAAATATCT встречается на удивление часто, 46 раз.

ОСТАНОВИТЕСЬ и задумайтесь. Каковы возможные недостатки объединения всех восходящих областей в одну строку и поиска часто встречающихся слов при определении мотива?

Упражнение. Определите ожидаемое количество вхождений 9-мера в 500 случайных строках ДНК, каждая из которых имеет длину 1000. **Примечание.** Выразите ответ в виде десятичной дроби с точностью до 0,0001.

Кей назвал AAAATATCT **вечерним элементом** и провел простой эксперимент, чтобы доказать, что это действительно регуляторный мотив, ответственный за циркадную экспрессию генов у *Arabidopsis thaliana*. После того как он вызвал мутацию вечернего элемента в восходящей области одного гена, ген больше не проявлял циркадных свойств.

Так как вечерний элемент в растениях очень консервативен, его легко найти. Мотивы, имеющие множество мутаций, менее уловимы. Например, если вы заразите муху бактерией, муха включит свои **иммунные гены** для борьбы с инфекцией. Таким образом, некоторые из генов с повышенным уровнем экспрессии после инфекции, вероятно, являются иммунными. Действительно, некоторые из этих генов имеют 12-меры, сходные с TCGGGGATTTCC, в их восходящей области, месте связывания фактора транскрипции, называемого **NF-kB**, который активирует различные иммунные гены у мух. Однако сайты связывания NF-kB далеко не так консервативны, как вечерний элемент. На рис. 2.1 показаны десять сайтов связывания NF-kB из генома *Drosophila melanogaster*; наиболее популярные нуклеотиды в каждом столбце показаны цветными буквами в верхнем регистре.

```
 1 T C G G G G g T T T t t
 2 c C G G t G A c T T a C
 3 a C G G G G A T T T t C
 4 T t G G G G A c T T t t
 5 a a G G G G A c T T C C
 6 T t G G G G A c T T C C
 7 T C G G G G A T T c a t
 8 T C G G G G A T T c C t
 9 T a G G G G A a c T a C
10 T C G G G t A T a a C C
```

Рис. 2.1 Десять потенциальных сайтов связывания NF-kB в геноме *Drosophila melanogaster*. Цветные буквы в верхнем регистре обозначают наиболее часто встречающийся нуклеотид в каждом столбце

Игра в прятки с мотивами

Наша цель состоит в том, чтобы превратить биологическую задачу поиска регуляторных мотивов в вычислительную задачу. Ниже мы имплантировали 15-мер скрытого сообщения в случайно выбранную позицию в каждой из десяти случайно сгенерированных строк ДНК. Этот пример имитирует сайт связывания фактора транскрипции, скрывающийся в восходящих областях десяти генов.

```
1 atgaccgggatactgataaaaaaaaggggggggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
2 acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaactttccgaataaaaaaaaggggggga
3 tgagtatccctgggatgacttaaaaaaaaggggggtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
4 gctgagaattggatgaaaaaaaaggggggtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
5 tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaataaaaaaaaggggggcttatag
```

```
 6 gtcaatcatgttcttgtgaatggatttaaaaaaaaggggggggaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
 7 cggttttggcccttgttagaggccccctgtaaaaaaaaggggggggcaattatgagagagctaatctatcgcgtgcgtgttcat
 8 aacttgagttaaaaaaaaggggggggctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
 9 ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcataaaaaaaaaggggggggaccgaaagggaag
10 ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttaaaaaaaaggggggga
```

ОСТАНОВИТЕСЬ и задумайтесь. Сможете ли вы найти имплантированное скрытое сообщение?

Это простая задача: применение алгоритма задачи часто встречающихся слов к конкатенации этих строк немедленно выявит наиболее часто встречающийся 15-мер, показанный ниже в виде имплантированной последовательности. Поскольку эти короткие строки были сгенерированы случайным образом, маловероятно, что они содержат другие часто встречающиеся 15-меры.

```
 1 atgaccgggatactgatAAAAAAAAGGGGGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
 2 acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaataAAAAAAAAGGGGGGGa
 3 tgagtatccctgggatgacttAAAAAAAAGGGGGGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
 4 gctgagaattggatgAAAAAAAAGGGGGGGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
 5 tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAAAAAAAAGGGGGGGcttatag
 6 gtcaatcatgttcttgtgaatggatttAAAAAAAAGGGGGGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
 7 cggttttggcccttgttagaggccccctAAAAAAAAGGGGGGGcaattatgagagagctaatctatcgcgtgcgtgttcat
 8 aacttgagttAAAAAAAAGGGGGGGctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
 9 ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatAAAAAAAAGGGGGGGaccgaaagggaag
10 ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttAAAAAAAAGGGGGGGa
```

Теперь представьте, что вместо того, чтобы имплантировать одну и ту же субстроку во все последовательности, мы подвергаем ее мутации, прежде чем вставлять в каждую последовательность, путем случайной замены нуклеотидов в четырех случайно выбранных позициях в каждом имплантированном 15-мере, как показано ниже.

```
 1 atgaccgggatactgatAgAAgAAAGGttGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
 2 acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaaacttttccgaatacAAtAAAAcGGcGGGa
 3 tgagtatccctgggatgacttAAAAtAAtGGaGtGGtgctctcccgatttttgaatatgtaggatcattcgccagggtccga
 4 gctgagaattggatgcAAAAAAAGGGattGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
 5 tccctttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAtAAtAAAGGaaGGGcttatag
 6 gtcaatcatgttcttgtgaatggatttAAcAAtAAGGGctGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
 7 cggttttggcccttgttagaggccccctAtAAAcAAGGaGGGccaattatgagagagctaatctatcgcgtgcgtgttcat
 8 aacttgagttAAAAAAtAGGGaGccctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
 9 ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatActAAAAAGGaGcGGaccgaaagggaag
10 ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttActAAAAAGGaGcGGa
```

Задача часто встречающихся слов нам не поможет, так как AAAAAAAAGGGGGGG даже не появляется в приведенных выше последовательностях. Возможно, тогда мы могли бы применить наше решение к задаче часто встречающихся слов с несовпадениями. Однако в главе 1 мы реализовали алгоритм для этой задачи, направленной на поиск скрытых сообщений с небольшим количеством несовпадений и небольшим размером *k*-меров (например, одно или два несовпадения для *DnaA*-боксов длиной 9). Этот алго-

ритм окажется слишком медленным при поиске имплантированного мотива из примера выше, поскольку он длиннее и имеет больше мутаций.

Кроме того, объединение всех последовательностей в одну строку неадекватно, поскольку оно неправильно моделирует биологическую задачу поиска мотива. *DnaA*-бокс представляет собой последовательность, которая сгруппирована или часто появляется в пределах относительно короткого интервала генома. Напротив, регуляторный мотив представляет собой последовательность, которая появляется по крайней мере один раз (возможно, с вариациями) в каждой из множества различных областей, разбросанных по всему геному.

Метод грубой силы поиска мотива

Для набора строк *Dna* и целого числа *d* *k-мер* является **(*k, d*)-мотивом**, если он встречается в каждой строке *Dna* не более чем с *d* несовпадениями. Например, имплантированный 15-мер в приведенных выше цепях представляет собой (15, 4)-мотив.

Задача поиска имплантированного мотива: *найдите все (k, d)-мотивы в наборе строк.*

Input: набор строк *Dna* и целых чисел *k* и *d*.

Output: все (*k, d*)-мотивы *Dna*.

Поиск методом грубой силы (также известный как **полный перебор**, или **исчерпывающий поиск**) – это общий метод решения задач, который исследует все возможные кандидаты в решения и проверяет, являются ли они решениями. Такие алгоритмы требуют минимальных усилий для разработки и гарантированно дают правильное решение, но они могут потребовать огромного количества времени работы, а количество кандидатов может быть слишком большим для проверки.

Метод грубой силы для решения задачи имплантированного мотива основан на наблюдении, что любой (*k, d*)-мотив должен иметь не более *d* несовпадений, кроме некоторого *k*-мера, появляющегося в первой цепи ДНК. Следовательно, мы можем сгенерировать все такие *k*-меры, а затем проверить, какие из них являются (*k, d*)-мотивами. Если вы забыли, как генерировать эти *k*-меры, вспомните: **ЗАРЯДНЫЕ СТАНЦИИ: Генерация окрестности строки**.

```
MotifEnumeration(Dna, k, d)
    Patterns ← пустой набор
    for каждого k-мера Pattern в первой строке Dna
        for каждого k-мера Pattern', отличающегося от Pattern как минимум d
        несовпадениями
```

```
        if Pattern' появляется в каждой строке Dna с как минимум d
        несовпадениями, добавьте Pattern' к Patterns
    удалите дубликаты из Patterns
    return Patterns
```

ОСТАНОВИТЕСЬ и задумайтесь. Оцените время работы **MotifEnumeration**.

MotifEnumeration, к сожалению, при больших значениях *k* и *d* работает довольно медленно, поэтому вместо этого мы попробуем применить другой подход. Может быть, мы сможем обнаружить имплантированную последовательность, просто идентифицируя два наиболее похожих *k*-мера между каждой парой строк *Dna*? Рассмотрим имплантированные 15-меры **AgAAgAAAGGttGGG** и **cAAtAAAAcGGGGcG**, каждый из которых отличается от **AAAAAAAAGGGGGGG** четырьмя несовпадениями. Хотя эти 15-меры похожи на правильный мотив **AAAAAAAAGGGGGGG**, они уже не так похожи друг на друга, так как имеют уже восемь несовпадений:

```
AgAAgAAAGGttGGG
||  ||    |  ||  |
cAAtAAAAcGGGGcG
```

Поскольку эти две имплантированные последовательности настолько различны, нас должно волновать, сможем ли мы найти их путем поиска наиболее похожих *k*-меров среди пар цепей в *Dna*.

В оставшейся части главы мы проверим наши алгоритмы поиска мотива, используя особенно сложный пример задачи с имплантированным мотивом. **Задача тонкого мотива** относится к имплантации 15-мера с четырьмя случайными мутациями в десять случайно сгенерированных строк длиной 600 нуклеотидов (типичная длина многих регуляторных областей выше по течению). Экземпляр задачи о тонком мотиве, который мы будем использовать, имеет имплантированный 15-мер **AAAAAAAAGGGGGGG**.

Оказывается, тысячи пар случайно встречающихся 15-меров в нашем наборе данных для задачи о тонком мотиве находятся на расстоянии менее 8 нуклеотидов друг от друга, что не позволяет нам идентифицировать истинные имплантированные мотивы путем попарных сравнений.

Считаем мотивы

От мотивов к матрицам профиля и консенсусным строкам

Хотя задача имплантированного мотива предлагает полезную абстракцию биологической задачи поиска мотива, она имеет некоторые ограничения.

Например, когда Стив Кей использовал массив ДНК для определения набора циркадных генов у растений, он не ожидал, что все гены в результирующем наборе будут иметь вечерний элемент (или его варианты) в своих восходящих областях. Точно так же биологи не ожидают, что все гены с повышенным уровнем экспрессии у инфицированных мух должны регулироваться NF-kB. Данные экспериментов с массивами ДНК по самой своей природе содержат большой шум, и некоторые гены, идентифицированные в этих экспериментах, не имеют ничего общего с суточными часами у растений или генами иммунитета у мух. Для таких зашумленных наборов данных любой алгоритм задачи имплантированного мотива потерпит неудачу, потому что если единственная последовательность не содержит сайта связывания фактора транскрипции, то (k, d)-мотива не существует!

Более подходящая формулировка задачи должна оценивать отдельные экземпляры мотивов в зависимости от того, насколько они похожи на «идеальный» мотив (т. е. сайт связывания транскрипционного фактора, который лучше всего связывается с транскрипционным фактором). Однако, поскольку идеальный мотив неизвестен, мы пытаемся выбрать k-меры из каждой строки и оценить их в зависимости от того, насколько они похожи *друг на друга*.

Чтобы определить результат, рассмотрим t цепей ДНК, каждая длиной n, и выберем k-меры из каждой цепи, чтобы сформировать набор мотивов *Motifs*, который мы представим в виде **матрицы мотивов** размерности $t \times k$. На рис. 2.2, где показана матрица мотивов для сайтов связывания NF-kB с рис. 2.1, мы указали наиболее часто встречающиеся нуклеотиды в каждом столбце матрицы мотивов заглавными буквами. Если в столбце есть несколько самых популярных нуклеотидов, то мы произвольно выбираем один из них, чтобы разорвать связь. Обратите внимание, что позиции 2 и 3 являются наиболее консервативными (нуклеотид **G** в этих позициях полностью консервативен), тогда как позиция 10 наименее консервативна.

Варьируя выбор k-меров в каждой цепи, мы можем построить большое количество различных матриц мотивов из данного образца цепей ДНК. Наша цель состоит в том, чтобы выбрать k-меры, дающие наиболее «консервативную» матрицу мотивов, т. е. матрицу с наибольшим количеством заглавных букв (и, следовательно, с наименьшим количеством строчных букв). Оставив в стороне вопрос о том, как мы выбираем такие k-меры, мы сначала сосредоточимся на том, как оценивать результирующие матрицы мотивов, определяя *Score*(*Motifs*) как количество непопулярных (строчных) букв в матрице мотивов *Motifs*. Наша цель – найти набор k-меров, который *минимизирует* этот показатель.

Упражнение. Минимально возможное значение *Score*(*Motifs*) равно 0 (если все строки в $t \times k$ матрице *Motifs* одинаковы). Каково максимально возможное значение *Score*(*Motifs*) в t и k?

Motifs

T	C	G	G	G	G	g	T	T	T	t	t
c	C	G	G	t	G	A	c	T	T	a	C
a	C	G	G	G	G	A	T	T	T	t	C
T	t	G	G	G	G	A	c	T	T	t	t
a	a	G	G	G	G	A	c	T	T	C	C
T	t	G	G	G	G	A	c	T	T	C	C
T	C	G	G	G	G	A	T	T	c	a	t
T	C	G	G	G	G	A	T	T	c	C	t
T	a	G	G	G	G	A	a	c	T	a	C
T	C	G	G	G	t	A	T	a	a	C	C

Score(*Motifs*) 3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = 30

Count(*Motifs*)

A:	2	2	0	0	0	0	9	1	1	1	3	0
C:	1	6	0	0	0	0	0	4	1	2	4	6
G:	0	0	10	10	9	9	1	0	0	0	0	0
T:	7	2	0	0	1	1	0	5	8	7	3	4

Profile(*Motifs*)

A:	.2	.2	0	0	0	0	.9	.1	.1	.1	.3	0
C:	.1	.6	0	0	0	0	0	.4	.1	.2	.4	.6
G:	0	0	1	1	.9	.9	.1	0	0	0	0	0
T:	.7	.2	0	0	.1	.1	0	.5	.8	.7	.3	.4

Consensus(*Motifs*) T C G G G G A T T T C C

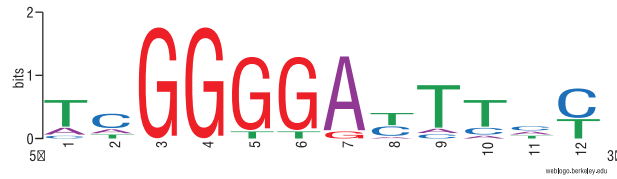


Рис. 2.2 От матрицы мотива к матрице счета, к матрице профиля, к согласованной строке и к логотипу мотива. Сайты связывания NF-kB образуют матрицу из 10×12 мотивов, причем наиболее часто встречающийся нуклеотид в каждом столбце показан прописными буквами, а все остальные нуклеотиды показаны строчными буквами. *Count*(*Motifs*) подсчитывает общее количество непопулярных (строчных) символов в матрице мотивов. Матрица мотивов дает в результате матрицу из 4×12 значений, содержащую счет нуклеотидов в каждом столбце матрицы мотивов; матрицу профиля, содержащую частоты нуклеотидов в каждом столбце матрицы мотива; и консенсусную строку, образованную наиболее часто встречающимся нуклеотидом в каждом столбце матрицы счета. Наконец, логотип мотива – это распространенный способ визуализации сохранения различных позиций в мотиве. Высота букв отображает информативность позиции

Мы можем построить **матрицу счета** $4 \times k$ *Count*(*Motifs*), подсчитывая количество вхождений каждого нуклеотида в каждом столбце матрицы мотивов; (*i, j*)-й элемент *Count*(*Motifs*) хранит количество раз, которое нуклеотид *i* появляется в столбце *j* мотивов. Далее мы разделим все элементы в матрице счета на *t*, количество строк в мотивах. Это приводит к **матрице профиля** *P* = *Profile*(*Motifs*), для которой $P_{i,j}$ представляет собой *частоту* *i*-го нуклеотида в *j*-м столбце матрицы мотивов. Обратите внимание, что элементы любого столбца матрицы профиля в сумме дают 1.

Наконец, мы формируем **консенсусную строку**, обозначенную *Consensus*(*Motifs*), из наиболее популярных нуклеотидов в каждом столбце матрицы мотивов (связи прерываются произвольно). Если мы правильно выберем *Motifs* из набора восходящих регионов, то *Consensus*(*Motifs*) предоставит идеального кандидата на регуляторный мотив для этих областей. Например, согласованной строкой для сайтов связывания NF-kB на рис. 2.2 является **TCGGGGATTTCC**.

На пути к более адекватной функции оценки мотивов

Рассмотрим второй столбец (содержащий 6 C, 2 A и 2 T) и последний столбец (содержащий 6 C и 4 T) в матрице мотивов на рис. 2.2. Оба этих столбца вносят 4 балла в *Score*(*Motifs*).

ОСТАНОВИТЕСЬ и задумайтесь. Имеет ли одинаковый биологический смысл сумма этих двух столбцов?

Для многих биологических мотивов в определенных позициях находятся два нуклеотида с примерно одинаковой способностью связываться с фактором транскрипции. Например, сайт связывания фактора транскрипции CSRE длиной 16 нуклеотидов у дрожжей *S. cerevisiae* состоит из пяти сильно консервативных позиций в дополнение к 11 слабо консервативным позициям, каждая из которых содержит два нуклеотида с одинаковыми частотами (рис. 2.3).

Следуя этому примеру, более подходящее представление консенсусной строки **TCGGGGATTTCC** для сайтов связывания NF-kB должно включать реальные альтернативы наиболее популярным нуклеотидам в каждом столбце (рис. 2.4). В этом смысле последний столбец (6 C, 4 T) в матрице мотивов на рис. 2.2 является «более консервативным», чем второй столбец (6 C, 2 A, 2 T), и должен иметь меньшую сумму.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	G/C	G/T	T/A	C/T	G/C	C/G	A	T	G/T	C/G	A	T	C/T	C/T	G/T

Рис. 2.3 Сайт связывания фактора транскрипции CSRE у *S. cerevisiae* имеет длину 16 нуклеотидов, но только пять из этих положений (1, 8, 9, 12, 13) сильно консервативны. Остальные 11 позиций могут иметь один из двух разных нуклеотидов

1	2	3	4	5	6	7	8	9	10	11	12
T	C	G	G	G	G	A	T/C	T	T	C	C/T

Рис. 2.4 Если мы рассмотрим нуклеотиды в каждом столбце матрицы мотивов сайтов связывания NF-kB с рис. 2.2 с частотой не менее 0,4, это даст представление сайтов связывания NF-kB с десятью позициями, представленными одним консенсусным нуклеотидом, и двумя позициями (8 и 12), представленными парой часто встречающихся нуклеотидов

Энтропия и motif logo

Каждый столбец *Profile*(*Motifs*) соответствует **распределению вероятностей** или набору неотрицательных чисел, сумма которых равна 1. Например, второй столбец на рис. 2.2 соответствует вероятностям 0,2, 0,6, 0,0 и 0,2 для A, C, G и T соответственно.

Profile(*Motifs*)												
A:	0,2	0,2	0	0	0	0	0,9	0,1	0,1	0,1	0,3	0
C:	0,1	0,6	0	0	0	0	0	0,4	0,1	0,2	0,4	0,6
G:	0	0	1	1	0,9	0,9	0,1	0	0	0	0	0
T:	0,7	0,2	0	0	0,1	0,1	0	0,5	0,8	0,7	0,3	0,4

Энтропия является мерой неопределенности распределения вероятностей $(p_1, \ldots, p_N)$ и определяется как

$$H(p_1, \ldots, p_N) = -\sum_{i=1}^{N} p_i \cdot \log_2 p_i.$$

Например, энтропия распределения вероятностей (0,2, 0,6, 0,0, 0,2), соответствующая второму столбцу матрицы профиля на рис. 2.2, равна

$$-(0{,}2 \cdot \log_2 0{,}2 + 0{,}6 \cdot \log_2 0{,}6 + 0{,}0 \cdot \log_2 0{,}0 + 0{,}2 \cdot \log_2 0{,}2) \approx 1{,}371,$$

тогда как энтропия более консервативного последнего столбца (0,0, 0,6, 0,0, 0,4) равна

$$-(0{,}0 \cdot \log_2 0{,}0 + 0{,}6 \cdot \log_2 0{,}6 + 0{,}0 \cdot \log_2 0{,}0 + 0{,}4 \cdot \log_2 0{,}4) \approx 0{,}971,$$

а энтропия очень консервативного 5-го столбца (0,0, 0,0, 0,9, 0,1) равна

$$-(0{,}0 \cdot \log_2 0{,}0 + 0{,}0 \cdot \log_2 0{,}0 + 0{,}9 \cdot \log_2 0{,}9 + 0{,}1 \cdot \log_2 0{,}1) \approx 0{,}467.$$

Обратите внимание, что технически $\log_2 0$ не определен, но при вычислении энтропии мы предполагаем, что $0 \cdot \log_2 0$ равно 0.

ОСТАНОВИТЕСЬ и задумайтесь. Каковы максимальное и минимальное возможные значения энтропии распределения вероятностей, содержащего четыре величины?

Энтропия полностью консервативного третьего столбца матрицы профиля на рис. 2.2 равна 0, что является минимально возможной энтропией. С другой стороны, столбец с равновероятными нуклеотидами (все вероятности равны 1/4) имеет максимально возможную энтропию $-4 \cdot 1/4 \cdot \log_2(1/4) = 2$. В общем, чем более консервативен столбец, тем меньше его энтропия. Таким образом, энтропия предлагает улучшенный метод оценки матриц мотивов: энтропия матрицы мотивов определяется как сумма энтропий ее столбцов. В этой книге для простоты мы продолжим использовать *Score*(*Motifs*), но на практике чаще используется величина энтропии.

Упражнение. Вычислите энтропию матрицы мотивов NF-kB (рис 2.2, вверху).

Другим применением энтропии является **motif logo (логотип мотива)**, диаграмма для визуализации сохранения мотива, состоящая из стопки букв для каждой позиции (см. нижнюю часть рис. 2.2). Относительные размеры букв указывают на их частоту в столбце. Относительная высота букв в каждом столбце основана на **информационном контенте** столбца, который определяется как $2 - H(p_1, \ldots, p_N)$. Чем ниже энтропия, тем выше информативность, а это означает, что высокие столбцы в мотиве логотипа более консервативны (рис. 2.2).

От поиска мотива к поиску медианной строки

Задача поиска мотива

Теперь, когда мы хорошо разобрались в оценке набора *k*-меров, мы готовы сформулировать задачу поиска мотива.

Задача поиска мотива: *имея набор строк, найдите набор k-меров, по одному из каждой строки, который минимизирует значение результирующего мотива.*

Input: набор строк *Dna* и целое число *k*.

Output: набор мотивов *k*-меров, по одному из каждой строки в *Dna*, сводящий к минимуму *Score*(*Motifs*) среди всех возможных вариантов *k*-меров.

«Алгоритм грубой силы» для задачи поиска мотива, **BruteForceMotifSearch**, рассматривает каждый возможный выбор мотивов *k*-меров из *Dna* (по одно-

му k-меру из каждой строки из n нуклеотидов) и выдает набор мотивов с минимальным значением. Поскольку в каждой из t последовательностей имеется $n - k + 1$ вариантов k-меров, существует $(n - k + 1)^t$ различных способов формирования мотивов. Для каждого выбора мотивов алгоритм вычисляет *Score*(*Motifs*), что требует $k \cdot t$ шагов. Таким образом, если предположить, что k намного меньше n, общее время работы алгоритма равно $O(n^t \cdot k \cdot t)$. Нам нужно придумать более быстрый алгоритм!

Переформулировка задачи поиска мотива

Поскольку **BruteForceMotifSearch** малоэффективен, попробуем искать мотивы по-другому. Вместо того чтобы исследовать все *Motifs* в *Dna* и впоследствии выводить консенсусную строку из мотивов,

$$Motifs \rightarrow Consensus(Motifs),$$

сначала изучим все потенциальные консенсусные строки k-меров, а затем сформируем наилучший набор возможных мотивов для каждой консенсусной строки,

$$Consensus(Motifs) \rightarrow Motifs.$$

Чтобы переформулировать задачу поиска мотива, нам нужно разработать альтернативный способ вычисления *Score*(*Motifs*). До сих пор мы вычисляли *Score*(*Motifs*), количество строчных букв в матрице мотивов, столбец за столбцом. Например, на рис. 2.2 мы вычислили *Score*(*Motifs*) для матрицы мотивов NF-kB как

$$3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = 30.$$

На рис. 2.5 показано, что *Score*(*Motifs*) можно так же легко вычислить построчно, как

$$3 + 4 + 2 + 4 + 3 + 2 + 3 + 2 + 4 + 3 = 30.$$

Обратите внимание, что каждый элемент в последней сумме представляет собой количество несовпадений между консенсусной строкой **TCGGGGATTTCC** и мотивом в соответствующей строке матрицы мотивов, т. е. расстояние Хэмминга между этими строками. Для первой строки матрицы мотивов на рис. 2.5 d(**TCGGGGATTTCC**, **TCGGGG**g**TTT**tt) = **3**.

Имея набор k-меров *Motifs* = {$Motif_1$, … , $Motif_t$} и k-мера *Pattern*, мы теперь определяем d(*Pattern*, *Motifs*) как сумму расстояний Хэмминга между *Pattern* и каждым $Motif_i$,

$$d(Pattern, Motifs) = \sum_{i=1}^{t} HammingDistance(Pattern, Motif_i).$$

	T	C	G	G	G	G	g	T	T	T	t	t	3
	c	C	G	G	t	G	A	c	T	T	a	C	4
	a	C	G	G	G	G	A	T	T	T	t	C	2
	T	t	G	G	G	G	A	c	T	T	t	t	4
Motifs	a	a	G	G	G	G	A	c	T	T	C	C	3
	T	t	G	G	G	G	A	c	T	T	C	C	2
	T	C	G	G	G	G	A	T	T	c	a	t	3
	T	C	G	G	G	G	A	T	T	c	C	t	2
	T	a	G	G	G	G	A	a	c	T	a	C	4
	T	C	G	G	G	t	A	T	a	a	C	C	+3
Score(*Motifs*)	3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = **30**												
Consensus(*Motifs*)	T	C	G	G	G	G	A	T	T	T	C	C	

Рис. 2.5 Матрицы мотивов и их счет в дополнение к консенсусной строке для сайтов связывания NF-kB, воспроизведенные с рис. 2.2. Вместо того чтобы подсчитывать неконсенсусные элементы (т. е. нуклеотиды нижнего регистра) столбец за столбцом, мы можем суммировать их ряд за рядом, как показано справа от матрицы мотивов. Каждое значение в конце строки соответствует расстоянию Хэмминга между этой строкой и строкой консенсуса

Поскольку *Score*(*Motifs*) соответствует сумме элементов нижнего регистра *Motifs* столбец за столбцом, а *d*(*Consensus*(*Motifs*), *Motifs*) соответствует сумме этих элементов строка за строкой, мы получаем, что

$$Score(Motifs) = d(Consensus(Motifs), Motifs).$$

Это уравнение дает нам прекрасную идею. Вместо того чтобы искать набор *k*-меров *Motifs*, минимизирующих

$$Score(Motifs),$$

давайте будем искать потенциальную консенсусную строку *Pattern*, минимизирующую

$$d(Pattern, Motifs)$$

среди всех возможных *k*-меров *Pattern* и всех возможных вариантов *k*-меров *Motifs* в *Dna*. Эта задача эквивалентна задаче поиска мотива.

Эквивалентная задача поиска мотива: *для заданного набора строк найдите Pattern и набор k-меров (по одному из каждой строки), которые минимизируют расстояние между всеми возможными Pattern и всеми возможными наборами k-меров.*

Input: набор строк *Dna* и целое число *k*.

Output: *k*-мер *Pattern* и набор *k*-меров *Motifs*, по одному из каждой строки *Dna*, минимизирующих *d*(*Pattern, Motifs*) среди всех возможных вариантов *Pattern* и *Motifs*.

Задача поиска медианной строки

Но подождите секунду – разве мы не усложнили себе задачу? Вместо того чтобы искать все *Motifs*, теперь нам нужно искать не только все *Motifs*, но также все *k*-меры *Pattern*. Ключевой момент для решения эквивалентной задачи поиска мотива заключается в том, что при заданном *Pattern* нам не нужно исследовать все возможные наборы *Motifs*, чтобы минимизировать *d*(*Pattern*, *Motifs*).

Чтобы объяснить, как это можно сделать, мы определяем *Motifs* (*Pattern, Dna*) как набор *k*-меров, который минимизирует *d*(*Pattern*, *Motifs*) для данного *Pattern* и всех возможных наборов *k*-меров *Motifs* в *Dna*. Например, для строк *Dna*, показанных ниже, пять цветных 3-меров представляют *Motifs*(**AAA**, *Dna*).

```
          ttaccttAAC
          gATAtctgtc
Dna       ACGgcgttcg
          ccctAAAgag
          cgtcAGAggt
```

ОСТАНОВИТЕСЬ и задумайтесь. Имея набор строк *Dna* и *k*-mer *Pattern*, разработайте быстрый алгоритм для создания *Motifs*(*Pattern, Dna*).

Причина, по которой нам не нужно рассматривать все возможные наборы мотивов *Motifs* в *Dna* = {Dna_1, … , Dna_t}, состоит в том, что мы можем генерировать *k*-меры в *Motifs*(*Pattern, Dna*) по одному; т. е. мы можем выбрать *k*-мер в Dna_i независимо от выбора *k*-меров во всех других строках *Dna*. Имея *k*-мер *Pattern* и более длинную строку *Text*, мы используем *d*(*Pattern, Text*) для обозначения минимального расстояния Хэмминга между *Pattern* и любым *k*-мером в *Text*,

$$d(Pattern, Text) = \min_{\text{all } k\text{-mers } Pattern' \text{ in } Text} HammingDistance(Pattern, Pattern').$$

Например,

$$d(\text{GATTCTCA}, \text{gcaaaGACGCTGAccaa}) = 3.$$

K-мер в *Text*, который достигает минимального расстояния Хэмминга с *Pattern*, обозначается *Motif*(*Pattern, Text*). Для приведенного выше примера

Motif(**GATTCTCA**, gcaaa**GACGCTGA**ccaa) = **GACGCTGA**.

Отметим, что обозначение *Motis*(*Pattern*, *Text*) неоднозначно, потому что в *Text* может быть несколько *k*-меров, которые достигают минимального расстояния Хэмминга с *Pattern*. Например, *Motif*(**AAG**, gc**AAT**cct**CAG**c) может быть либо **AAT**, либо **CAG**. Если в тексте есть несколько *k*-меров, имеющих минимальное расстояние Хэмминга с *Pattern*, мы выбираем первый такой *k*-мер в тексте как *Motif*(*Pattern*, *Text*).

Имея *k*-mer *Pattern* и набор строк *Dna* = {Dna_1, ..., Dna_t}, мы определяем *d*(*Pattern, Dna*) как сумму расстояний между *Pattern* и всеми строками в *Dna*,

$$d(Pattern, Dna) = \sum_{i=1}^{t} d(Pattern, Dna_i).$$

Например, для строк *Dna*, показанных ниже, *d*(**AAA**, *Dna*) = **1** + **1** + **2** + **0** + **1** = **5**.

	ttaccttAAC	1
	gATAtctgtc	1
Dna	ACGgcgttcg	2
	ccctAAAgag	0
	cgtcAGAggt	1

Наша цель состоит в том, чтобы найти *k*-мер *Pattern*, который минимизирует *d*(*Pattern, Dna*) по всем *k*-мерам *Pattern*, та же задача, которую пытается решить задача поиска эквивалентного мотива. Такой *k*-мер мы называем **медианной строкой** (Median String) для *Dna*.

Задача поиска медианной строки: *найдите медианную строку.*

Input: набор строк *Dna* и целое число *k*.

Output: *k*-мер *Pattern*, минимизирующий *d*(*Pattern, Dna*) среди всех возможных вариантов *k*-меров.

Обратите внимание, что для нахождения медианной строки требуется решить задачу двойной минимизации. Мы должны найти *k*-мер *Pattern*, который минимизирует *d*(*Pattern, Dna*), где эта функция вычисляет себя сама путем взятия минимума по всем вариантам *k*-меров из каждой строки в *Dna*. Псевдокод алгоритма грубой силы **Median String** приведен ниже.

```
Median String(Dna, k)
    distance ← ∞
    for каждого k-мера Pattern от AA...AA до TT...TT
        if distance > d(Pattern, Dna)
            distance ← d(Pattern, Dna)
```

```
        Median ← Pattern
    return Median
```

ЗАРЯДНАЯ СТАНЦИЯ (Решение задачи средней строки). Хотя этот псевдокод короткий, он не лишен потенциальных ловушек. Изучите эту зарядную станцию, если попадете в одну из них.

ОСТАНОВИТЕСЬ и задумайтесь. Вместо того чтобы тратить время на поиск всех возможных k-меров в **Median String**, можете ли вы просмотреть только все те k-меры, которые есть в *Dna*?

Почему мы переформулировали задачу поиска мотива?

Чтобы понять, почему мы переформулировали задачу поиска мотива как эквивалентную задачу поиска медианной строки, рассмотрим время выполнения **Median String** и **BruteForceMotifs**. Предыдущий алгоритм вычисляет $d(Pattern, Dna)$ для каждого из 4^k k-меров *Pattern*. Каждое вычисление $d(Pattern, Dna)$ требует одного прохода по каждой строке *Dna*, что требует приблизительно $k \cdot n \cdot t$ операций для строк t длины n в *Dna*. Таким образом, **Median String** имеет время выполнения $O(4^k \cdot n \cdot k \cdot t)$, что на практике выгодно отличается от времени выполнения $O(n^t \cdot k \cdot t)$ метода **BruteForceMotifSearch**, так как длина мотива (k) обычно не превышает 20 нуклеотидов, а длина t измеряется тысячами.

Задача поиска медианной строки преподает нам важный урок: иногда переформулирование задачи может привести к значительному уменьшению времени, необходимого для ее выполнения. В этом случае очевидно умозаключение о том, что *Score*(*Motifs*) может быть так же легко вычислен по строкам, как и по столбцам, что дает нам более быстрый алгоритм **Median String.**

Конечно, окончательной проверкой алгоритма биоинформатики является то, как он работает на практике. К сожалению, поскольку **Median String** должен учитывать 4^k k-меров, он становится слишком медленным для задачи о тонком мотиве, для которой $k = 15$. Давайте запустим **Median String** с $k = 13$ в надежде, что он определит подстроку правильного 15-мера мотива. Алгоритму попрежнему потребуется полдня для выполнения процедуры на нашем компьютере, и он возвратит медианную строку **AAAAAtAGaGGGG** (с расстоянием 29). Этот 13-мер не является подстрокой имплантированной последовательности **AAAAAAAAGGGGGGG**, но приближается к ней.

ОСТАНОВИТЕСЬ и задумайтесь. Как немного неточная медианная строка длины 13 может помочь нам найти правильную медианную строку длиной 15?

До сих пор мы предполагали, что значение *k* известно заранее, что на практике не так. В результате мы вынуждены запускать наши алгоритмы поиска мотива для разных значений *k*, а затем пытаться получить правильную длину мотива. Поскольку некоторые регуляторные мотивы довольно длинные – позже в этой главе мы будем искать биологически важный мотив длиной 20, а **Median String** может оказаться слишком медленным, чтобы найти их.

Жадный алгоритм поиска мотива

Использование матрицы профиля для бросания костей

Многие алгоритмы представляют собой итерационные процедуры, которые должны выбирать из множества альтернатив на каждой итерации. Некоторые из этих альтернатив могут привести к правильным решениям, а другие – нет. **Жадные алгоритмы** выбирают «наиболее привлекательную» альтернативу на каждой итерации. Например, жадный алгоритм в шахматах будет пытаться побить самую ценную фигуру противника на каждом ходу. Тем не менее любой, кто играл в шахматы, знает, что эта стратегия взгляда только на один ход вперед, скорее всего, приведет к катастрофическим результатам. Как правило, большинство жадных алгоритмов не могут найти точное решение задачи; вместо этого они часто представляют собой быстрые **эвристики**, которые обменивают точность на скорость, чтобы найти приблизительное решение. Тем не менее для многих биологических задач, которые мы будем изучать в этой книге, жадные алгоритмы окажутся весьма полезными.

В этом разделе мы рассмотрим жадный метод поиска мотивов. Опять же, пусть *Motifs* будет набором *k*-меров, взятых из *t* строк *Dna*. Вспомните из предыдущего обсуждения энтропии, что мы можем рассматривать каждый столбец *Profile*(*Motifs*) как четырехгранный кубик. Таким образом, матрицу профиля с *k* столбцами можно рассматривать как набор *k* игральных костей, которые мы будем бросать, чтобы случайным образом сгенерировать *k*-меры. Например, если первый столбец матрицы профиля равен (**0,2**, **0,1**, **0,0**, **0,7**), то мы генерируем **A** как первый нуклеотид с вероятностью **0,2**, **C** с вероятностью **0,1**, **G** с вероятностью **0,0** и **T** с вероятностью **0,7**.

На рис. 2.6 мы воспроизводим матрицу профиля для сайтов связывания NF-kB с рис. 2.2, где одиночный окрашенный вход в *i*-м столбце соответствует *i*-му нуклеотиду в **ACGGGGATTACC**. Вероятность *Pr*(**ACGGGGATTACC**|*Profile*) того, что *Profile* генерирует **ACGGGGATTACC**, вычисляется простым умножением выделенных элементов в матрице профиля.

K-мер, как правило, имеет более высокую вероятность, когда он больше похож на консенсусную строку профиля. Например, для матрицы профиля NF-kB, показанной на рис. 2.6, и ее консенсусной строки **TCGGGGATTTCC**

$$Pr(\mathbf{TCGGGGATTTCC}|Profile) = \mathbf{0,7}\cdot\mathbf{0,6}\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{0,9}\cdot\mathbf{0,9}\cdot\mathbf{0,9}\cdot\mathbf{0,5}\cdot\mathbf{0,8}\cdot\mathbf{0,7}\cdot\mathbf{0,4}\cdot\mathbf{0,6}$$
$$= 0{,}0205753$$

что больше значения *Pr*(ACGGGGATTACC|*Profile*) = 0,000839808, которое мы вычислили на рис. 2.6.

Profile	A:	0,2	0,2	0	0	0	0	0,9	1	1	0,1	0,3	0
	C:	,1	0,6	0	0	0	0	0	0,4	0,1	0,2	0,4	0,6
	G:	0	0	1	1	0,9	0,9	0,1	0	0	0	0	0
	T:	0,7	0,2	0	0	0,1	0,1	0	0,5	0,8	0,7	0,3	0,4
Pr(ACGGGGATTACC|*Profile*) =		0,2	· 0,6	· 1	· 1	· 0,9	· 0,9	· 0,9	· 0,5	· 0,8	· 0,1	· 0,4	· 0,6	= 0,000839808

Рис. 2.6 Мы можем сгенерировать случайную по матрице профиля, выбрав *i*-й нуклеотид в строке с вероятностью, соответствующей этому нуклеотиду в *i*-м столбце матрицы профиля. Вероятность того, что матрица профиля даст заданную строку, определяется как произведение вероятностей отдельных нуклеотидов

Упражнение: Вычислить *Pr*((TCGTGGATTTCC|*Profile*), где *Profile* – это матрица, представленная на рис. 2.6.

Имея матрицу профиля *Profile*, мы можем оценить вероятность каждого *k*-мера в строке *Text* и найти **наиболее вероятный *k*-мер с *Profile*** в *Text*, т. е. *k*-мер, который с наибольшей вероятностью был сгенерирован *Profile* среди все *k*-меров в *Text*. Для матрицы профиля NF-kB ACGGGGATTACC является наиболее вероятным 12-мером *Profile* в ggtACGGGGATTACCt. Действительно, каждый второй 12-мер в этой строке имеет вероятность 0. В общем, если в *Text* есть несколько наиболее вероятных *k*-меров с *Profile*, то мы выбираем первый такой *k*-мер, встречающийся в *Text*.

Задача поиска наиболее вероятного *k*-мера с *Profile*: *найдите наиболее вероятный k-мер в строке, содержащий Profile.*

Input: строка *Text*, целое число *k* и матрица *Profile* размерностью 4×*k*.

Output: наиболее вероятный *k*-мер в тексте, содержащий *Profile*.

Предложенный нами жадный алгоритм поиска мотивов **GreedyMotifSearch** начинается с формирования матрицы мотивов из произвольно выбранных *k*-меров в каждой строке *Dna* (которая в нашей конкретной реализации является первым *k*-мером в каждой строке). Затем он пытается улучшить эту исходную матрицу мотивов, пробуя каждый из *k*-меров в Dna_1 в качестве первого мотива. Для заданного выбора *k*-мера $Motif_1$ в Dna_1 он строит матрицу профиля *Profile* для этого одиночного *k*-мера и устанавливает $Motif_2$ равным наиболее вероятному *k*-меру с *Profile* в Dna_2. Затем он выполняет итерацию, обновляя *Profile* как матрицу профиля, сформированную из $Motif_1$ и $Motif_2$, и устанавливает $Motif_3$ равным вероятному *k*-меру с *Profile* в Dna_3. В общем, после нахождения $i - 1$ *k*-мера *Motifs* в первых $i - 1$ строках *Dna* алгоритм **GreedyMotifSearch** строит

Profile(*Motifs*) и выбирает наиболее вероятный *k*-мер с *Profile* из Dna_i по этой матрице профиля. После получения *k*-мера из каждой строки для создания набора *Motifs* **GreedyMotifSearch** проверяет, превосходит ли *Motifs* текущую коллекцию мотивов с наивысшими суммами, а затем перемещает $Motif_1$ на один символ в Dna_1, начиная весь процесс создания *Motifs* заново.

```
GreedyMotifSearch(Dna, k, t)
    BestMotifs ← матрица мотивов, сформированная первыми k-мерами в каждой
    строке Dna
    for каждого k-мера Motif первой строки Dna
        Motif1 ←Motif
        for i = 2 до t
            сформируйте Profile из мотивов Motif_1, ..., Motif_{i-1}
            Motif_i ← наиболее вероятный k-мер Profile в i-й строке Dna
        Motifs ← (Motif_1, ..., Motif_t)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    return BestMotifs
```

Если вас не устраивает производительность **GreedyMotifSearch** – даже если вы правильно его реализовали, – подождите, пока мы не обсудим этот алгоритм в следующем разделе.

Анализ жадного алгоритма поиска мотива

В отличие от **Median String**, **GreedyMotifSearch** работает быстро и может быть запущен с $k = 15$ для решения задачи тонкого мотива (напомним, что мы остановились на $k = 13$ в случае **Median String**). Однако он меняет скорость на точность и выдает 15-мерный **gtAAAtAgaGatGtG** (общее расстояние: 58), который сильно отличается от истинного имплантированного мотива **AAAAAAAAGGGGGGG**.

ОСТАНОВИТЕСЬ и задумайтесь. Почему **GreedyMotifSearch** работает так плохо?

На первый взгляд **GreedyMotifSearch** может показаться весьма разумным алгоритмом, но это не так! Посмотрим, найдет ли **GreedyMotifSearch**(4, 1)-мотив **ACGT**, внедренный в следующие строки *Dna*:

```
ttACCTtaac
gATGTctgtc
acgGCGTtag
ccctaACGAg
cgtcagAGGT
```

Предположим оптимистичный сценарий, в котором алгоритм уже правильно выбрал имплантированный 4-мер **ACCT** из первой строки *Dna* и построил соответствующий *Profile*:

A:	**1**	0	0	0
C:	0	**1**	**1**	0
G:	0	0	0	0
T:	0	0	0	**1**

Алгоритм теперь готов к поиску наиболее вероятного 4-мера *Profile* во второй последовательности. Проблема, однако, в том, что в матрице профиля так много нулей, что вероятность каждого 4-мера, кроме **ACCT**, равна нулю! Таким образом, если **ACCT** не присутствует в каждой строке *Dna*, маловероятно, что **GreedyMotifSearch** найдет внедренный мотив. Нули в матрице профиля – это не просто мелкая неприятность, а скорее постоянная проблема, которую мы должны решить.

Поиск мотива и Оливер Кромвель

Какова вероятность того, что завтра не взойдет солнце?

В 1650 году, после того как шотландцы провозгласили Карла II королем во время Гражданской войны в Англии, Оливер Кромвель обратился к шотландской церкви со знаменитым обращением. Убедив их увидеть ошибку их королевского союза, он умолял:

> *«Умоляю вас, именем Христа, подумайте о том, что вы можете ошибаться».*

Шотландцы отклонили апелляцию, и Кромвель в ответ вторгся в Шотландию. Позже его цитата вдохновила статистическую максиму, называемую **правилом Кромвеля**, которая гласит, что мы не должны использовать вероятности 0 или 1, если только мы не говорим о логических утверждениях, которые могут быть только истинными или ложными. Другими словами, мы должны допустить небольшую вероятность крайне маловероятных событий, таких как «эту книгу написали инопланетяне» или «солнце не взойдет завтра». Мы не можем говорить о вероятности первого события, но в XVIII веке французский математик Пьер-Симон Лаплас действительно оценил вероятность того, что солнце не взойдет завтра (как 1/1826251), учитывая, что оно вставало каждый день в течение последних 5000 лет. Хотя эта оценка была высмеяна его современниками, подход Лапласа к этому вопросу сейчас играет важную роль в статистике.

В любом наблюдаемом наборе данных существует вероятность, особенно для событий с низкой вероятностью или небольших наборов данных, что событие с ненулевой вероятностью не произойдет. Следовательно, его наблю-

даемая частота равна нулю; однако, установление эмпирической вероятности события равной нулю, представляет собой неверное упрощение, которое может вызвать проблемы. Искусственно регулируя вероятность редких событий, эти проблемы можно смягчить.

Правило преемственности Лапласа

Правило Кромвеля относится к вычислению вероятности строки по матрице профиля. Например, рассмотрим следующий *Profile*:

Profile	A:	0,2	0,2	0	0	0	0	**0,9**	0,1	0,1	0,1	0,3	0
	C:	0,1	**0,6**	0	0	0	0	0	0,4	0,1	0,2	**0,4**	**0,6**
	G:	0	0	**1**	1	**0,9**	**0,9**	0,1	0	0	0	0	0
	T:	**0,7**	0,2	0	**0**	0,1	0,1	0	**0,5**	**0,8**	**0,7**	0,3	0,4

Pr(TCGTGGATTTCC|*Profile*) = **0,7** · **0,6** · **1** · **0** · **0,9** · **0,9** · **0,9** · **0,5** · **0,8** · **0,7** · **0,4** · **0,6** = 0

Четвертый символ TCGTGGATTTCC приводит к тому, что *Pr*(TCGTGGATTTCC|*Profile*) становится равным нулю. В результате всей строке присваивается нулевая вероятность, хотя TCGTGGATTTCC отличается от согласованной строки только в одной позиции. Если уж на то пошло, TCGTGGATTTCC имеет такую же низкую вероятность, как и AAATCTTGGAA, которая отличается от консенсусной строки в каждой позиции.

Чтобы улучшить эту неточную оценку, биоинформатики часто заменяют нули небольшими числами, называемыми **псевдосчетами**. Простейший подход к введению псевдосчетов, называемый **правилом преемственности Лапласа**, похож на принцип, который Лаплас использовал для расчета вероятности того, что солнце не взойдет завтра. В случае мотивов псевдосчет часто сводится к добавлению 1 (или другого небольшого числа) к каждому элементу *Count*(*Motifs*). Предположим, что у нас есть следующие матрицы – мотива, счета и профиля:

Motifs

T	A	A	C
G	T	C	T
A	C	T	A
A	G	G	T

Count(*Motifs*)

A:	2	1	1	1
C:	0	1	1	1
G:	1	1	1	0
T:	1	1	1	2

Profile(*Motifs*)

2/4	1/4	1/4	1/4
0	1/4	1/4	1/4
1/4	1/4	1/4	0
1/4	1/4	1/4	2/4

Правило последовательности Лапласа добавляет 1 к каждому элементу *Count*(*Motifs*), обновляя две матрицы до следующего:

Count(*Motifs*)					*Profile*(*Motifs*)				
A:	2+1	1+1	1+1	1+1		3/8	2/8	2/8	2/8
C:	0+1	1+1	1+1	1+1		1/8	2/8	2/8	2/8
G:	1+1	1+1	1+1	0+1		2/8	2/8	2/8	1/8
T:	1+1	1+1	1+1	2+1		2/8	2/8	2/8	3/8

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы использовали правило преемственности Лапласа, чтобы устранить недостатки **GreedyMotifSearch**?

Улучшенный алгоритм жадного поиска мотивов

Единственное изменение, которое нам нужно внести в **GreedyMotifSearch**, чтобы исключить нули из матриц профилей, которые он создает, – это заменить строку 6 псевдокода в **GreedyMotifSearch** (показано зеленым):

```
GreedyMotifSearch(Dna, k, t)
    Сформировать набор k-меров BestMotifs путем отбора 1первых k-меров из
    каждой строки Dna
    for каждого k-мера Motif в первой строке Dna
        Motif_1 ← Motif
        for i = 2 до t
            сформировать Profile из мотивов Motif_1, ..., Motif_{i-1}
            Motif_i ← наиболее вероятный k-мер в Profile в i-й строке Dna
        Motifs ← (Motif_1, ..., Motif_t)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    output BestMotifs
```

После включения правила наследования Лапласа **GreedyMotifSearch** определяется следующим образом:

```
GreedyMotifSearch(Dna, k, t)
    Сформировать набор k-меров BestMotifs путем отбора первых k-меров из
    каждой строки Dna
    for каждого k-мера Motif в первой строке Dna
        Motif_1 ← Motif
        for i = 2 до t
            применить правило преемственности Лапласа, чтобы сформировать
            Profile из мотивов Motif_1, ..., Motif_{i-1}
            Motif_i ← наиболее вероятный k-мер в Profile в i-й строке Dna
```

```
    Motifs ← (Motif_1, ..., Motif_t)
    if Score(Motifs) < Score(BestMotifs)
      BestMotifs ← Motifs
  output BestMotifs
```

Теперь мы применим правило преемственности Лапласа для поиска (4, 1)-мотива **ACCT**, внедренного в следующие строки *Dna*:

```
        ttACCTtaac
        gATGTctgtc
Dna     acgGCGTtag
        ccctaACGAg
        cgtcagAGGT
```

Снова предположим, что алгоритм уже выбрал имплантированный 4-мер **ACCT** из первой последовательности. Мы можем построить соответствующие матрицы счета и профиля, используя правило преемственности Лапласа:

Motifs **ACCT**

Count(*Motifs*)

A:	1+1	0+1	0+1	0+1
C:	0+1	1+1	1+1	0+1
G:	0+1	0+1	0+1	0+1
T:	0+1	0+1	0+1	1+1

Profile(*Motifs*)

2/5	1/5	1/5	1/5
1/5	2/5	2/5	1/5
1/5	1/5	1/5	1/5
1/5	1/5	1/5	2/5

Мы используем эту матрицу профиля для вычисления вероятностей всех 4-меров во второй строке *Dna*:

gATG	ATGT	TGTc	GTct	Tctg	ctgt	tgtc
$1/5^4$	$4/5^4$	$1/5^4$	$4/5^4$	$2/5^4$	$2/5^4$	$1/5^4$

Во второй строке имеется два наиболее вероятных 4-мера с профилем (**ATGT** and **GT**ct); давайте предположим, что нам снова повезет, и выберем имплантированный 4-мер **ATGT**. Теперь у нас есть следующие матрицы мотивов, счета и профиля:

Motifs **ACCT**
ATGT

Count(*Motifs*)

A:	2+1	0+1	0+1	0+1
C:	0+1	1+1	1+1	0+1
G:	0+1	0+1	1+1	0+1
T:	0+1	1+1	0+1	2+1

Profile(*Motifs*)

3/6	1/6	1/6	1/6
1/6	2/6	2/6	1/6
1/6	1/6	2/6	1/6
1/6	2/6	1/6	3/6

Мы используем эту матрицу профиля для вычисления вероятностей всех 4-меров в третьей строке *Dna*:

acgG	cgGC	gGCG	GCGT	CGTt	GTta	Ttag
$12/6^4$	$2/6^4$	$2/6^4$	$12/6^4$	$3/6^4$	$2/6^4$	$2/6^4$

Опять же, во второй строке есть два наиболее вероятных 4-мера с профилем (acgG и GCGT). На этот раз предположим, что вместо GCGT имплантирован 4-мер acgG. Теперь у нас есть следующие матрицы мотивов, счета и профиля:

Motifs

ACCT
ATGT
acgG

Count(*Motifs*)

A:	3+1	0+1	0+1	1+1
C:	0+1	2+1	1+1	0+1
G:	0+1	0+1	2+1	1+1
T:	0+1	1+1	0+1	2+1

Profile(*Motifs*)

4/7	1/7	1/7	1/7
1/7	3/7	2/7	1/7
1/7	1/7	3/7	2/7
1/7	2/7	1/7	3/7

Мы используем эту матрицу профиля для вычисления вероятностей всех 4-меров в четвертой строке *Dna*:

ccct	ccta	ctaA	taAC	aACG	ACGA	CGAg
$18/7^4$	$3/7^4$	$2/7^4$	$1/7^4$	$16/7^4$	$36/7^4$	$2/7^4$

Несмотря на то что мы пропустили имплантированный 4-мер в третьей последовательности, мы теперь нашли имплантированный 4-мер в четвертой последовательности в *Dna* как наиболее вероятный 4-мер *Profile* ACGA. Это дает нам следующие матрицы мотивов, счета и профиля:

Motifs

ACCT
ATGT
acgG
ACGA

Count(*Motifs*)

A:	4+1	0+1	0+1	0+1
C:	0+1	3+1	1+1	0+1
G:	0+1	0+1	3+1	1+1
T:	0+1	1+1	0+1	2+1

Profile(*Motifs*)

5/8	1/8	1/8	2/8
1/8	4/8	2/8	1/8
1/8	1/8	4/8	2/8
1/8	2/8	1/8	3/8

Теперь мы используем этот профиль для вычисления вероятностей всех 4-меров в пятой строке в *Dna*:

cgtc	gtca	tcag	cagA	agAG	gAGG	AGGT
$1/7^4$	$8/8^4$	$8/8^4$	$8/8^4$	$10/8^4$	$8/8^4$	$60/8^4$

Наиболее вероятный 4-мер с *Profile* пятой строки *Dna* – это **AGGT**, имплантированный 4-мер. В результате **GreedyMotifSearch** создал следующую матрицу мотивов, которая подразумевает правильную согласованную строку **ACGT**:

Motifs	**ACCT** **ATGT** acg**G** **ACGA** **AGGT**
Consensus(*Motifs*)	**ACGT**

Теперь вы увидели силу псевдосчетов, проиллюстрированных на небольшом примере. Запуск **GreedyMotifSearch** с псевдосчетами для решения задачи о тонком мотиве выдает набор 15-мерных мотивов с *Score*(*Motifs*) = 41 и *Consensus*(*Motifs*) = **AAAAAtAgaGGGGtt**. Таким образом, правило наследования Лапласа дало значительное улучшение по сравнению с исходным **GreedyMotifSearch**, который выдавал согласованную строку **gTtAAAtAgaGatGtG** со *Score*(Motifs) = 58.

Вы можете быть довольны работой **GreedyMotifSearch**, но вы уже должны знать, что ваши авторы никогда не бывают удовлетворены до конца. Можем ли мы разработать еще более точный алгоритм поиска мотивов?

Рандомизированный поиск мотива

Игра в кости для поиска мотивов

Теперь мы обратимся к **рандомизированным алгоритмам**, которые подбрасывают монеты и кидают кости для поиска мотивов. Принятие случайных алгоритмических решений может показаться катастрофической идеей – просто представьте себе шахматную партию, в которой каждый ход будет решаться броском кости. Однако французский математик и естествоиспытатель XVIII века граф де Бюффон впервые доказал, что рандомизированные алгоритмы могут быть полезны. Он сделал это, случайным образом бросая иглы на параллельные деревянные планки и используя результаты этого эксперимента для более точного вычисления константы π (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Игла Бюффона**).

Рандомизированные алгоритмы могут быть неинтуитивными, потому что им не хватает контроля над традиционными алгоритмами. Часть рандомизированных алгоритмов – **алгоритмы Лас-Вегаса**, обеспечивают гарантированно точные решения задач, несмотря на то, что они основаны на принятии случайных решений в процессе вычисления. Однако большинство рандомизированных алгоритмов, включая алгоритмы поиска мотивов, которые мы рассмотрим в этой главе, являются **алгоритмами Монте-Карло**. Эти алгоритмы не гарантируют точного решения, но они быстро находят *приближенные* реше-

ния. Из-за скорости выполнения их можно запускать много раз, что позволяет нам выбрать наилучшее приближение из тысяч запусков.

Ранее мы определили *Profile*(*Motifs*) как матрицу профиля, построенную из набора *k*-меров *Motifs* в строке *Dna*. Теперь, имея набор строк *Dna* и произвольный *Profile*, в виде матрицы размерностью 4×*k*, мы определяем *Motifs*(*Profile*, *Dna*) как набор *k*-меров, образованных *k*-мерами с наибольшей вероятность содержания *Profile* в каждой последовательности *Dna*. Например, рассмотрим следующий *Profile* и *Dna*:

Profile

A:	4/5	0	0	1/5
C:	0	3/5	1/5	0
G:	1/5	1/5	4/5	0
T:	0	1/5	0	4/5

Dna

```
ttaccttaac
gatgtctgtc
acggcgttag
ccctaacgag
cgtcagaggt
```

Взяв наиболее вероятный 4-мер *Profile* из каждого ряда *Dna*, мы получим следующие 4-меры (показаны красным):

Motifs(*Profile*, *Dna*)

```
ttacctaac
gatgtctgtc
acggcgttag
ccctaacgag
cgtcagaggt
```

В общем, мы можем начать с набора случайно выбранных *k*-меров *Motifs* в *Dna*, построить *Profile*(*Motifs*) и использовать этот профиль для создания нового набора *k*-меров:

$$Motifs(Profile(Motifs), Dna).$$

Зачем нам это делать? Потому что мы надеемся, что *Motifs*(*Profile*(*Motifs*),*Dna*) даст лучший результат, чем первоначальная коллекция *k*-меров *Motifs*. Затем мы можем сформировать матрицу профиля этих *k*-меров,

$$Profile(Motifs(Profile(Motifs), Dna)),$$

и использовать ее для формирования наиболее вероятных *k*-меров,

$$Motifs(Profile(Motifs(Profile(Motif), Dna)), Dna).$$

Можно продолжить итерацию…

$$\ldots Profile(Motifs(Profile(Motifs(Profile(Motifs), Dna)), Dna))\ldots$$

до тех пор, пока счет построенных мотивов продолжает улучшаться, что и делает **RandomizedMotifSearch**. Для реализации этого алгоритма вам потребу-

ется случайным образом выбрать начальный набор *k*-меров, образующих матрицу мотивов *Motifs*. Для этого вам понадобится генератор случайных чисел (обозначается *RandomNumber*(*N*)), который с равной вероятностью выдает любое целое число от 1 до *N*. Вы можете рассматривать этот генератор случайных чисел как беспристрастный *N*-гранный кубик.

```
RandomizedMotifSearch(Dna, k, t)
    Случайно выберите k-меры Motifs = (Motif_1, ..., Motif_t) в каждой
    последовательности Dna
    BestMotifs ← Motifs
    while до бесконечности
        Profile ← Profile(Motifs)
        Motifs ← Motifs(Profile, Dna)
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
        else
            return BestMotifs
```

Упражнение. Докажите, что процедура **RandomizedMotifSearch** в конце концов закончится.

Поскольку однократный запуск **RandomizedMotifSearch** может привести к довольно бедному набору мотивов, биоинформатики обычно запускают этот алгоритм тысячи раз. В каждом прогоне они начинают с нового случайно выбранного набора *k*-меров, набирая лучший набор *k*-меров, найденный во всех этих прогонах.

Почему рандомизированный поиск мотивов работает

На первый взгляд **RandomizedMotifSearch** кажется обреченным. Как этот алгоритм, который начинает со случайного предположения, может найти что-то полезное? Чтобы исследовать **RandomizedMotifSearch**, давайте запустим его на пяти коротких цепях с имплантированным (4, 1)-мотивом ACGT (показан заглавными буквами ниже) и представим, что он выбирает следующие 4-мерные *Motifs* (показаны красным) на первой итерации. Как и ожидалось, он пропускает имплантированный мотив почти в каждой строке.

```
          ttACCTtaac
          gATGTctgtc
Dna       ccgGCGTtag
          cactaACGAg
          cgtcagAGGT
```

Ниже, мы строим матрицу профиля *Profile*(*Motifs*) выбранных 4-меров.

Motifs			
t	a	a	c
G	T	c	t
c	c	g	G
a	c	t	a
A	G	G	T

Profile(*Motifs*)				
A:	0,4	0,2	0,2	0,2
C:	0,2	0,4	0,2	0,2
G:	0,2	0,2	0,4	0,2
T:	0,2	0,2	0,2	0,4

Теперь мы можем вычислить вероятности каждого 4-мера в *Dna* по этой матрице профиля. Например, вероятность первого 4-мера в первой цепи *Dna* равна $Pr(\text{ttAC}|Profile) = 0{,}2 \cdot 0{,}2 \cdot 0{,}2 \cdot 0{,}2 = 0{,}0016$. Максимальные вероятности в каждой строке показаны ниже красным цветом.

ttAC	tACC	ACCT	CCTt	CTta	Ttaa	taac
0,0016	0,0016	**0,0128**	0,0064	0,0016	0,0016	0,0016
gATG	ATGT	TGTc	GTct	Tctg	ctgt	tgtc
0,0016	**0,0128**	0,0016	0,0032	0,0032	0,0032	0,0016
ccgG	cgGC	gGCG	GCGT	CGTt	GTta	Ttag
0,0064	0,0036	0,0016	**0,0128**	0,0032	0,0016	0,0016
cact	acta	ctaA	taAC	aACG	ACGA	CGAg
0,0032	0,0064	0,0016	0,0016	0,0032	**0,0128**	0,0016
cgtc	gtca	tcag	cagA	agAG	gAGG	AGGT
0,0016	0,0016	0,0016	0,0032	0,0032	0,0032	**0,0128**

Мы выбираем наиболее вероятные 4-меры в каждой строке выше в качестве *Motifs* нашей новой коллекции (показано ниже). Обратите внимание, что в этой коллекции собраны все пять имплантированных мотивов *Dna*!

	tt**ACCT**taac
	g**ATGT**ctgtc
Dna	ccg**GCGT**tag
	cacta**ACGA**g
	cgtcag**AGGT**

ОСТАНОВИТЕСЬ и задумайтесь. Как это возможно, что случайно выбранные *k*-меры привели нас к правильным имплантированным *k*-мерам? Если вы думаете, что мы подделали этот пример, выберите свои собственные начальные 4-меры и посмотрите, что произойдет.

Для задачи о тонком мотиве с имплантированным 15-мерным AAAAAAAAGGGGGGG, когда мы запускаем **RandomizedMotifSearch**

100 000 раз (каждый раз с новыми случайно выбранными *k*-мерами), он выдает 15-меры, показанные на рис. 2.7, как самый низкий счет коллекции мотивов по всем итерациям, в результате чего получается согласованная строка **AAAAAAAAacaGGGG** со счетом 43. Эти строки лишь немного менее консервативны, чем набор имплантированных (15, 4)-мотивов со счетом 40 (или мотив, возвращенный **GreedyMotifSearch** со счетом 41), и это в значительной степени фиксирует внедренный мотив. Более того, в отличие от **GreedyMotifSearch**, с **RandomizedMotifSearch** можно выполнять большее количество итераций, чтобы обнаруживать все лучшие и лучшие мотивы.

		Score
Motifs	AAAtAcAgACAGcGt	5
	AAAAAAtAgCAGGGt	3
	tAAAAtAAACAGcGG	3
	AcAgAAAAAaAGGGG	3
	AAAAtAAAACtGcGa	4
	AtAgAcgAACAcGGt	6
	cAAAAAgAgaAGGGG	4
	AtAgAAAAggAaGGG	5
	AAgAAAAAAgAGaGG	3
	cAtAAtgAACtGtGa	7
Consensus(*Motifs*)	AAAAAAAAACAGGGG	43

Рис. 2.7 Набор строк *Motifs* с наименьшим счетом, созданный в результате 100 000 запусков **RandomizedMotifSearch**, вместе с их консеснусной строкой и счетом для задачи о тонком мотиве

ОСТАНОВИТЕСЬ и задумайтесь. Даст ли ваш запуск **RandomizedMotifSearch** похожую консенсусную строку? Сколько раз нужно запустить **RandomizedMotifSearch** для получения имплантированного (15, 4)-мотива с расстоянием 40?

Хотя мотивы, выдаваемые **RandomizedMotifSearch**, немного менее консервативны, чем мотивы, выдаваемые **MedianString**, **RandomizedMotifSearch** имеет то преимущество, что может находить более длинные мотивы (поскольку **MedianString** становится слишком медленным для более длинных мотивов). В эпилоге мы увидим, почему эта функция важна на практике.

Почему рандомизированный алгоритм работает так хорошо?

В предыдущем разделе мы начали с набора имплантированных мотивов (с консенсусом ACGT), в результате чего была получена следующая матрица профиля.

A:	**0,8**	0,0	0,0	0,2
C:	0,0	**0,6**	0,2	0,0
G:	0,2	0,2	**0,8**	0,0
T:	0,0	0,2	0,0	**0,8**

Если бы строки в *Dna* были действительно случайными, то можно было бы ожидать, что все нуклеотиды в выбранных *k*-мерах будут равновероятными, что приводит к ожидаемому профилю, в котором каждая запись равна примерно 0,25:

A:	0,25	0,25	0,25	0,25
C:	0,25	0,25	0,25	0,25
G:	0,25	0,25	0,25	0,25
T:	0,25	0,25	0,25	0,25

Такой **однородный профиль** практически бесполезен для поиска мотива, потому что, согласно этому профилю, ни одна строка не является более вероятной, чем любая другая, и потому что он не дает никаких подсказок о том, как выглядит имплантированный мотив.

На противоположном конце спектра, если бы нам невероятно повезло, мы бы выбрали имплантированные *k*-меры *Motifs* с самого начала, что привело бы к первой из двух матриц профиля выше. На практике мы, скорее всего, получили бы профиль где-то посередине между этими двумя крайностями, например следующий:

A:	**0,4**	0,2	0,2	0,2
C:	0,2	**0,4**	0,2	0,2
G:	0,2	0,2	**0,4**	0,2
T:	0,2	0,2	0,2	**0,4**

Эта матрица профиля уже начала указывать нам на имплантированный мотив ACGT, т. е. ACGT является наиболее вероятным 4-мером, который может генерироваться этим профилем. К счастью, **RandomizedMotifSearch** разработан таким образом, что последующие шаги имеют хорошие шансы привести нас к этому имплантированному мотиву (хотя и не наверняка).

Если вы все еще сомневаетесь в эффективности рандомизированных алгоритмов, рассмотрите следующий аргумент. Мы уже заметили, что если бы строки в *Dna* были случайными, то **RandomizedMotifSearch** стартовал бы с почти однородного профиля, и работать было бы не с чем. Однако ключевое наблюдение состоит в том, что последовательности в *Dna* не случайны, потому что они включают имплантированный мотив! Эти множественные появления одного и того же мотива могут отклонить матрицу профиля от однородного профиля к имплантированному мотиву. Например, снова рассмотрим исходные случайно выбранные *k*-меры *Motifs* (показаны красным):

```
          ttACCTtaac
          gATGTctgtc
Dna       ccgGCGTtag
          cactaACGAg
          cgtcagAGGT
```

Вы увидите, что 4-мер **AGGT** в последней цепи просто случайно захватывает имплантированный мотив. На самом деле профиль, сформированный из оставшихся 4-меров (**taac**, **GTct**, **ccgG**, and **acta**)), является однородным. Обратите внимание, что только полностью захваченные мотивы (типа **AGGT**), а не частично захваченные мотивы (типа **GTct** или **ccgG**) вносят вклад в статистическую погрешность в матрице профиля.

Упражнение. Вычислите вероятность того, что десять случайно выбранных 15-меров из десяти цепей длиной 600 нуклеотидов (например, в задаче о тонком мотиве) захватят хотя бы один имплантированный 15-мер. Точность результата должна быть не ниже 0,000001.

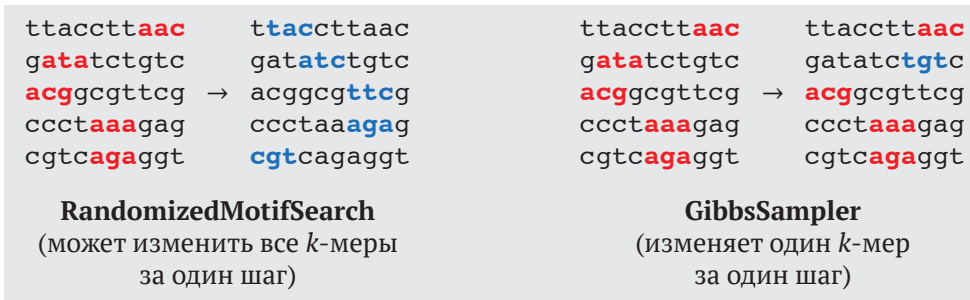
Хотя вероятность того, что случайно выбранные *k*-меры соответствуют *всем* имплантированным мотивам, незначительна, вероятность того, что они захватывают *по крайней мере один* имплантированный мотив, значительна. Даже в случае сложных задач с поиском мотивов, для которых эта вероятность мала, мы можем запускать **RandomizedMotifSearch** много раз, так что почти наверняка будет найден хотя бы один имплантированный мотив, тем самым мы получим статистическую погрешность, указывающую на правильный мотив.

К сожалению, захвата одного имплантированного мотива часто недостаточно, чтобы направить **RandomizedMotifSearch** к оптимальному решению. Поэтому, поскольку количество начальных позиций *k*-меров огромно, стратегия случайного выбора мотивов часто не так успешна, как в простом примере выше. Вероятность того, что эти случайно выбранные *k*-меры смогут привести нас к оптимальному решению, относительно мала.

Упражнение. Вычислите вероятность того, что десять случайно выбранных 15-меров из десяти 600-нуклеотидных цепей в задаче на тонкие мотивы захватят по крайней мере два имплантированных 15-мера. Точность результата должна быть не ниже 0,000001.

Сэмплирование по Гиббсу

Обратите внимание, что **RandomizedMotifSearch** может изменить все *t* строк в *Motifs* за одну итерацию. Эта стратегия может оказаться безрассудной, поскольку некоторые правильные мотивы (захваченные в *Motifs*) потенциально могут быть отброшены на следующей итерации. **GibbsSampler** – это более осторожный итерационный алгоритм, который на каждой итерации проверяет только один *k*-мер из текущего набора мотивов и решает, либо сохранить его, либо заменить новым. Таким образом, этот алгоритм перемещается с большей осторожностью в пространстве всех мотивов, как показано ниже.



Как и **RandomizedMotifSearch**, **GibbsSampler** начинает со случайно выбранных *k*-меров в каждой из *t* строк ДНК, но на каждой итерации делает случайный, а не детерминированный выбор. Он использует случайно выбранные *k*-меры *Motifs* = (*Motif*$_1$, …, *Motif*$_t$), чтобы получить другой (надеемся, больший по счету) набор *k*-меров. В отличие от **RandomizedMotifSearch**, который определяет новые мотивы по четкой процедуре как

$$Motifs(Profile(Motifs), Dna),$$

GibbsSampler случайным образом выбирает целое число *i* от 1 до *t*, а затем случайным образом изменяет один *k*-мер *Motif*$_i$.

Чтобы описать, как **GibbsSampler** обновляет *Motifs*, нам потребуется немного более продвинутый генератор случайных чисел. При заданном распределении вероятностей (p_1, …, p_n) этот генератор случайных чисел, обозначенный как *Random*(p_1, …, p_n), моделирует *n*-стороннюю несимметричную (неравновероятную) игральную кость и выдает целое число *i* с вероятностью p_i. Например, стандартный шестигранный кубик представляет собой генератор случайных чисел.

$$Random(1/6, 1/6, 1/6, 1/6, 1/6, 1/6),$$

тогда как несимметричную кость можно представить генератором случайных чисел

$$\text{R } Random(0{,}1, 0{,}2, 0{,}3, 0{,}05, 0{,}1, 0{,}25).$$

GibbsSampler дополнительно обобщает генератор случайных чисел, используя функцию $Random(p_1, \ldots, p_n)$, определенную для любого набора неотрицательных чисел, т. е. обязательно удовлетворяющую условию $\sum_{i=1}^{n} p_i = 1$. В частности, если $\sum_{i=1}^{n} p_i = C > 0$, то $Random(p_1, \ldots, p_n)$ определяется как $Random(p_1/C, \ldots, p_n/C)$, где $(p_1/C, \ldots, p_n/C)$ – распределение вероятностей. Например, при значениях $(p_1, p_2, p_3) = (0{,}1, 0{,}2, 0{,}3)$ с $0{,}1 + 0{,}2 + 0{,}3 = 0{,}6$,

$$Random(0{,}1, 0{,}2, 0{,}3) = Random(0{,}1/0{,}6, 0{,}2/0{,}6, 0{,}3/0.6) = Random(1/6, 1/3, 1/2).$$

ОСТАНОВИТЕСЬ и задумайтесь. Выполните $Random(p_1, \ldots, p_n)$ так, чтобы она использовала $RandomNumber(X)$ (для правильно выбранного целого числа X) в качестве подпрограммы.

Ранее мы определили понятие – наиболее вероятный k-мер *Profile* в строке. Теперь мы определяем ***Profile*-случайный сгенерированный *k*-мер** в строке *Text*. Для каждого k-мера *Profile* в тексте вычислите вероятность $Pr(Pattern|Profile)$, в результате чего $n = |Text| - k + 1$ вероятностей $(p_1, \ldots, p_n)$. Сумма этих вероятностей не обязательно равна 1, но мы все равно можем сформировать генератор случайных чисел $Random(p_1, \ldots, p_n)$ на их основе. **GibbsSampler** использует этот генератор случайных чисел для выбора случайно сгенерированного профиля k-мера на каждом шаге: если игральная кость выбрасывает число i, то мы определяем случайно сгенерированный профиль k-мер как i-й k-мер в *Text*. Хотя приведенный ниже псевдокод повторяет эту процедуру N раз, на практике **GibbsSampler** зависит от различных правил останова, которые выходят за рамки этой главы.

```
GibbsSampler (Dna, k, t, N)
    Случайно выбранные k-меры Motifs = (Motif1, ..., Motift) в каждой строке Dna
    BestMotifs ← Motifs
    for j ← 1 до N
        i ← Random(t)
        Profile ← матрица прифиля, сформированная из всех строк в Motifs за
        исключением Motif_i
        Motifi ← Profile-случайно сгенерированный k-мер в i-й строке
        if Score(Motifs) < Score(BestMotifs)
            BestMotifs ← Motifs
    return BestMotifs
```

ОСТАНОВИТЕСЬ и задумайтесь. Обратите внимание, что в отличие от **RandomizedMotifSearch**, который всегда перемещается от мотивов с более высоким счетом к мотивам с более низким, **GibbsSampler** может перемещаться от мотивов с более низким счетом к мотивам с более высоким. Какой в этом смысл?

Сэмплирование по Гиббсу в действии

Мы проиллюстрируем, как **GibbsSampler** работает с теми же строками *Dna*, которые мы рассматривали ранее. Представьте, что на начальном этапе алгоритм выбрал следующие 4-меры (показаны красным) и случайным образом выбрал третью строку для удаления. Чтобы быть более точным, **GibbsSampler** на самом деле не удаляет третью строку; он игнорирует ее на этом конкретном шаге и может анализировать ее снова на последующих шагах.

	ttACCT**taac**		ttACCT**taac**
	gAT**GTct**gtc		gAT**GTct**gtc
Dna	**ccgG**CGTtag	→	----------
	c**acta**ACGAg		c**acta**ACGAg
	cgtcag**AGGT**		cgtcag**AGGT**

Это приводит к следующим матрицам мотива, счета и профиля.

Motifs

t	a	a	c
G	T	c	t
a	c	t	a
A	G	G	T

Count(*Motifs*)

A:	2	1	1	1
C:	0	1	1	1
G:	1	1	1	0
T:	1	1	1	2

Profile(*Motifs*)

2/4	1/4	1/4	1/4
0	1/4	1/4	1/4
1/4	1/4	1/4	0
1/4	1/4	1/4	2/4

Обратите внимание, что матрица профиля лишь немного более консервативна, чем однородный профиль, что заставляет нас задаться вопросом, есть ли у нас шанс приблизиться к имплантированному мотиву. Теперь мы используем эту матрицу профиля для вычисления вероятностей всех 4-меров в удаленной строке ccgGCGTtag:

ccgG	cgGC	gGCG	GCGT	CGTt	GTta	Ttag
0	0	0	1/128	0	1/256	0

Обратите внимание, что все, кроме двух, из этих вероятностей равны нулю. Эта ситуация аналогична той, с которой мы столкнулись с **GreedyMotifSearch**, и, как и раньше, нам нужно дополнить нулевые вероятности небольшими псевдосчетами, чтобы избежать катастрофических результатов.

Применение правила наследования Лапласа к приведенной выше матрице счета дает следующие обновленные матрицы счета и профиля:

Count(*Motifs*)

A:	3	2	2	2
C:	1	2	2	2
G:	2	2	2	1
T:	2	2	2	3

Profile(*Motifs*)

3/8	2/8	2/8	2/8
1/8	2/8	2/8	2/8
2/8	2/8	2/8	1/8
2/8	2/8	2/8	3/8

После добавления псевдосчетов вероятности 4-меров в удаленной строке ccgGCGTtag пересчитываются следующим образом:

ccgG	cgGC	gGCG	GCGT	CGTt	GTta	Ttag
$4/8^4$	$8/8^4$	$8/8^4$	$24/8^4$	$12/8^4$	$16/8^4$	$8/8^4$

Поскольку сумма этих вероятностей C = 80/84, наша гипотетическая семигранная кость представлена генератором случайных чисел

$$Random\left(\frac{4/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}, \frac{24/8^4}{80/8^4}, \frac{12/8^4}{80/8^4}, \frac{16/8^4}{80/8^4}, \frac{8/8^4}{80/8^4}\right)$$
$$= Random\left(\frac{4}{80}, \frac{8}{80}, \frac{8}{80}, \frac{24}{80}, \frac{12}{80}, \frac{16}{80}, \frac{8}{80}\right).$$

Давайте предположим, что после броска этого семигранного кубика мы придем к случайно сгенерированному 4-меру *Profile* GCGT (четвертый 4-мер в удаленной строке). Удаленная строка ccgGCGTtag теперь снова добавляется в коллекцию мотивов, и **GCGT** заменяет ранее выбранную ccgG в третьей строке *Dna*, как показано ниже. Затем мы бросаем правильный пятигранный кубик и случайным образом для удаления выбираем первую строку из *Dna*.

	ttACCT**taac**		----------
	gAT**GTct**gtc		gAT**GTct**gtc
Dna	ccg**GCGT**tag	→	ccg**GCGT**tag
	c**acta**ACGAg		c**acta**ACGAg
	cgtcag**AGGT**		cgtcag**AGGT**

После построения матриц мотива и профиля получаем следующее:

Motifs

G	T	c	t
G	C	G	T
a	c	t	a
A	G	G	T

Profile(*Motifs*)

A:	2/4	0	0	1/4
C:	0	2/4	1/4	0
G:	2/4	1/4	2/4	0
T:	0	1/4	1/4	3/4

Обратите внимание, что матрица профиля выглядит более смещенной в сторону имплантированного мотива, чем предыдущая матрица профиля. Мы обновляем матрицы счета и профиля с помощью псевдосчетов:

Count(*Motifs*)

A:	3	1	1	2
C:	1	3	2	1
G:	3	2	3	1
T:	1	2	2	4

Profile(*Motifs*)

3/8	1/8	1/8	2/8
1/8	3/8	2/8	1/8
3/8	2/8	3/8	1/8
1/8	2/8	2/8	4/8

Затем мы вычисляем вероятности всех 4-меров в удаленной строке ttACCTtaac:

ttAC	tACC	ACCT	CCTt	CTta	Ttaa	taac
$2/8^4$	$2/8^4$	$72/8^4$	$24/8^4$	$8/8^4$	$4/8^4$	$1/8^4$

Бросая семигранный кубик, мы приходим к случайно сгенерированному *k*-меру *Profile* **ACCT**, который добавляем в коллекцию *Motifs*. После повторного броска пятигранного кубика мы случайным образом выбираем для удаления четвертую строку.

	ttACCTtaac		ttACCTtaac
	gATGTctgtc		gATGTctgtc
Dna	ccgGCGTtag	→	ccgGCGTtag
	cactaACGAg		----------
	cgtcagAGGT		cgtcagAGGT

Далее мы добавляем псевдосчеты и строим результирующие матрицы счета и профиля:

Motifs

A	C	C	T
G	T	c	t
G	C	G	T
A	G	G	T

Count(*Motifs*)

A:	3	1	1	1
C:	1	3	3	1
G:	3	2	3	1
T:	1	2	1	5

Profile(*Motifs*)

3/8	1/8	1/8	1/8
1/8	3/8	3/8	1/8
3/8	2/8	3/8	1/8
1/8	2/8	1/8	5/8

Теперь мы вычисляем вероятности всех 4-меров в удаленной строке cactaACGAg:

cact	acta	ctaA	taAC	aACG	ACGA	CGAg
$15/8^4$	$9/8^4$	$2/8^4$	$1/8^4$	$9/8^4$	$27/8^4$	$2/8^4$

Нужно бросить семигранный кубик, чтобы получить случайно сгенерированный 4-мер *Profile*. Предполагая наиболее вероятный сценарий, в котором мы выбираем **ACGA**, мы обновляем выбранные 4-меры следующим образом:

```
        ttACCTtaac
        gATGTctgtc
Dna     ccgGCGTtag
        cactaACGAg
        cgtcagAGGT
```

Как видите, алгоритм начинает сходиться. Будьте уверены, что последующая итерация выдаст все имплантированные мотивы после того, как мы выберем вторую строку в *Dna* (когда неправильный 4-мер GTct, вероятно, изменится на имплантированный (4, 1)-мотив ATGT).

ОСТАНОВИТЕСЬ и задумайтесь. Запустите **GibbsSampler** на задаче тонкого мотива. Что вы получили?

Хотя **GibbsSampler** работает хорошо во многих случаях, он может выдать субоптимальное решение, особенно для сложных задач поиска с неуловимыми мотивами. **Локальный оптимум** — это решение, оптимальное в небольшом соседствующем наборе решений, в отличие от **глобального оптимума**, или оптимальное решение среди всех возможных решений. Поскольку **GibbsSampler** исследует только небольшое подмножество решений, он может «застрять» в локальном оптимуме. По этой причине, как и в случае **RandomizedMotifSearch**, его следует запускать много раз в надежде, что один из этих запусков даст мотивы с лучшими счетами. Тем не менее сходимость к локальному оптимуму – лишь одна из многих проблем, которые мы должны учитывать при поиске мотивов; см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Осложнения в поиске мотивов**, чтобы узнать о некоторых других проблемах.

Когда мы запускаем **GibbsSampler** 2000 раз на задаче о тонком мотиве с имплантированным 15-мером **AAAAAAAAGGGGGGG** (каждый раз с новыми случайно выбранными *k*-мерами для $N = 200$ итераций), он выдает набор мотивов с консенсусом **AAAAAAgAGGGGGGt** и *Score*(*Motifs*), равным 38. Это даже ниже, чем счет 40, ожидаемый от имплантированных мотивов!

Эпилог. Как туберкулез впадает в спячку, чтобы спрятаться от антибиотиков?

Туберкулез (TB) – это инфекционное заболевание, вызываемое бактерией *Mycobacterium tuberculosis* (MTB), от которой ежегодно умирает более миллиона человек. Хотя распространение туберкулеза значительно сократилось благодаря антибиотикам, в настоящее время появляются штаммы, устойчивые

ко всем доступным методам лечения. MTB успешен как патоген, потому что он может сохраняться в организме человека в течение десятилетий, не вызывая заболевания; на самом деле одна треть населения мира заражена **латентными формами MTB**, при которых MTB находится в состоянии покоя в организме хозяина и может реактивироваться, или не сделать этого, в более позднее время. Широкое распространение латентных форм затрудняет борьбу с эпидемиями туберкулеза. Поэтому биологи заинтересованы в том, чтобы выяснить, что делает заболевание латентным и как MTB активируется в организме хозяина.

Остается неясным, почему MTB может оставаться латентным так долго и как он выживает во время латентности. Устойчивость латентного туберкулеза к антибиотикам означает, что MTB может иметь способность отключать экспрессию большинства генов и оставаться в состоянии покоя, подобно медведям, впадающим в зимнюю спячку. Спячка у бактерий называется **споруляцией**, потому что многие бактерии образуют защитные и метаболически спящие *споры*, которые могут выживать в жестких условиях, позволяя бактериям сохраняться в окружающей среде до тех пор, пока условия не улучшатся.

Гипоксия, или дефицит кислорода, часто связана со латентными формами туберкулеза. Биологи обнаружили, что MTB становится бездействующим в среде с низким содержанием кислорода, по-видимому, с идеей, что легкие хозяина восстановятся достаточно, чтобы потенциально распространить болезнь в будущем. Поскольку MTB проявляет замечательную способность выживать в течение многих лет без кислорода, важно идентифицировать гены MTB, ответственные за развитие латентного состояния в условиях гипоксии. Биологов интересует поиск **фактора транскрипции**, который «чувствует» нехватку кислорода и запускает генетическую программу, влияющую на экспрессию многих генов, позволяя MTB адаптироваться к гипоксии.

В 2003 году биологи обнаружили **регулятор выживания в состоянии покоя (DosR)** – фактор транскрипции, который регулирует многие гены, экспрессия которых резко меняется в условиях гипоксии. Однако оставалось неясным, как DosR регулирует эти гены, и его сайт связывания фактора транскрипции оставался неизвестным. Пытаясь решить эту загадку, биологи провели эксперимент с массивом ДНК и обнаружили 25 генов, уровни экспрессии которых значительно менялись в условиях гипоксии. Имея восходящие области этих генов, каждая из которых имеет длину 250 нуклеотидов, мы хотели бы обнаружить скрытое сообщение, которое DosR использует для контроля экспрессии этих генов.

Чтобы немного упростить задачу, мы выбрали только 10 из 25 генов, получив **базу данных DosR**. Мы попытаемся идентифицировать мотивы в этом наборе данных, используя арсенал разработанных нами инструментов поиска мотивов. Однако мы не будем подсказывать вам мотив DosR.

Какой размер k-мера мы должны выбрать для анализа базы данных DosR с использованием **MedianString** и **RandomizedMotifSearch**? Принимая случайные предположения и запуская эти алгоритмы для k от 8 до 12, вы получите согласованные строки, показанные ниже.

MedianString			RandomizedMotifSearch		
k	Consensus	Score	*k*	Consensus	Score
8	CATCGGCC	11	8	CCGACGGG	13
9	GGCGGGGAC	16	9	CCATCGGCC	16
10	GGTGGCCACC	19	10	CCATCGGCCC	21
11	GGACTTCCGGC	20	11	ACCTTCGGCCC	25
12	GGACTTCCGGCC	23	12	GGACCAACGGCC	28

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы сделать вывод о месте связывания DosR на основе этих медианных строк? Как вы думаете, какова длина сайта связывания?

Обратите внимание, что, хотя консенсусные строки, выдаваемые **RandomizedMotifSearch**, обычно отклоняются от медианных строк, консенсусный 12-мер (GGACCAACGGCC, со счетом 28) очень похож на медианную строку (GGACTTCCGGCC, со счетом 23).

Хотя мотивы, выдаваемые **RandomizedMotifSearch**, немного менее консервативны, чем мотивы, выдаваемые **MedianString**, прежний алгоритм имеет то преимущество, что может находить более длинные мотивы (поскольку **MedianString** становится слишком медленным для более длинных мотивов). Мотив длиной 20, выданный **RandomizedMotifSearch**, представляет собой CGGGACCTACGTCCCTAGCC (со счетом 57). Как показано ниже, согласованные строки длины 12, найденные **RandomizedMotifSearch** и **MedianString**, «встроены» с небольшими вариациями в более длинный мотив длины 20:

```
GGACCAACGGCC
CGGGACCTACGTCCCTAGCC
GGACTTCCGGCC
```

Наконец, в 2000 прогонах с $N = 200$ **GibbsSampler** выдал ту же согласованную строку длины 20 для набора данных DosR, что и **RandomizedMotifSearch**, но сгенерировал другой набор мотивов с меньшим счетом 55.

Как вы уже уяснили в этой главе, разные алгоритмы поиска мотивов дают несколько разные результаты, и остается неясным, как идентифицировать мотив DosR в MTB. Попробуйте ответить на этот вопрос и найти все предполагаемые мотивы DosR в MTB, а также все гены, которые они регулируют. Мы предоставим восходящие области всех 25 генов, идентифицированных в исследовании DosR, чтобы помочь вам решить следующую задачу.

Заключительная задача. Выведите профиль мотива DosR и найдите все его предполагаемые проявления в *Mycobacterium tuberculosis*.

☞ **Загрузить данные 2.1 (база данных DosR)**

Зарядная станция

Решение задачи медианной строки

Первая потенциальная проблема с реализацией **MedianString** – это написание функции для вычисления $\sum_{i=1}^{t} d(Pattern, Dna_i)$, суммы расстояний между *Pattern* в каждой строке в $Dna = \{Dna_1, \ldots, Dna_t\}$. Эта задача решается с помощью следующего псевдокода.

```
DistanceBetweenPatternAndStrings(Pattern, Dna)
    K ← |Pattern|
    distance ← 0
    for каждой строки Text в Dna
        HammingDistance ← ∞
        for каждого k-мера Pattern' в Text
            if HammingDistance > HammingDistance(Pattern, Pattern')
                HammingDistance ← HammingDistance(Pattern, Pattern')
        distance ← distance + HammingDistance
    return distance
```

Чтобы решить задачу медианной строки, нам нужно перебрать все возможные 4^k k-меры *Pattern* перед вычислением $d(Pattern, Dna)$. Приведенный ниже псевдокод представляет собой модификацию **MedianString** с использованием функции *NumberToPattern* (см. **ЗАРЯДНАЯ СТАНЦИЯ: Преобразование Patterns в Numbers и наоборот**, которая применяется для преобразования всех целых чисел от 0 до $4^k - 1$ во все возможные k-меры.

```
MedianString(Dna, k)
    distance ← ∞
    for i ← 0 до 4^k – 1
        Pattern ← NumberToPattern(i, k)
        if distance > DistanceBetweenPatternAndStrings(Pattern, Dna)
            distance ← DistanceBetweenPatternAndStrings(Pattern, Dna)
            Median ← Pattern
    return Median
```

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем использовать **Neighbours** из раздела **ЗАРЯДНАЯ СТАНЦИЯ: Генерация окрестности строки**, чтобы получить **AllStrings**?

Сопутствующие материалы

Экспрессия генов

Гены кодируют белки, а белки управляют функциональностью клеток. Следовательно, чтобы реагировать на изменения в окружающей среде, клетки должны контролировать уровни своих белков. Поток информации от ДНК к РНК и белку означает, что клетка может регулировать количество белков, которые она производит как во время транскрипции (ДНК в РНК), так и во время трансляции (РНК в белок).

Транскрипция начинается, когда РНК-полимераза связывается с **промотор-последовательностью** на цепи молекулы ДНК, которая часто располагается непосредственно выше по цепи от начальной точки транскрипции. Инициация транскрипции является для клетки удобной контрольной точкой для регуляции экспрессии генов, поскольку она находится в самом начале процесса производства белка. Гены, транскрибируемые в клетке, контролируются различными регуляторами транскрипции, которые могут усиливать или подавлять транскрипцию.

ДНК-чипы

ДНК-чип (микрочип) представляет собой набор молекул ДНК, прикрепленных к твердой поверхности. Каждое пятно на чипе содержит уникальную цепь ДНК, называемую **пробой**, которая измеряет уровень экспрессии определенного гена, известного как **мишень**. В большинстве массивов пробы синтезируются, а затем прикрепляются к стеклянному или кремниевому чипу (рис. 2.8).

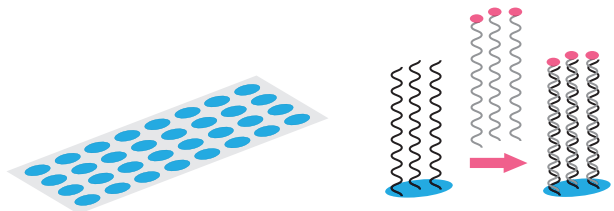


Рис. 2.8 ДНК с флуоресцентной меткой связывается с комплементарной пробой на ДНК-чипе

Далее флуоресцентно меченные мишени связываются с соответствующей пробой (например, когда их цепи комплементарны), генерируя флуоресцентный сигнал при освещении. Сила этого сигнала зависит от количества молекул мишени, которые связываются с пробой в данном месте. Таким образом, чем выше уровень экспрессии гена, тем выше интенсивность его флуоресцентного сигнала на чипе. Поскольку микрочип может содержать миллионы проб, биологи могут измерить экспрессию множества генов в одном эксперименте,

с одним микрочипом. В эксперименте с ДНК-чипом, который идентифицировал «вечерний элемент» у *Arabidopsis thaliana*, была измерена экспрессия 8000 генов.

Игла Бюффона

Граф де Бюффон был выдающимся натуралистом XVIII века, чьи труды по естествознанию в то время имели большой успех. Однако его первая статья была написана в области математики; в 1733 году он написал эссе о средневековой французской игре под названием «Le jeu de franc carreau» (игра «франк-карро»). В этой игре один игрок подбрасывает монету в воздух, и монета приземляется на шахматную доску. Игрок выигрывает, если монета полностью попадает в одну из клеток на доске, и проигрывает в противном случае (рис. 2.9 (слева)). Бюффон задал естественный вопрос: какова вероятность того, что игрок выиграет?

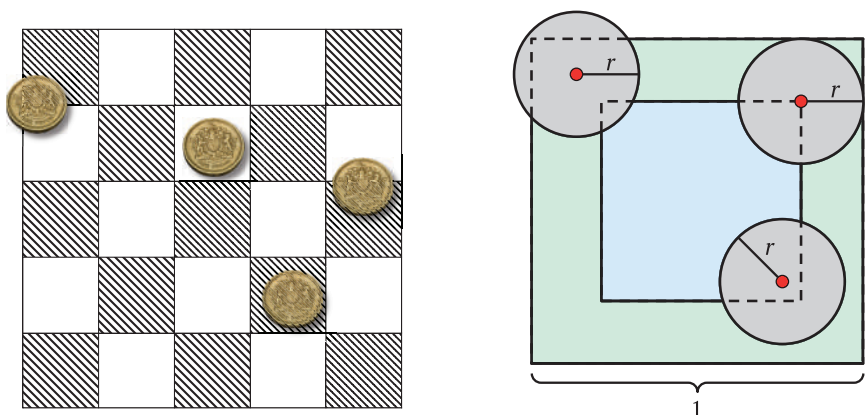


Рис. 2.9 (Слева) Игра «франк-карро» с четырьмя монетами. Две монеты приземлились в одном из квадратов шахматной доски и считаются выигрышными, а две другие приземлились на границе и считаются проигравшими. (Справа) Три монеты показаны на одной клетке шахматной доски (внешний зеленый квадрат); одна монета в проигрыше, другая в выигрыше, а третья спредставляет собой граничный случай. Вы можете видеть, что если монета имеет радиус r, то вероятность выигрыша в игре соответствует вероятности того, что центр монеты (показанный красной точкой) приземлится в пределах синего квадрата, длина стороны которого $1 - 2r$. Эта вероятность является отношением площади квадратов, что равно $(1 - 2r)^2$

Предположим, что шахматная доска состоит только из одного квадрата со стороной 1, что радиус монеты $r < 1/2$ и что центр монеты всегда попадает в квадрат. Тогда игрок может выиграть только в том случае, если центр круга попадает в воображаемый центральный квадрат со стороной $1 - 2r$ (рис. 2.9

(справа)). Если предположить, что монета падает где-нибудь на большом квадрате с одинаковой вероятностью, то вероятность того, что монета полностью попадет в меньшую клетку, определяется отношением площадей двух квадратов, или $(1 - 2r)^2$.

Четыре десятилетия спустя Бюффон опубликовал статью, описывающую аналогичную игру, в которой игрок равномерно бросает иголку на пол, покрытый длинными деревянными досками одинаковой ширины. В этой игре, известной как **игла Бюффона**, игрок выигрывает, если игла попадает на одну из досок полностью. Обратите внимание, что вычисление вероятности выигрыша теперь усложняется тем фактом, что стрелка описывается не только своим положением, но и ориентацией. Тем не менее первая игра дает нам представление о том, как решить эту задачу: как только мы зафиксируем положение центра иглы, ее совокупность различных возможных ориентаций образует круг (рис. 2.10 (слева)).

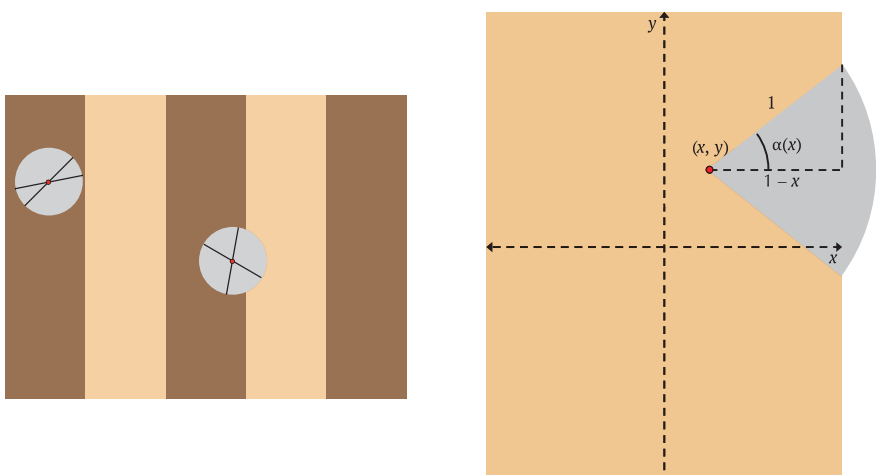


Рис. 2.10 (Слева) Как только мы зафиксируем центр иглы (показан красной точкой), совокупность возможных ориентаций иглы образует круг. В кружке слева стрелка всегда будет лежать в пределах темно-коричневой доски независимо от ее ориентации. В круге справа одна из двух игл лежит внутри темно-коричневой панели, а другая показана пересекающей соседнюю доску. (Справа) Как только мы зафиксируем точку (x, y) в качестве центра иглы, существует критический угол $\alpha(x)$ такой, что все углы между $-\alpha(x)$ и $\alpha(x)$ заставят иглу пересечься со следующей доской. На этом рисунке длина иглы равна ширине доски

Вероятность того, что игрок выиграет, зависит от соотношения длины иглы и ширины доски. Будем считать, что обе эти длины равны 2, и найдем вероятность *проигрыша* вместо выигрыша. С этой целью мы сначала зададим бо-

лее простой вопрос: если центр иглы будет каждый раз приземляться в одном и том же месте, то какова вероятность того, что игла пересечет доску?

Чтобы ответить на этот вопрос, давайте отобразим доску, на которую падает игла, на координатную плоскость с осью Y, разделяющей доску на две меньшие панели шириной 1 (рис. 2.10 (справа)). Если центр иглы находится в положении (x, y) с $x > 0$, то ее ориентация может быть представлена углом θ, где θ находится в промежутке от $-\pi/2$ до $\pi/2$ радиан. Если $\theta = 0$, то игла пересечет вертикальную линию $x = 1$; если $\theta = \pi/2$, то игла не пересечет линию $x = 1$. Что еще более важно, поскольку центральное положение иглы фиксировано, должен существовать некоторый критический угол $\alpha(x)$ такой, что игла всегда касается этой прямой, если $-\alpha(x) \leqslant \theta \leqslant \alpha(x)$. Если игла падает случайным образом, то любое значение θ между $-\pi/2$ и $\pi/2$ равновероятно, поэтому мы получаем, что вероятность выпадения при данном положении иглы равна $2 \cdot \alpha(x)/\pi$.

Следуя тем же рассуждениям, стрелка может занять любое положение x с равной вероятностью. Поэтому, чтобы найти вероятность проигрыша, $Pr(loss)$, мы должны вычислить «среднее» значение $2 \cdot \alpha(x)/\pi$, при том что x непрерывно изменяется от -1 до 1. Это среднее значение можно представить с помощью интеграла,

$$\Pr(loss) = \frac{\int_{-1}^{1} \frac{1 \cdot \alpha(x)}{\pi}}{1-(-1)} = \int_{-1}^{1} \frac{\alpha(x)}{\pi} dx = 2\int_{0}^{1} \frac{\alpha(x)}{\pi} dx.$$

Вернувшись к рис. 2.10 (справа) и применив элементарную тригонометрию, мы определим, что $\cos \alpha(x)$ равен $1 - x$, так что $\alpha(x) = \arccos(1 - x)$. После внесения этой замены в приведенное выше уравнение и обращения к нашему запыленному учебнику по математическому анализу мы определим, что вероятность $Pr(loss)$ должна быть равна $2/\pi$. Нетрудно видеть, что эта вероятность будет одинаковой при падении иглы на любое количество деревянных досок.

Но какое отношение игла Бюффона имеет к рандомизированным алгоритмам? В 1812 году не кто иной, как Лаплас, указал, что иглу Бюффона можно использовать для вычисления числа π; так родился первый в мире алгоритм Монте-Карло. В частности, мы можем вычислить вероятность проигрыша P_e *эмпирически*, тупо подбросив иглу в воздух тысячи раз (или попросив компьютер сделать это за нас). Как только мы вычислили эту эмпирическую вероятность, можно заключить, что P_e приблизительно равно $2/\pi$ и, таким образом,

$$\pi \approx 2/P_e.$$

ОСТАНОВИТЕСЬ и задумайтесь. Как изменится это приближение в следующих случаях?

1. Длина иглы меньше ширины досок.
2. Игла длиннее, чем ширина досок.

Сложности в поиске мотива

Поиск мотива становится затруднительным, если **фоновое распределение нуклеотидов** в образце смещено. В этом случае поиск *k*-меров с минимальным счетом или энтропией может привести к биологически нерелевантному мотиву, состоящему из наиболее часто встречающихся нуклеотидов в выборке. Например, если A имеет частоту 85 %, а T, G и C имеют частоту 5 %, то *k*-мер AA…AA может представлять мотив с минимальным счетом, тем самым маскируя биологически значимые мотивы. Например, соответствующий мотив GCCG со счетом 5 в приведенном ниже примере проигрывает мотиву **aaaa** со счетом 1.

```
          taaaaGTCGa
          acGCTGaaaa
Dna       aaaaGCCTat
          aCCCGaataa
          agaaaaGGCG
```

Поэтому, чтобы найти биологически значимые мотивы в образцах со смещенными частотами нуклеотидов, вы можете использовать обобщение энтропии, называемое «относительной энтропией» (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Относительная энтропия**).

Еще одна сложность в поиске мотивов заключается в том, что многие мотивы лучше всего представлены в алфавите, отличном от алфавита из четырех нуклеотидов. Пусть W обозначает либо A, либо T; S обозначает либо G, либо C; K обозначает либо G, либо T, а Y обозначает либо C, либо T. Теперь рассмотрим мотив CSKWYWWATKWATYYK, который представляет мотив CSRE в дрожжах (вспомните рис. 2.3 со стр. 76). Этот сильный мотив в гибридном алфавите соответствует 2^{11} различным мотивам в стандартном 4-буквенном алфавите нуклеотидов. Однако каждый из этих 2^{11} мотивов слишком слаб, чтобы его можно было найти с помощью алгоритмов, которые мы рассмотрели в этой главе.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	G/C	G/T	T/A	C/T	G/C	C/G	A	T	G/T	C/G	A	T	C/T	C/T	G/T

Относительная энтропия

Для набора строк *Dna* **относительная энтропия** матрицы профиля $4\times k$, $P = (p_{r,j})$ определяется как

$$\sum_{j=1}^{k} \sum_{r\in\{A,C,G,T\}} p_{r,j} \cdot \log_2(p_{r,j}/b_r) =$$

$$\sum_{j=1}^{k} \sum_{r\in\{A,C,G,T\}} p_{r,j} \cdot \log_2(p_{r,j}) - \sum_{j=1}^{k} \sum_{r\in\{A,C,G,T\}} p_{r,j} \cdot \log_2(b_r),$$

где b_r – частота встречаемости нуклеотида r в *Dna*. Обратите внимание, что сумме в выражении для энтропии предшествует отрицательный знак ($-\sum_{j=1}^{k}\sum_{r\in\{A,C,G,T\}} p_{r,j}\cdot\log_2(p_{r,j})$), тогда как сумма в левой части уравнения относительной энтропии не имеет этого отрицательного знака. Следовательно, хотя мы минимизировали энтропию матрицы мотивов, теперь мы попытаемся максимизировать относительную энтропию.

Выражение $-\sum_{j=1}^{k}\sum_{r\in\{A,C,G,T\}} p_{r,j}\cdot\log_2(b_r)$ называется **перекрестной энтропией** матрицы профиля P; обратите внимание, что относительная энтропия матрицы профиля – это просто разница между перекрестной энтропией профиля и его энтропией. Например, относительная энтропия для мотива GCCG в примере из раздела **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Сложность поиска мотива** равна 9,85 – 3,53 = 6,32, как показано ниже. В этом примере b_A = 0,5, b_C = 0,18, b_G = 0,2 и b_T = 0,12.

Motifs	G	T	C	G
	G	C	T	G
	G	C	C	T
	C	C	C	G
	G	G	C	G
Profile(*Motifs*)	A: 0,0	0,0	0,0	0,0
	C: 0,2	0,6	0,8	0,0
	G: 0,8	0,2	0,0	0,8
	T: 0,0	0,2	0,2	0,2
Энтропия	0,72 + 1,37 + 0,72 + 0,72 = 3,53			
Перекрестная энтропия	2,35 + 2,56 + 2,47 + 2,47 = 9,85			

Для более консервативного, но нерелевантного мотива аааа относительная энтропия равна 4,18 – 0,72 = 3,46, как показано ниже. Таким образом, GCCG проигрывает аааа по энтропии, но выигрывает по относительной энтропии.

Motifs	a	a	a	a
	a	a	a	a
	a	a	a	a
	a	t	a	a
	a	a	a	a
Profile(*Motifs*)	A: 1,0	0,8	1,0	1,0
	C: 0,0	0,0	0,0	0,0
	G: 0,0	0,0	0,0	0,0
	T: 0,0	0,2	0,0	0,0
Энтропия	0,0 + 0,72 + 0,0 + 0,0 = 0,72			
Перекрестная энтропия	0,94 + 1,36 + 0,94 + 0,94 = 4,18			

Библиографические примечания

Konopka и Benzer, 1971[1], вывели мух с аномально короткими (19 ч) и длинными (28 ч) суточными ритмами, а затем проследили эти аномалии до конкретного гена. Harmer et al., 2000[2], открыли сайт связывания фактора вечерней транскрипции, который управляет циркадными ритмами у растений. Отличное подтверждение этого открытия дано Cristianini and Hahn, 2006[3]. Park et al., 2003[4], обнаружили транскрипционный фактор, который опосредует гипоксическую реакцию *Mycobacterium tuberculosis*.

Hertz and Stormo, 1999[5], описали первый жадный алгоритм поиска мотивов. Общая структура выборки Гиббса была описана Geman and Geman, 1984[6], и была названа «сэмплирование Гиббса» в связи с ее сходством с некоторыми методами в статистической механике (Josiah Willard Gibbs был одним из основателей статистической механики). Lawrence et al., 1993[7], адаптировали сэмплирование Гиббса для поиска мотивов.

[1] https://www.ncbi.nlm.nih.gov/pubmed/5002428.
[2] https://www.ncbi.nlm.nih.gov/pubmed/11118138.
[3] https://www.cambridge.org/core/books/introduction-to-computational-genomics/863C62220C06825CE3B8F8E462D0390F.
[4] https://www.ncbi.nlm.nih.gov/pubmed/12694625.
[5] https://www.ncbi.nlm.nih.gov/pubmed/10487864.
[6] https://ieeexplore.ieee.org/document/4767596.
[7] https://science.sciencemag.org/content/262/5131/208.

Глава 3

Как мы собираем геномы?

Алгоритмы графов

Взрывающиеся газеты

Представьте, что мы сложили сто пачек газетного номера «Нью-Йорк Таймс» от 27 июня 2000 года на заряд динамита, а затем подожгли фитиль. Продолжите эту фантазию дальше и предположите, что газеты не сгорели полностью, а вместо этого разлетелись тлеющими клочками причудливой формы. Как мы могли бы использовать эти крошечные обрывки газет, чтобы выяснить, что было напечатано в новостях от 27 июня 2000 года? Мы назовем эту сумасшедшую головоломку **Задачей газет** (рис. 3.1).

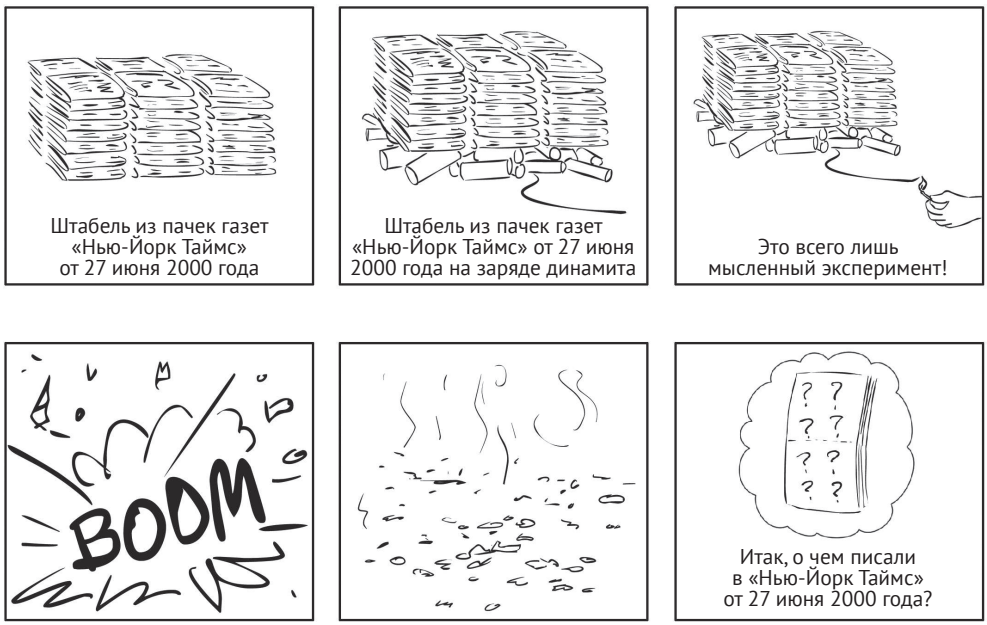


Рис. 3.1 Не пытайтесь повторить это дома! Но, каким бы безумием вам это ни казалось, Задача газет служит аналогом вычислительной основы сборки генома

Задача газет гораздо сложнее, чем может показаться. Поскольку у нас было много экземпляров одного и того же номера газеты и поскольку мы, несомненно, потеряли некоторую информацию во время взрыва, мы не можем просто склеить один из экземпляров газеты так же, как собираем пазл. Вместо этого нам нужно использовать *перекрывающиеся* фрагменты из разных экземпляров газеты, чтобы реконструировать новости дня, как показано на рис. 3.2.

Хорошо, спросите вы, *но какое отношение взрывающиеся газеты имеют к биологии*? Определение порядка нуклеотидов в геноме, или **секвенирование генома**, представляет собой фундаментальную задачу биоинформатики. Геномы различаются по длине; ваш собственный геном составляет примерно 3 млрд нуклеотидов, тогда как геном *Amoeba dubia*, аморфного одноклеточ-

ного организма, примерно в 200 раз длиннее! Этот одноклеточный организм соперничает с редким японским цветком *Paris japonica* за звание вида с самым длинным геномом. Первый секвенированный геном, принадлежащий бактериофагу φX174 (т. е. вирусу, который охотится на бактерии), имел всего 5386 нуклеотидов, и его секвенирование было сделано Фредериком Сенгером в 1977 году. Спустя четыре десятилетия после этого открытия, получившего Нобелевскую премию, секвенирование генома вырвалось на передний план исследований в области биоинформатики, поскольку стоимость секвенирования генома резко упала. Из-за снижения стоимости секвенирования у нас теперь есть тысячи секвенированных геномов, в том числе геномов многих млекопитающих (рис. 3.3).

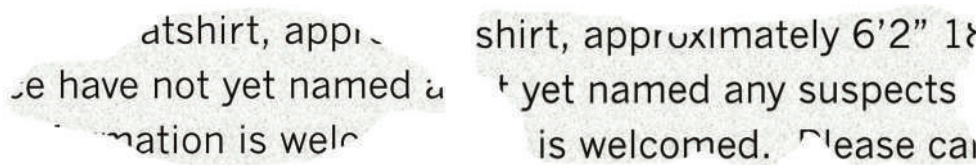


Рис. 3.2 В задаче о газетах нам нужно использовать перекрывающиеся клочки бумаги, чтобы восстановить текст

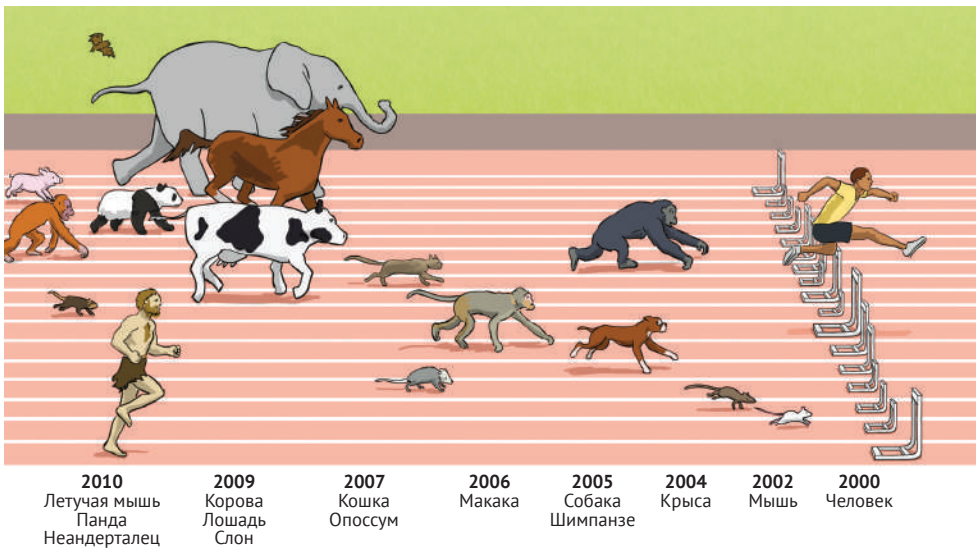


Рис. 3.3 Первые млекопитающие с секвенированными геномами

Чтобы секвенировать геном, мы должны преодолеть некоторые практические препятствия. Самым большим препятствием является тот факт, что у биологов до сих пор нет технологии, позволяющей читать нуклеотиды генома от начала до конца так же, как вы читаете книгу. Лучшее, что они могут

сделать, – это секвенировать гораздо более короткие фрагменты ДНК, называемые **ридами** («reads»). Причины, по которым исследователи могут секвенировать только небольшие фрагменты ДНК, но не геномы целиком, особенно длинные, требуют отдельного обсуждения, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Краткая история технологий секвенирования ДНК**. В этой главе наша цель – превратить очевидный недостаток в полезный инструмент для пересборки генома.

Традиционный метод секвенирования геномов можно описать следующим образом. Исследователи берут небольшой образец ткани или крови, содержащий миллионы клеток с идентичной ДНК, используют биохимические методы для разбиения ДНК на фрагменты, а затем секвенируют эти фрагменты для получения ридов (рис. 3.4). Сложность в том, что исследователи не знают, из какого места в геноме взялись эти риды, и поэтому для реконструкции генома они должны использовать перекрывающиеся риды. Таким образом, сборка генома из его ридов, или **ассемблирование генома,** аналогична Задаче газет.

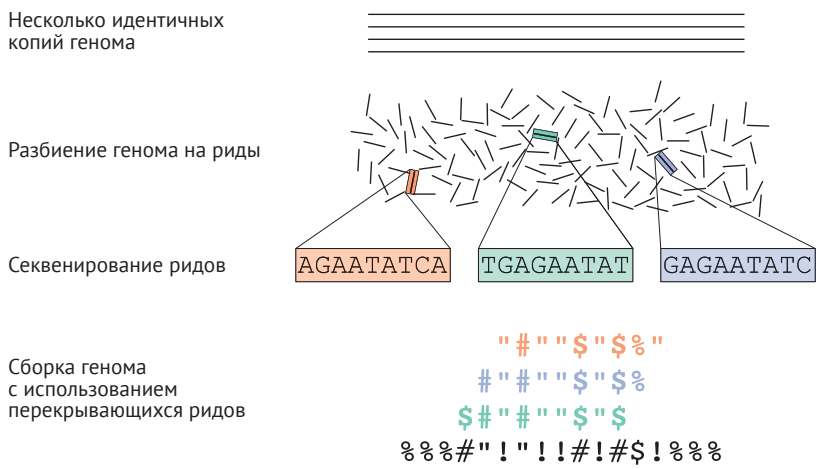


Рис. 3.4 При секвенировании ДНК множество идентичных копий генома разрываются в случайных местах для создания коротких ридов, которые затем секвенируются и пересобираются в нуклеотидную цепь генома

Несмотря на то что исследователи секвенировали множество геномов, такой гигантский геном, как геном *Amoeba dubia*, по-прежнему остается вне досягаемости современных технологий секвенирования. Вы можете подумать, что барьер для секвенирования такого генома является экспериментальным, но это не так; биологи могут легко генерировать достаточно ридов для анализа большого генома, но сборка этих ридов по-прежнему представляет собой вычислительную задачу невообразимой трудности.

Задача реконструкции строки

Сборка генома сложнее, чем вы думаете

Прежде чем мы представим вычислительную задачу, моделирующую сборку генома, мы обсудим несколько практических сложностей, которые делают сборку генома более сложной, чем Задача газет.

Во-первых, молекула ДНК является двухцепочечной, и у нас нет возможности *априори* узнать, из какой цепи взят данный рид, а это означает, что мы не можем знать, использовать ли нам рид или его комплемент при сборке конкретной цепи генома. Во-вторых, современные секвенсоры несовершенны, и считывания, которые они генерируют, часто содержат ошибки. Ошибки секвенирования усложняют сборку генома, потому что они не позволяют нам идентифицировать все перекрывающиеся риды. В-третьих, некоторые области генома могут быть вообще не охвачены ни одним из ридов, что делает невозможным полную реконструкцию всего генома целиком.

Поскольку риды, сгенерированные современными секвенсорами, часто имеют одинаковую длину, мы можем с уверенностью предположить, что все риды являются k-мерами с некоторым значением k. В первой части этой главы предполагается идеальная (и нереальная) ситуация, в которой все риды прочитаны из одной и той же цепи, не имеют ошибок и демонстрируют **идеальное покрытие**, так что каждая последовательная, неперекрывающая подстрока, k-мер генома, генерируется как рид. Позже мы покажем, как смягчить эти предположения для получения более реалистичных наборов данных.

Реконструкция строк из k-меров

Теперь мы готовы определить вычислительную задачу моделирования сборки генома. Для строки *Text* ее ***k*-мер-композиция** $Composition_k(Text)$ представляет собой набор всех подстрок (k-меров) строки *Text* (включая повторяющиеся k-меры). Например,

$$Composition_3(\text{TATGGGGTGC}) = \{\text{ATG, GGG, GGG, GGT, GTG, TAT, TGC, TGG}\}.$$

Обратите внимание, что мы перечислили k-меры в **лексикографическом порядке** (т. е. в том порядке, в котором они появляются в словаре), а не в порядке их появления в TATGGGGTGC. Мы сделали это, потому что, когда они генерируются, правильный порядок ридов неизвестен.

Задача восстановления строки: *сделайте k-мер-композицию из строки.*

Input: строка *Text* и целое число k.

Output: *Composition*$_k$(*Text*), где *k*-меры расположены в лексикографическом порядке.

Решение задачи композиции строк – простое упражнение, но для моделирования сборки генома нам нужно решить обратную задачу.

Задача реконструкции строки: *восстановите строку из ее k-мер-композиции.*

Input: целое число *k* и набор *Patterns k*-меров.

Output: строка *Text* с *k*-мер-композицией, идентичной *Patterns* (если такая строка существует).

Прежде чем мы попросим вас решить задачу реконструкции строки, давайте рассмотрим следующий пример 3-мер-композиции:

```
AAT ATG GTT TAA TGT
```

Самый естественный способ решить задачу реконструкции строк – это имитировать решение Задачи газет и «соединить» пару *k*-меров, если они перекрываются в *k* – 1 символах. В приведенном выше примере легко увидеть, что строка должна начинаться с TAA, потому что 3-мер не оканчивается на TA. Это означает, что следующий 3-мер в цепи должен начинаться с AA. Существует только один 3-мер, удовлетворяющий этому условию, AAT:

```
TAA
  AAT
```

В свою очередь, AAT может быть расширен только с помощью ATG, **который может быть расширен только с помощью** TGT и т. д., что приводит нас к реконструкции **TAATGTT**:

```
TAA
  AAT
    ATG
      TGT
        GTT
TAATGTT
```

Похоже, мы закончили с задачей реконструкции строк и можем перейти к следующей главе. Но для уверенности рассмотрим еще одну 3-мер-композицию:

```
AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT
```

Упражнение. Восстановите строку с помощью этой 3-мер-композиции.

Если мы снова начнем с TAA, то следующий 3-мер в цепи должен начинаться с AA, а такой 3-мер только один, AAT. В свою очередь, AAT может быть расширен только с помощью ATG:

```
TAA
  AAT
    ATG
TAATG
```

ATG может быть расширен либо с помощью TGC, либо TGG, либо TGT. Теперь мы должны решить, какой из этих 3-меров выбрать. Выбираем TGT:

```
TAA
  AAT
    ATG
      TGT
TAATGT
```

После TGT наш единственный выбор – GTT:

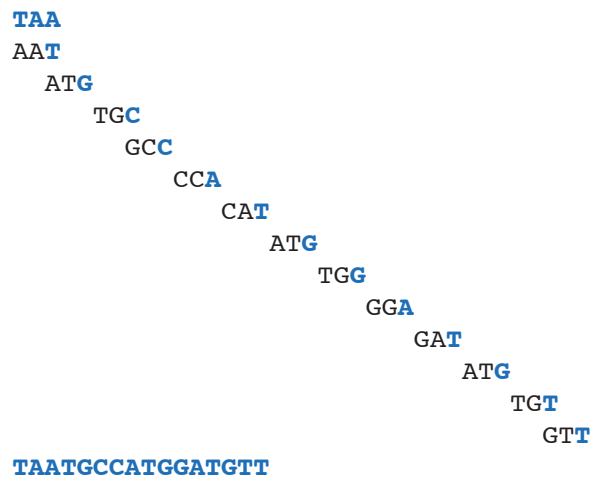
```
TAA
  AAT
    ATG
      TGT
        GTT
TAATGTT
```

К сожалению, сейчас мы застряли на GTT, потому что в композиции нет 3-меров, начинающихся с TT! Мы могли бы попытаться продолжить TAA влево, но в композиции нет 3-меров, оканчивающихся на TA.

Возможно, вы сами нашли эту ловушку и уже знаете, как из нее выбраться. Если вы хороший шахматист, то думаете на несколько шагов вперед и никогда не расширите ATG на TGT, пока не достигнете конца генома. Помня об этом, давайте сделаем шаг назад, вместо этого расширив ATG на TGC:

```
TAA
  AAT
    ATG
      TGC
TAATGC
```

Продолжая процесс, получаем следующую сборку:



Однако эта сборка неверна, потому что мы использовали только 14 из 15 3-меров композиции (мы опустили GGG), что сделало наш реконструированный геном на один нуклеотид короче.

Повторы усложняют сборку генома

Сложность сборки этого генома возникает из-за того, что ATG трижды повторяется в 3-мер-композиции, что дает нам три варианта выбора: TGG, TGC и TGT, с помощью которых можно расширить ATG. Повторяющиеся подстроки в геноме не представляют серьезной проблемы, когда у нас всего 15 ридов, но при миллионах ридов повторы значительно усложняют возможность «заглянуть вперед» и построить правильную сборку.

Если вы просмотрели **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Вероятности паттернов в строке** из главы 1, вы знаете, насколько маловероятно увидеть длинное повторение в случайно сгенерированной последовательности нуклеотидов. Вы также знаете, что настоящие геномы вовсе не случайны. Действительно, примерно 50 % генома человека состоит из повторов; например, **последовательность Alu** длиной примерно 300 нуклеотидов повторяется более миллиона раз, при этом каждый раз вставляется/удаляется/заменяется лишь несколько нуклеотидов (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Геном человека**).

Аналогией, иллюстрирующей сложность сборки генома с большим количеством повторов, является головоломка Triazzle® (рис. 3.5). Люди обычно собирают пазлы, соединяя соответствующие друг другу части. Однако каждая фигура в триазле соответствует более чем одной другой фигуре; на рис. 3.5 каждая лягушка появляется несколько раз. Если вы будете действовать небрежно, то,

скорее всего, вы состыкуете большинство частей, но не сможете подобрать остальные. И все же в Triazzle всего 16 фрагментов, что должно заставить нас задуматься о сборке генома из миллионов ридов.

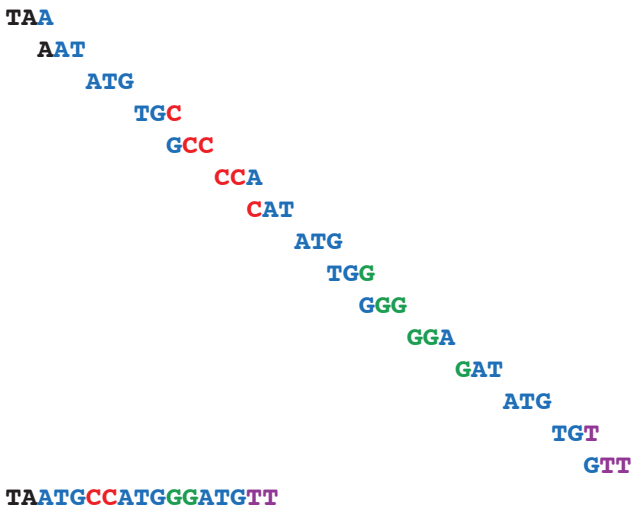
Рис. 3.5 Каждый триазл состоит всего из 16 частей, но сопровождается предупреждением: «Это сложнее, чем кажется!»

Упражнение. Разработайте стратегию сборки головоломки Triazzle.

Реконструкция строк как прогулка по графу перекрытий

От строки к графу

Повторы в геноме требуют какого-то способа заглянуть вперед, чтобы заранее увидеть правильную сборку. Возвращаясь к нашему предыдущему примеру, вы, возможно, уже обнаружили, что **TA**ATGCCATGGGATGTT является решением задачи реконструкции строки для набора 15 3-меров в последнем разделе, как показано ниже. Обратите внимание, что мы используем разные цвета для каждого интервала строки между вхождениями **ATG**.



ОСТАНОВИТЕСЬ и задумайтесь. Является ли это единственным решением задачи реконструкции строки для данного набора 3-меров?

На рис. 3.6 последовательные 3-меры в **TAATGCCATGGGATGTT** соединены друг с другом, образуя геномный путь.

Рис. 3.6 Пятнадцать 3-меров с цветовой кодировкой, составляющих **TAATGCCATGGGATGTT**, присоединяются к геномному пути в соответствии с их порядком в геноме

Задача реконструкции строки по пути генома: *реконструировать строку по пути ее генома.*

Input: строка k-меров $Pattern_1, \ldots, Pattern_n$ такая, что последние $k - 1$ символов $Pattern_i$ равны первым $k - 1$ символам $Pattern_{i+1}$ для $1 \leqslant i \leqslant n - 1$.

Output: строка *Text* длины $k + n - 1$ такая, что i-й k-мер в *Text* равен $Pattern_i$ (для $1 \leqslant i \leqslant n$).

Реконструировать геном по его геномному пути легко: по мере того как мы движемся слева направо, 3-меры «расшифровывают» **TAATGCCATGGGATGTT**, добавляя один новый символ в геном в каждом новом 3-мере. К сожалению, построение пути генома этой строки требует, чтобы мы знали геном заранее.

ОСТАНОВИТЕСЬ и задумайтесь. Могли бы вы построить путь генома, если бы знали только 3-мер-композицию генома?

В этой главе мы будем использовать термины **префикс** и **суффикс** для обозначения первых k – 1 нуклеотидов и последних k – 1 нуклеотидов k-мера соответственно. Например, *Prefix*(**TAA**) = **TA** и *Suffix*(**TAA**) = **AA**. Отметим, что суффикс 3-мера в пути генома равен префиксу следующего 3-мера в пути. Например, *Suffix*(**TAA**) = *Prefix*(**AAT**) = **AA** в пути генома для **TAATGCCATGGGATGTT**.

Это наблюдение предлагает метод построения пути генома строки по ее k-мер-композиции: мы будем использовать стрелку, чтобы соединить любой k-мер *Pattern* с k-мером *Pattern'*, если суффикс *Pattern* равен префиксу *Pattern'*.

ОСТАНОВИТЕСЬ и задумайтесь. Примените правило, которое мы только что описали, к 3-мер-композиции **TAATGCCATGGGATGTT**. Можете ли вы реконструировать геномный путь **TAATGCCATGGGATGTT**?

Если мы будем следовать правилу соединения двух 3-меров стрелкой каждый раз, когда суффикс одного равен префиксу другого, то мы соединим все последовательные 3-меры в **TAATGCCATGGGATGTT**, как показано на рис. 3.6. Однако, поскольку мы не знаем этот геном заранее, нам приходится соединять и многие другие пары 3-меров. Например, каждое из трех вхождений **ATG** должно быть связано с **TGC**, **TGG** и **TGT**, как показано на рис. 3.7.

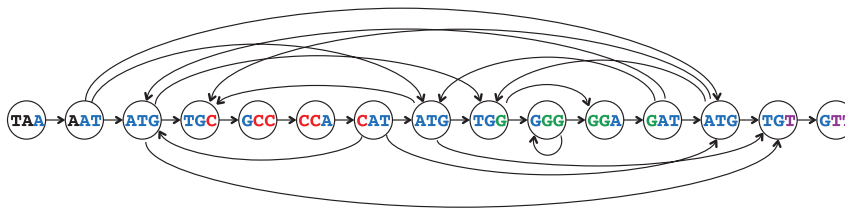


Рис. 3.7 Граф, показывающий все связи между узлами, представляющими 3-мер-композицию **TAATGCCATGGGATGTT**. Этот граф имеет 15 узлов и 28 ребер. Обратите внимание, что геном все еще можно расшифровать, пройдясь по горизонтальным ребрам от **TAA** до **GTT**

На рис. 3.7 представлен пример **графа** или сети **узлов**, соединенных **ребрами**. Этот конкретный граф является примером **ориентированного графа**, ребра которого имеют направление и представлены стрелками (в отличие от **неориентированного графа**, ребра которого не имеют направлений). Если вы не знакомы с графами, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Графы**.

Геном исчезает

Геном все еще можно наблюдать на графе на рис. 3.7, проследив горизонтальный путь от **TA**A к **GTT**. Но при секвенировании генома мы заранее не знаем, как правильно упорядочить риды. Поэтому мы упорядочим 3-меры лексикографически, что даст граф перекрытий, показанный на рис. 3.8. Геномный путь исчез!

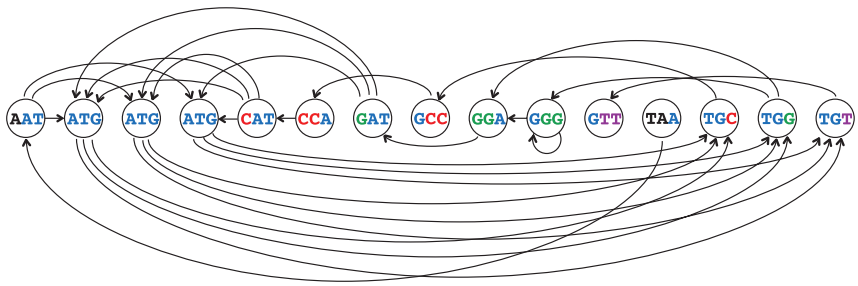


Рис. 3.8 Тот же граф, что и на рис. 3.7, с 3-мерами, упорядоченными лексикографически. Путь через граф, представляющий правильную сборку, теперь труднее увидеть

Путь генома мог исчезнуть для невооруженного глаза, но он все равно должен быть, так как мы просто переставили узлы графа. Действительно, на рис. 3.9 (вверху) показан путь генома, представляющий строку **TA**ATGCCATGGGATGTT. Однако, если бы мы дали вам этот граф в начале, вам нужно было бы найти путь через граф, который проходит каждый узел ровно один раз; такой путь «объясняет» все 3-меры в 3-мер-композиции генома. Хотя найти такой путь в настоящее время так же сложно, как попытаться собрать геном вручную, тем не менее граф дает нам хороший способ визуализировать взаимосвязи перекрытий между ридами.

ОСТАНОВИТЕСЬ и задумайтесь. Можно ли реконструировать любые другие строки, следуя по пути, проходящему через все узлы на рис. 3.8?

Чтобы обобщить построение графа на рис. 3.8 для произвольного набора *k*-меров *Patterns*, мы формируем узел для каждого *k*-мера в *Patterns* и соединяем *k*-меры *Pattern* и *Pattern'* направленным ребром, если *Suffix*(*Pattern*) = *Prefix*(*Pattern'*). Результирующий граф называется графом перекрытия этих *k*-меров, обозначаемым *Overlap*(*Patterns*).

Задача графа перекрытия: *постройте граф перекрытия набора k-меров.*

Input: набор *k*-меров *Patterns*.
Output: граф перекрытия *Overlap*(Patterns).

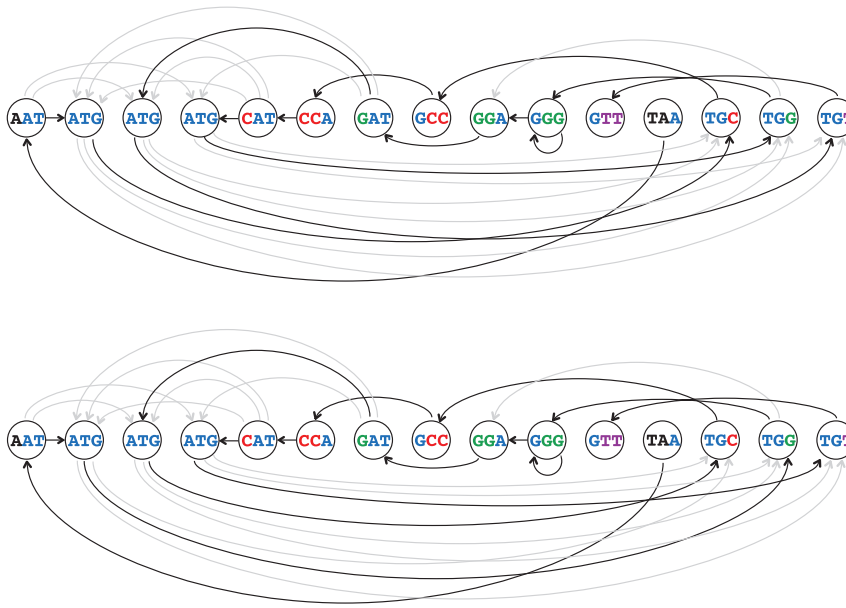


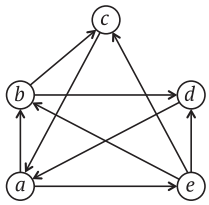
Рис. 3.9 (Вверху) Путь генома, обозначающий **TAATGCCATGGGATGTT**, выделен на графе перекрытий. (Внизу) Другой путь Гамильтона в графе перекрытий описывает геном **TAATGGGATGCCATGTT**. Эти два генома отличаются заменой позиций **CC** и **GG**, но имеют одинаковый состав 3-меров

Два способа представления графов

Если вы никогда раньше не работали с графами, вам может быть интересно, как представить графы в ваших программах. Чтобы сделать небольшое отступление от нашего обсуждения сборки генома, рассмотрим граф на рис. 3.10 (вверху слева); мы можем перемещаться по узлам этого графа, не изменяя его. (Для другого примера: графы на рис. 3.7 и 3.8 одинаковы). В результате, когда мы представляем граф с помощью вычислений, единственная информация, которую нам нужно хранить, – это пара узлов, которые соединяет каждое ребро.

Есть два стандартных способа представления графа. Для ориентированного графа с n узлами **матрица смежности (матрица соседства)** $n \times n$ ($A_{i,j}$) определяется следующим правилом: $A_{i,j} = 1$, если направленное ребро соединяет узел i с узлом j, и $A_{i,j} = 0$ в противном случае. Другой, более экономичный способ представления графа – использование **списка смежности**, для которого мы просто перечисляем все узлы, соединенные с каждым узлом (рис. 3.10).

Граф



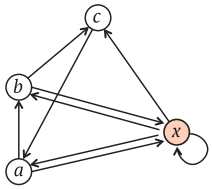
Матрица смежности

	a	*b*	*c*	*d*	*e*
a	0	1	0	0	1
b	0	0	1	1	0
c	1	0	0	0	0
d	1	0	0	0	0
e	0	1	1	1	0

Список смежности

a смежно *b* и *e*
b смежно *c* и *d*
c смежно *a*
d смежно *a*
e смежно *b*, *c*, и *d*

Граф



Матрица смежности

	a	*b*	*c*	*x*
a	0	1	0	1
b	0	0	1	1
c	1	0	0	0
x	1	1	1	1

Список смежности

a смежно *b* и *x*
b смежно *c* и *x*
c смежно *a*
x смежно *a*, *b*, *c*, и *x*

Рис. 3.10 (Вверху) Граф с пятью узлами и девятью ребрами, за которыми следуют матрица смежности и список смежности. (Внизу) Граф, полученный путем склеивания узлов *d* и *e* в один узел *x* вместе с матрицей смежности и списком смежности нового графа

Гамильтоновы пути и универсальные строки

Теперь мы знаем, что для решения задачи реконструкции строк мы ищем путь в графе перекрытий, который посещает каждый узел ровно один раз. Путь в графе, по одному разу посещающий каждый узел, называется гамильтоновым путем в честь ирландского математика Уильяма Гамильтона (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Игра «Икосиан»**). Как показано на рис. 3.9, в графе может быть несколько гамильтоновых путей.

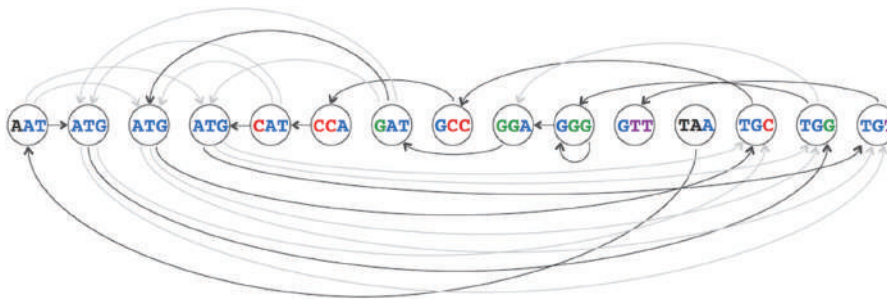


Рис. 3.11 В дополнение к гамильтонову пути, который реконструирует **TAATGCCATGGGATGTT**, другой гамильтонов путь в графе перекрытий описывает геном **TAATGGGATGCCATGTT**. Эти два генома отличаются заменой положений **CC** и **GG**, но имеют одинаковую композицию 3-меров

Задача гамильтонова пути: *построить гамильтонов путь в графе.*

Input: ориентированный граф.

Output: путь, проходящий через каждую вершину графа ровно один раз (если такой путь существует).

Мы пока не просим вас решить задачу о гамильтоновом пути, так как неясно, как разработать для нее эффективный алгоритм. Вместо этого мы хотим, чтобы вы познакомились с Николаасом де Брюйном, голландским математиком. В 1946 году де Брюйн интересовался решением чисто теоретической задачи, описанной следующим образом. *Двоичная строка* – это строка, состоящая только из 0 и 1; бинарная строка является ***k*-универсальной**, если она содержит каждый двоичный *k*-мер ровно один раз. Например, 0001110100 является 3-универсальной строкой, поскольку она содержит каждый из восьми двоичных 3-меров (000, 001, 011, 111, 110, 101, 010, и 100) ровно один раз.

Поиск *k*-универсальной строки можно свести к решению задачи реконструкции строки, когда композиция *k*-меров представляет собой совокупность всех бинарных *k*-меров. Таким образом, нахождение *k*-универсальной строки эквивалентно нахождению гамильтонова пути в графе перекрытий, сформированном на всех бинарных *k*-мерах (рис. 3.12). Хотя гамильтонов путь на рис. 3.12 можно легко найти вручную, де Брюйна интересовало построение *k*-универсальных цепей для произвольных значений *k*. Например, чтобы найти универсальную строку для $k = 20$, вам придется рассмотреть граф с более чем миллионом узлов. Совершенно непонятно, как найти гамильтонов путь в таком огромном графе и вообще существует ли такой путь!

Вместо поиска гамильтоновых путей в огромных графах де Брюйн разработал совершенно другой (и несколько неинтуитивный) способ представления композиции *k*-меров с помощью графа. Позже в этой главе мы узнаем, как он использовал этот метод для построения универсальных строк.

Упражнение. Постройте 4-универсальную строку.

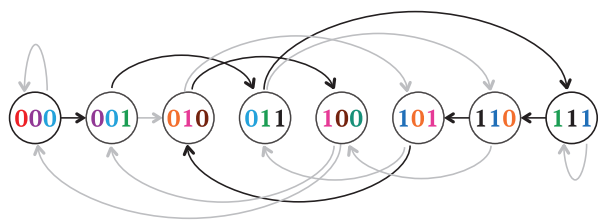


Рис. 3.12 Гамильтонов путь (соединяющий узлы 000 и 100), выделенный на графе перекрытий всех бинарных 3-меров. Этот путь представляет собой 3-универсальную двоичную строку 0001110100

Другой граф для реконструкции строк

Склеивание узлов и графы де Брюйна

Снова представим геном **TA**ATGCCATGGGATGTT в виде последовательности его 3-меров:

TAA AAT ATG TGC GCC CCA CAT ATG TGG GGG GGA GAT ATG TGT GTT

На этот раз вместо того, чтобы приписывать эти 3-меры *узлам*, мы припишем их *ребрам*, как показано на рис. 3.13. Вы можете еще раз реконструировать геном, следуя по этому пути слева направо, добавляя по одному новому нуклеотиду на каждом шагу. Поскольку каждая пара последовательных ребер представляет собой последовательные 3-меры, которые перекрываются в двух нуклеотидах, мы пометим каждый узел этого графа 2-мером, представляющим перекрывающиеся нуклеотиды, общие для ребер по обе стороны от узла. Например, узел с входящим краем CAT и исходящим краем ATG помечен как AT.

Рис. 3.13 Геном **TA**ATGCCATGGGATGTT, представленный в виде пути с ребрами (а не узлами), помеченными 3-мерами, и узлами, помеченными 2-мерами

Ничего здесь не кажется новым, пока мы не начнем **склеивать** одинаково помеченные узлы. На рис. 3.14 (вверху) мы приближаем три узла AT все ближе и ближе друг к другу, пока они не будут склеены в один узел. Обратите внимание, что есть также три узла, помеченные TG, которые мы склеиваем на рис. 3.14 (в середине). Наконец, мы склеиваем вместе два узла, помеченные GG (GG и GG), как показано на рис. 3.14 (внизу), что дает ребро особого типа, называемое **петлей (loop)**, соединяющей GG с самим собой.

Количество узлов в полученном графе (рис. 3.14 (внизу справа)) уменьшилось с 16 до 11, а количество ребер осталось прежним. Этот граф называется **графом де Брюйна** для **TA**ATGCCATGGGATGTT и обозначается $Debruijn_3$(**TA**ATGCCATGGGATGTT). Обратите внимание, что этот граф де Брюйна имеет три разных ребра, соединяющих AT с TG, что представляет собой три копии повторяющегося ATG.

В общем, для заданного генома *Text*, $PathGraph_k(Text)$ – это путь, состоящий из |*Текст*| – k + 1 ребер, где i-е ребро этого пути помечено i-м k-мером в *Text*, а i-й узел пути помечен i-м (k – 1)-мером в *Text*. Граф $Debruijn_k(Text)$ формируется путем склеивания одинаково помеченных узлов в $PathGraph_k(Text)$.

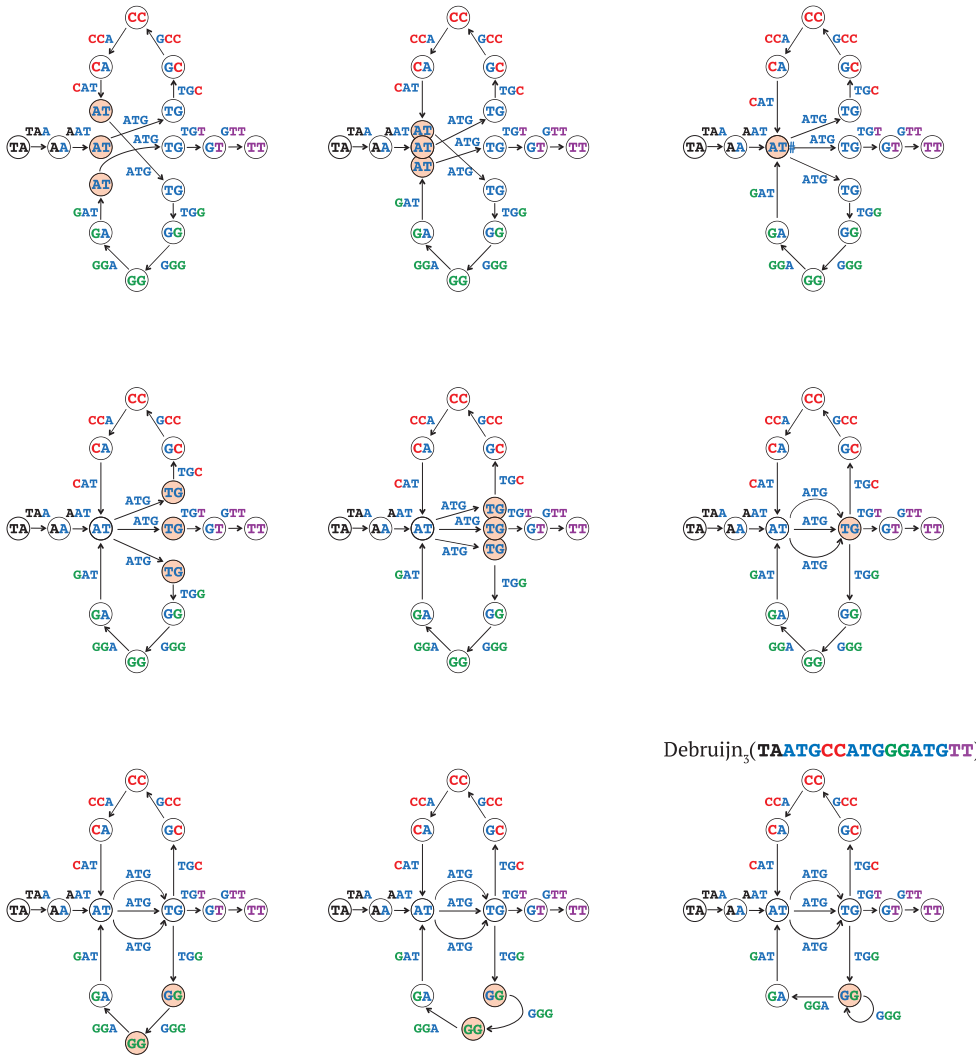


Рис. 3.14 (Вверху) Приведение трех узлов, обозначенных **AT** на рис. 3.13, ближе (слева) и ближе (посередине) друг к другу, чтобы в конечном итоге склеить их в один узел (справа). (В середине) Приведение трех узлов, помеченных **TG**, ближе (слева) и ближе (посередине) друг к другу, чтобы в конечном итоге склеить их в один узел (справа). (Внизу) Приведение двух узлов с пометкой GG ближе (слева) и ближе (посередине) друг к другу, чтобы в конечном итоге склеить их в один узел (справа). Путь с 16 узлами с рис. 3.13 был преобразован в граф *Debruijn*$_3$(**TAATGCCATGGGATGTT**) с 11 узлами

Граф де Брюйна из задачи о строках: *построить граф де Брюйна строки.*

Input: строка *Text* и целое число *k*.
Output: $Debruijn_k(Text)$.

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующие вопросы.

1. Если бы мы дали вам граф де Брюйна $Debruijn_k(Text)$, но не дали *Text*, смогли бы вы реконструировать *Text*?
2. Постройте графы де Брюйна $Debruijn_2(Text)$, $Debruijn_3(Text)$ и $Debruijn_4(Text)$ для *Text* = **TA**ATGCCATGGGATGTT. На что вы обратили внимание?
3. Как граф $Debruijn_3$(**TA**ATGCCATGGGATGTT) соотносится с $Debruijn_3$(**TA**ATGGGATGCCATGTT)?

ЗАРЯДНАЯ СТАНЦИЯ: Влияние склеивания на матрицу смежности. На рис. 3.10 (внизу) показано, как операция склеивания влияет на матрицу смежности и список смежности графа. Изучите эту зарядную станцию, чтобы увидеть, как работает склейка для графа де Брюйна.

Прогулка по графу де Брюйна

Эйлеровы пути

Несмотря на то что мы склеили узлы, чтобы сформировать граф де Брюйна, мы не изменили его ребра, поэтому путь от **TA** к **TT**, реконструирующий геном, все еще скрыт в $Debruijn_3$(**TA**ATGCCATGGGATGTT) (рис. 3.15), хотя этот путь стал «запутанным» после склейки. Таким образом, решение задачи реконструкции строк сводится к поиску пути в графе де Брюйна, который посещает каждое *ребро* ровно один раз. Такой путь называется **эйлеровым путем** в честь великого математика Леонарда Эйлера.

Задача эйлерова пути: *построить эйлеров путь в графе.*

Input: ориентированный граф.
Output: путь, проходящий через каждое ребро в графе ровно один раз (если такой путь существует).

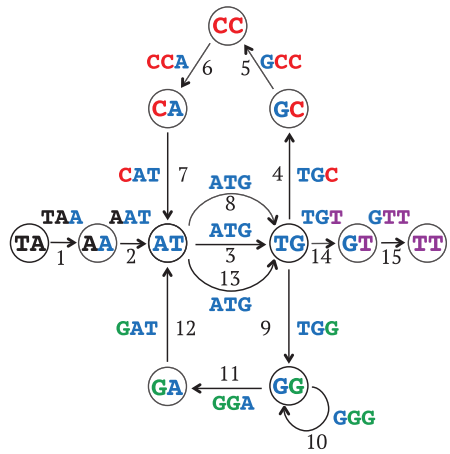


Рис. 3.15 Путь от **TA** к **TT**, определяющий геном **TA**ATGCCATGGGATGTT, «запутался» в графе де Брюйна. Нумерация пятнадцати ребер пути указывает на эйлеров путь, реконструирующий геном

Теперь у нас есть альтернативный способ решения задачи реконструкции строк, который сводится к поиску эйлерова пути в графе де Брюйна. Но погодите – чтобы построить граф де Брюйна генома, мы склеили узлы $PathGraph_k(Text)$. Однако для построения этого графа нам нужно знать правильный порядок k-меров в *Text*!

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы построить $Debruijn_k(Text)$, если не знаете *Text*, но знаете его k-мер-композицию?

Другой способ построения графов де Брюйна

На рис. 3.16 (вверху) представлена 3-мер-композиция **TA**ATGCCATGGGATGTT в виде графа композиции $CompositionGraph_3$(**TA**ATGCCATGGGATGTT). Как и в графе де Брюйна, каждый 3-мер соответствует направленному ребру, при этом его префикс обозначает первый узел ребра, а его суффикс – второй узел ребра. Однако ребра этого графа изолированы, а это означает, что никакие два ребра не имеют общего узла.

ОСТАНОВИТЕСЬ и задумайтесь. Для *Text* = **TA**ATGCCATGGGATGTT склейте узлы с одинаковыми метками в $CompositionGraph_3(Text)$. Чем полученный граф отличается от графа $Debruijn_3(Text)$, полученного при склеивании узлов с одинаковыми метками в $PathGraph_3(Text)$?

На рис. 3.16 (вверху) показано, как изменяется $CompositionGraph_3(Text)$ после склеивания узлов с одинаковыми метками для *Text* = **TA**ATGCCATGGGATGTT. Эти операции склеивают пятнадцать изолированных ребер $CompositionGraph_3(Text)$ в путь $PathGraph_3(Text)$. Последующие операции склеивания выполняются точно так же, как когда мы склеивали узлы $PathGraph_3(Text)$, что приводит к $Debruijn_3(Text)$. Таким образом, мы можем построить граф де Брюйна из 3-мер-композиции этого генома, не зная генома!

ОСТАНОВИТЕСЬ и задумайтесь. На рис. 3.16 мы идентифицировали **ATG** и **TGC** как перекрывающиеся 3-меры. На самом деле, поскольку геном неизвестен, мы не знаем, что следует за **ATG** – **TGC**, **TGG** или **TGT**. Что произошло бы, если бы вместо этого мы идентифицировали **ATG** и **TGG** как перекрывающиеся 3-меры? Убедитесь, что конечным результатом будет тот же граф, что и на рис. 3.15.

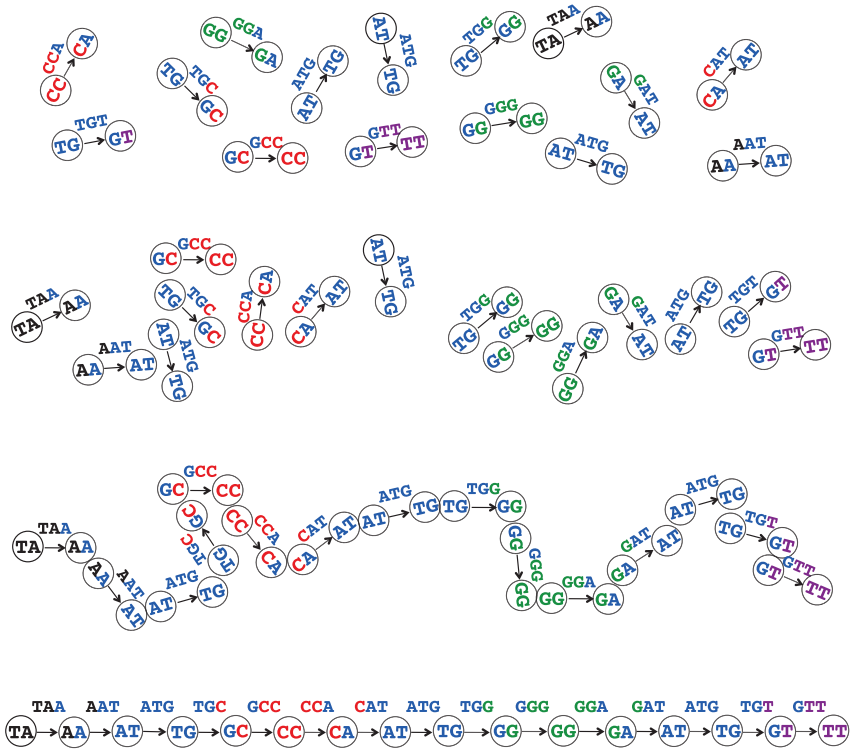


Рис. 3.16 Склеивание нескольких узлов с одинаковыми метками преобразует граф $CompositionGraph_3$(TAATGCCATGGGATGTT) (вверху) в граф $PathGraph_3$(**TA**ATGCCATGGGATGTT) (внизу). Склеивание всех одинаково помеченных узлов дает $Debruijn_3$(**TA**ATGCCATGGGATGTT) из рис. 3.15

Для произвольной строки *Text* мы определяем $CompositionGraph_k(Text)$ как граф, состоящий из $|Text| - k + 1$ изолированных ребер, где ребра помечены *k*-мерами в *Text*; каждое ребро, помеченное ребром *k*-мера, соединяет узлы, помеченные префиксом и суффиксом этого *k*-мера.

Граф $CompositionGraph_k(Text)$ – это просто набор изолированных ребер, представляющих *k*-меры в *k*-мер-композиции *Text*, а это означает, что мы можем построить $CompositionGrap_k(Text)$ из *k*-мер-композиции *Text*. Склеивание узлов с одинаковыми метками в $CompositionGraph_k(Text)$ дает *Debruijnk*(*Text*).

Имея произвольный набор *k*-меров *Patterns* (где некоторые *k*-меры могут появляться несколько раз), мы определяем *CompositionGraph*(*Patterns*) как граф с |*Patterns*| изолированных ребер. Каждое ребро помечено *k*-мером из *Patterns*, а начальный и конечный узлы ребра помечены префиксом и суффиксом *k*-мера, помечающего это ребро. Затем мы определяем *DeBruijn*(*Patterns*) путем склеивания одинаково помеченных узлов в *CompositionGraph*(*Patterns*), что дает следующий алгоритм.

DeBruijn(*Patterns*)
    представьте каждый *k*-мер в *Patterns* как изолированное ребро между его префиксом и суффиксом
    склейте все узлы с одинаковыми метками, получив граф *DeBruijn*(Patterns)
    return *DeBruijn*(*Patterns*)

Построение графов де Брюйна из композиции k-меров

Построение графа де Брюйна путем склеивания узлов с одинаковыми метками поможет нам позже, когда мы будем обобщать понятие графа де Брюйна для других приложений. Теперь мы опишем еще один полезный способ построения графов де Брюйна без склеек.

В наборе *k*-меров *Patterns* все узлы $Debruijn_k(Pattern)$ являются уникальными (k – 1)-мерами, встречающимися как префикс или суффикс 3-меров в *Pattern*. Например, скажем, нам дан следующий набор 3-меров:

AAT ATG ATG ATG CAT CCA GAT GCC GGA GGG GTT TAA TGC TGG TGT

Тогда набор из одиннадцати *уникальных* 2-меров, встречающихся в качестве префикса или суффикса в этой коллекции, выглядит следующим образом:

AA AT CA CC GA GC GG GT TA TG TT

Для каждого *k*-мера в *Patterns* мы соединяем его префиксный узел с его суффиксным узлом направленным ребром, чтобы получить *DeBruijn*(*Patterns*). Вы можете убедиться, что этот процесс создает тот же самый граф де Брюйна, с которым мы работали (рис. 3.17).

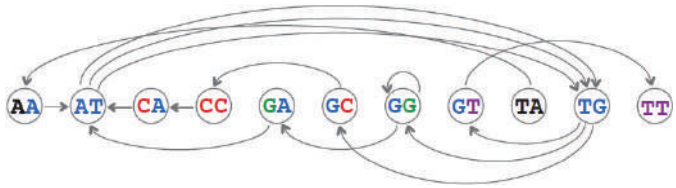


Рис. 3.17 Приведенный выше граф де Брюйна такой же, как и граф на рис. 3.15, хотя и нарисован по-другому

Задача построения графа де Брюйна из *k*-меров: *постройте граф де Брюйна набора k-меров.*

Input: набор *k*-меров *Patterns*.
Output: *Debruijn*(*Patterns*).

Графы де Брюйна в сравнении с графами перекрытия

Теперь у нас есть два способа решения задачи реконструкции строки. Мы можем либо найти гамильтонов путь в графе перекрытий, либо найти эйлеров путь в графе де Брюйна (рис. 3.18). Возможно, ваш внутренний голос уже начал жаловаться: *стоило ли мне тратить время на изучение двух очень мало отличающихся способов решения одной и той же задачи*? В конце концов, мы изменили только одно слово в формулировках гамильтоновой и эйлеровой задач о путях: от поиска пути, проходящего через каждый *узел* ровно один раз, до поиска пути, проходящего через каждое *ребро* ровно один раз.

ОСТАНОВИТЕСЬ и задумайтесь. С каким графом вы бы предпочли работать, с графом перекрытий или с графом де Брюйна?

Мы предполагаем, что вы, вероятно, предпочтете работать с графом де Брюйна, так как он меньше. Однако это было бы неправильной причиной предпочтения одного графа другому. В случае реальных задач сборки оба графа будут иметь миллионы узлов, поэтому все, что имеет значение, – это найти *эффективный алгоритм* реконструкции генома. Если мы сможем найти эффективный алгоритм для задачи о гамильтоновых путях, но не для задачи об эйлеровых путях, тогда вам следует выбрать граф перекрытий, даже если он выглядит более сложным.

Выбор между этими двумя графами является ключевым решением проблемы, обсуждаемой в этой главе. Чтобы помочь вам принять это решение, мы попросим вас сесть на борт нашей биоинформационной машины времени и совершить экскурсию в XVIII век.

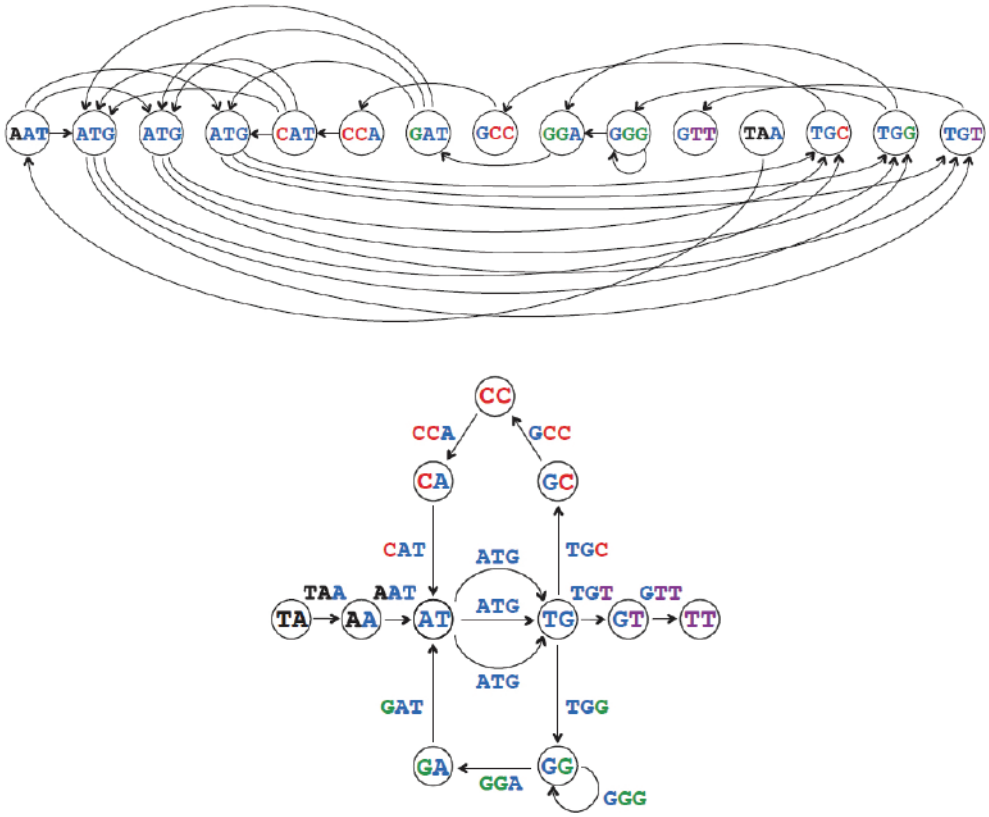


Рис. 3.18 Граф перекрытия (вверху) и граф де Брюйна (внизу) для одного и того же набора 3-меров

Семь мостов Кенигсберга

Наше место назначения – 1735 год, прусский город Кенигсберг. Этот город, который сегодня называется Калининградом, Россия, ранее занимал оба берега реки Прегель, а также два речных острова; семь мостов соединяли эти четыре разные части города, как показано на рис. 3.19 (вверху). Жители Кенигсберга любили гулять и задавались простым вопросом: можно ли выйти из своего дома, пройти по каждому мосту ровно один раз и вернуться домой? Этот вопрос стал известен как **задача кенигсбергских мостов**.

Упражнение. Существует ли решение задачи кенигсбергских мостов?

В 1735 году Леонард Эйлер нарисовал граф на рис. 3.19 (внизу), который мы называем «граф *Кенигсберг*»; узлы этого графа представляют четыре сектора города, а его ребра представляют семь мостов, соединяющих разные сектора. Обратите внимание, что ребра графа *Кенигсберг* ненаправлены, а это означает, что их можно пройти в любом направлении.

ОСТАНОВИТЕСЬ и задумайтесь. Переопределите задачу кенигсбергских мостов как вопрос о графе *Кенигсберг*.

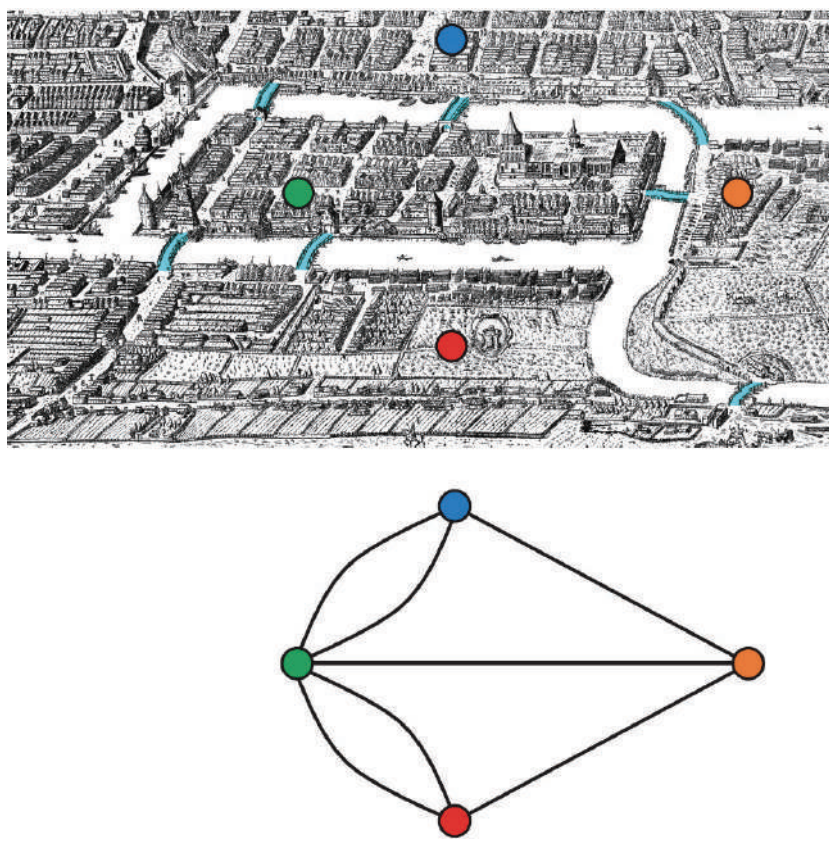


Рис. 3.19 (Вверху) Карта Кенигсберга, адаптированная из иллюстрации Иоахима Беринга 1613 года. Город состоял из четырех секторов, обозначенных синими, красными, желтыми и зелеными точками. Семь мостов, соединяющих разные части города, выделены голубым цветом, чтобы их было легче увидеть. (Внизу) Граф *Кенигсберг*

Мы уже определили эйлеров путь как путь в графе, проходящий через каждое ребро графа ровно один раз. Цикл, который проходит каждое ребро графа ровно один раз, называется эйлеровым циклом, и мы говорим, что граф, содер-

жащий такой цикл, является эйлеровым. Заметим, что эйлеров цикл в графе *Кенигсберг* немедленно обеспечил бы жителей города той прогулкой, которую они хотели. Теперь мы можем переопределить задачу кенигсбергских мостов как пример следующей более общей задачи.

Задача эйлерова цикла: *найдите эйлеров цикл в графе.*

Input: граф.

Output: эйлеров цикл в этом графе, если такой путь существует.

Эйлер решил задачу о кенигсбергских мостах, показав, что *ни один* обход не может пройти по каждому мосту ровно один раз (т. е. граф *Кенигсберг* не является эйлеровым), что вы, возможно, уже поняли сами. Однако его реальный вклад и причина, по которой его считают основателем **теории графов**, области математики, которая процветает и сегодня, заключается в том, что он доказал теорему, определяющую, когда граф будет иметь эйлеров цикл. Из его теоремы немедленно следует эффективный алгоритм построения эйлерова цикла в любом эйлеровом графе, имеющем даже миллионы ребер. Кроме того, этот алгоритм может быть легко расширен до алгоритма построения эйлерова *пути* (в графе, имеющем такой путь), что позволит нам решить задачу реконструкции строки с помощью графа де Брюйна.

С другой стороны, оказывается, что еще никому не удавалось найти эффективный алгоритм, решающий задачу гамильтоновых путей. Поиск такого алгоритма или доказательство того, что эффективного алгоритма для этой задачи не существует, лежит в основе одного из самых фундаментальных вопросов информатики, до сих пор остающегося без ответа.

Компьютерщики классифицируют алгоритм как **полиномиальный**, если время его выполнения может быть ограничено полиномом в зависимости от длины входных данных (То есть ограниченным степенной функцией. – *Прим. ред.*). С другой стороны, алгоритм является **экспоненциальным**, если время его выполнения зависит от длины входных данных экспоненциально.

Упражнение. Классифицируйте алгоритмы, с которыми мы столкнулись в главе 1, как полиномиальные или экспоненциальные.

Хотя алгоритм Эйлера является полиномиальным, задача гамильтоновых путей принадлежит к особому классу задач, для которых все попытки разработать полиномиальный алгоритм потерпели неудачу (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Разрешимые и неразрешимые задачи**). Однако вместо того, чтобы пытаться решить задачу, которая десятилетиями ставила ученых в тупик, мы отложим в сторону граф перекрытий и вместо этого сосредоточимся на подходе к сборке генома на основе графа де Брюйна.

В течение первых двух десятилетий после изобретения методов секвенирования ДНК биологи собирали геномы, используя графы перекрытий, поскольку они еще не знали, что задача кенигсбергских мостов содержит ключ к сборке ДНК (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: от Эйлера до Гамильтона и де Брюйна**). Действительно, графы перекрытий использовались для сборки генома человека, но биоинформатикам потребовалось некоторое время, чтобы понять, что граф де Брюйна, впервые построенный для решения чисто теоретической задачи, имеет отношение к сборке генома. Более того, когда граф де Брюйна был впервые применен в биоинформатике, он считался экзотической математической концепцией с ограниченным практическим применением. Сегодня использование графа де Брюйна стало доминирующим методом сборки генома.

Теорема Эйлера

Теперь мы рассмотрим метод Эйлера для решения задачи эйлерова цикла. Эйлер работал с неориентированными графами, такими как как граф *Кенигсберг*, но мы рассмотрим аналог его алгоритма для ориентированных графов, чтобы его метод можно было применить к сборке генома.

Представим себе муравья, которого мы назовем Лео, ползущего по ребрам эйлерова цикла. Каждый раз, когда Лео, идя по ребру, входит в узел графа, он может выйти из этого узла по другому, неиспользованному ребру. Таким образом, чтобы граф был эйлеровым, количество входящих ребер в любом узле должно быть равно количеству исходящих ребер в этом узле. Мы определяем степени входа и выхода узла v (обозначаемые $In(v)$ и $Out(v)$ соответственно) как количество ребер, ведущих в v и из него. Узел v **сбалансирован**, если $In(v) = Out(v)$, и граф **сбалансирован**, если все его узлы сбалансированы. Поскольку Лео всегда должен иметь возможность покинуть узел по неиспользуемому ребру, любой эйлеров граф должен быть сбалансирован. На рис. 3.20 показаны сбалансированный и несбалансированный графы.

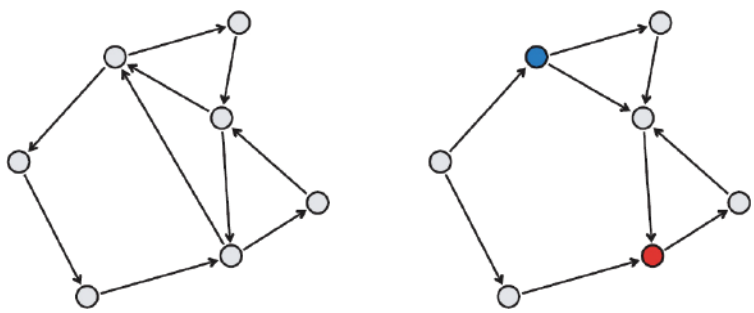


Рис. 3.20 Сбалансированный (слева) и несбалансированный (справа) ориентированные графы. Для (несбалансированного) синего узла v $In(v) = 1$ и $Out(v) = 2$, тогда как для (несбалансированного) красного узла ω $In(\omega) = 2$ и $Out(\omega) = 1$

ОСТАНОВИТЕСЬ и задумайтесь. Теперь мы знаем, что каждый эйлеров граф сбалансирован; но является ли каждый сбалансированный граф эйлеровым?

Граф на рис. 3.21 является сбалансированным, но не эйлеровым, поскольку он несвязен, а это означает, что некоторые узлы не могут быть достигнуты из других узлов. В любом несвязном графе невозможно найти эйлеров цикл. Напротив, мы говорим, что ориентированный граф является **сильно связным**, если из любого узла можно добраться до любого другого узла.

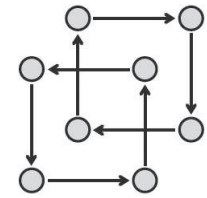


Рис. 3.21 Сбалансированный несвязный граф

Теперь мы знаем, что эйлеров граф должен быть сбалансированным и сильно связным. Теорема Эйлера утверждает, что этих двух условий достаточно, чтобы гарантировать, что произвольный граф является эйлеровым. В результате это означает, что мы можем определить, является ли граф эйлеровым, даже не рисуя циклы.

Теорема Эйлера. Каждый сбалансированный сильно связный ориентированный граф является эйлеровым.

Доказательство. Пусть *Graph* – произвольный сбалансированный и сильно связный ориентированный граф. Чтобы доказать, что *Graph* имеет эйлеров цикл, поместите Лео в любой узел v_0 графа (зеленый узел на рис. 3.22) и позвольте ему случайным образом пройти по графу при условии, что он не может пройти одно и то же ребро дважды.

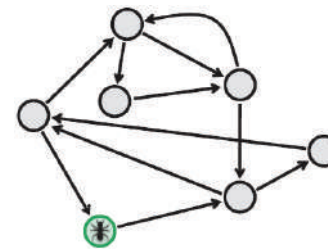


Рис. 3.22 Лео начинает с зеленого узла v_0 и проходит через сбалансированный и сильно связный граф

Если бы Лео невероятно повезло – или он был бы гением, – то он прошел бы каждое ребро ровно один раз и вернулся бы обратно в v_0. Однако велика вероятность, что он «застрянет» где-нибудь, прежде чем завершит эйлеров цикл, а это означает, что он достигнет узла и не найдет неиспользуемых ребер, выходящих из этого узла.

ОСТАНОВИТЕСЬ и задумайтесь. Где находится Лео, когда он застревает? Может ли он застрять в любом узле графа или только в определенных узлах?

Оказывается, единственный узел, в котором Лео может застрять, – это стартовый узел v_0! Причина в том, что *Graph* сбалансирован: если Лео войдет в любой узел, кроме v_0 (через входящее ребро), то он всегда сможет выйти через не-

используемое исходящее ребро. Единственным исключением из этого правила является начальный узел v_0, так как Лео использовал одно из исходящих ребер v_0 на своем первом проходе. Теперь, поскольку Лео вернулся в v_0, результатом его прогулки стал цикл, который мы называем $Cycle_0$ (рис. 3.23 (слева)).

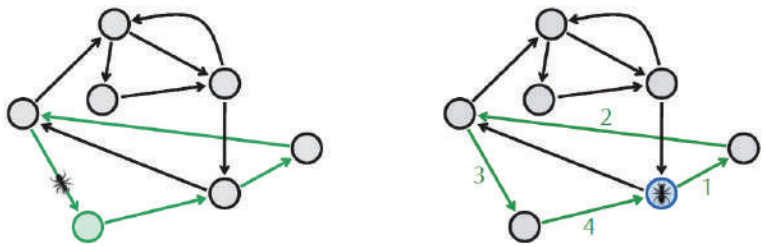


Рис. 3.23 (Слева) Лео создает цикл $Cycle_0$ (образованный зелеными ребрами), когда он застревает в зеленом узле v_0. В этом случае он еще не посетил все ребра графа. (Справа) Начиная с нового узла v_1 (показанного синим цветом), Лео сначала проходит по $Cycle_0$, возвращаясь к v_1. Обратите внимание, что синий узел v_1, в отличие от зеленого узла v_0, имеет неиспользуемые исходящие и входящие ребра

ОСТАНОВИТЕСЬ и задумайтесь. Есть ли способ дать Лео другие инструкции, чтобы он выбрал более длинный путь по графу, прежде чем застрянет?

Как мы уже упоминали, если $Cycle_0$ эйлеров, то мы закончили наше путешествие. В противном случае, поскольку *Graph* сильно связный, некоторый узел в $Cycle_0$ должен иметь неиспользуемые ребра, входящие в него и выходящие из него (почему?). Назвав этот узел v_1, мы просим Лео начать с v_1 вместо v_0 и пройти $Cycle_0$ (таким образом вернувшись к v_1), как показано на рис. 3.23 (справа).

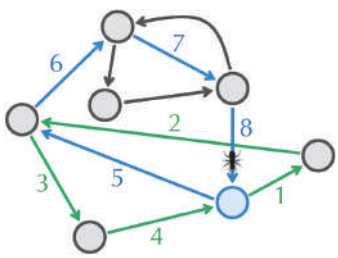


Рис. 3.24 Пройдя ранее построенный зеленый цикл $Cycle_0$, Лео продолжает идти и в конце концов создает более длинный цикл $Cycle_1$, состоящий из зеленого и синего циклов, объединенных в единый цикл

Лео, вероятно, раздражен тем, что мы попросили его пройти точно такой же цикл, поскольку, как и прежде, он в конце концов вернется к v_1, узлу, с которого он начал. Однако теперь есть неиспользованные ребра, начинающиеся в этом узле, и поэтому он может продолжать идти от v_1, используя каждый раз новое ребро. Тот же аргумент, который мы использовали ранее, подразумевает, что Leo в конечном итоге должен застрять на v_1. Результатом прогулки Лео является новый цикл $Cycle_1$ (рис. 3.24), который длиннее, чем $Cycle_0$.

Если $Cycle_1$ является эйлеровым циклом, то Лео выполнил свою работу. В противном

случае мы выбираем узел v_2 в $Cycle_1$, в который входят и выходят неиспользуемые ребра (красный узел на рис. 3.25 (слева)). Начиная с v_2, мы просим Лео пройти $Cycle_1$ и вернуться к v_2, как показано на рис. 3.25 (слева). После этого он будет случайным образом ходить, пока не застрянет на v_2, создавая еще больший цикл, который мы называем $Cycle_2$.

На рис. 3.25 (справа) цикл $Cycle_2$ оказывается эйлеровым, хотя для произвольного графа это определенно не так. В общем, Leo генерирует все большие и большие циклы на каждой итерации, и поэтому мы гарантируем, что рано или поздно какой-то $Cycle_m$ пройдет все ребра графа. Этот цикл должен быть эйлеровым, и поэтому мы (и Лео) закончили наш путь. ■

ОСТАНОВИТЕСЬ и задумайтесь. Сформулируйте и докажите аналог теоремы Эйлера для неориентированных графов.

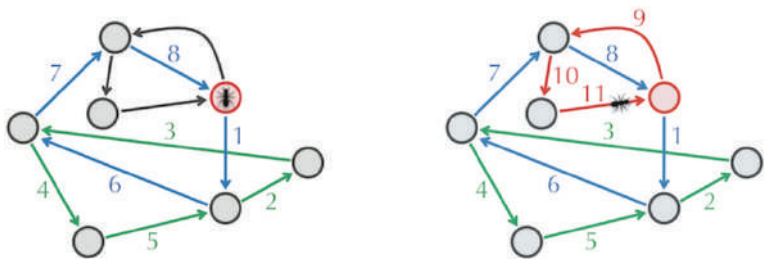


Рис. 3.25 (Слева) Начиная с нового узла v_2 (показанного красным), Лео сначала проходит по ранее построенному $Cycle_1$ (показанному зеленым и синим ребрами). (Справа) Завершив обход $Cycle_1$, Лео продолжает случайный обход графа и наконец создает эйлеров цикл

От теоремы Эйлера к алгоритму нахождения эйлеровых циклов

Построение эйлеровых циклов

Доказательство теоремы Эйлера представляет собой пример того, что математики называют **конструктивным доказательством**, которое не только дает желаемый результат – доказательство, но также дает нам метод построения нужного нам объекта. Короче говоря, мы отслеживаем движения Лео до тех пор, пока он неизбежно не создаст эйлеров цикл в сбалансированном и сильно связном графе *Graph*, как показано в следующем псевдокоде.

EulerianCycle(*Graph*)
постройте цикл *Cycle* из произвольных прогулок в графе *Graph* (не проходите ни одного ребра дважды!)
while остаются непройденные ребра в *Graph*
выберите узел *newStart* в цикле *Cycle* с непройденными ребрами
постройте цикл *Cycle'* путем изменения *Cycle* (начиная с *newStart*) и произвольного передвижения далее
Cycle ← *Cycle'*
return *Cycle*

Это может быть не очевидным, но хорошая реализация **EulerianCycle** будет работать за линейное время. Для достижения такого ускорения времени вычисления выполнения вам потребуется использовать эффективную структуру данных, чтобы поддерживать текущий цикл, который строит Leo, а также список неиспользуемых ребер, связанных с каждым узлом, и список узлов в текущем цикле, у которых есть неиспользованные ребра.

От эйлеровых циклов к эйлеровым путям

Теперь мы можем проверить, есть ли в ориентированном графе эйлеров *цикл*, но как насчет эйлерова *пути*? Рассмотрим граф де Брюйна на рис. 3.26 (слева), который, как мы уже знаем, имеет эйлеров путь, но не имеет эйлерова цикла, поскольку узлы **TA** и **TT** не сбалансированы. Однако мы можем преобразовать этот эйлеров путь в эйлеров цикл, добавив единственное ребро, соединяющее **TT** с **TA**, как показано на рис. 3.26 (справа).

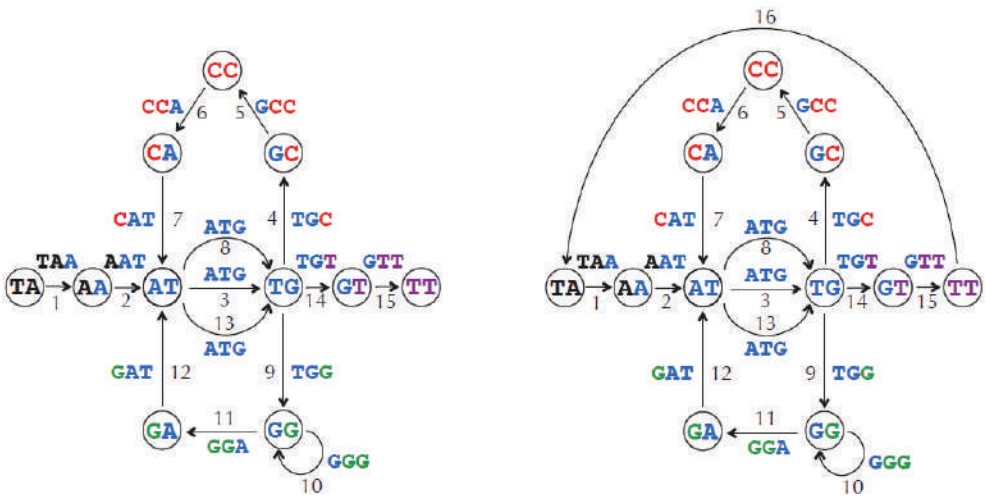


Рис. 3.26 Преобразование эйлерова пути (слева) в эйлеров цикл (справа) путем добавления ребра

ОСТАНОВИТЕСЬ и задумайтесь. Сколько несбалансированных узлов имеет граф с эйлеровым путем?

В более общем случае рассмотрим граф, который не имеет эйлерова цикла, но имеет эйлеров путь. Если эйлеров путь в этом графе соединяет узел v с другим узлом ω, то граф **почти сбалансирован**, что означает, что все его узлы, кроме v и ω, сбалансированы. В этом случае добавление дополнительного ребра из ω в v превращает эйлеров путь в эйлеров цикл. Таким образом, почти сбалансированный граф имеет эйлеров путь тогда и только тогда, когда добавление ребра между его несбалансированными узлами делает граф сбалансированным и сильно связным.

Теперь у вас есть метод сборки генома, поскольку задача реконструкции строки сводится к поиску эйлерова пути в графе де Брюйна, сгенерированном из ридов.

Упражнение. Найдите аналог почти сбалансированного условия, которое будет определять, когда неориентированный граф имеет эйлеров путь.

Аналог теоремы Эйлера для неориентированных графов очевидно подразумевает, что в Кенигсберге XVIII века нет эйлерова пути, но в современном Калининграде дело обстоит иначе (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Семь мостов Калининграда**).

Создание универсальных строк

Теперь, когда вы знаете, как использовать граф де Брюйна для решения задачи реконструкции строки, вы также можете построить k-универсальную строку для любого значения k. Отметим, что де Брюйна интересовало построение k-универсальных *круговых* строк. Например, **0**0**0**1**1**1**0**1 – это 3-универсальная циклическая строка, поскольку она содержит каждый из восьми бинарных 3-меров ровно один раз (рис. 3.27).

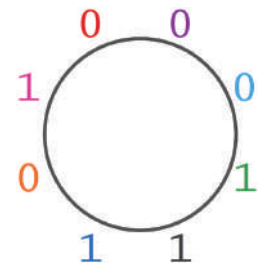


Рис. 3.27 Циклическая 3-универсальная строка **0**0**0**1**1**1**0**1 содержит каждый из бинарных 3-меров (**0**0**0**, **0**0**1**, **0**1**1**, **1**1**1**, **1**1**0**, **1**0**1**, **0**1**0**, и **1**0**0**) ровно один раз

Задача *k*-универсальной круговой строки: *найдите k-универсальную круговую строку.*

Input: целое число k.

Output: k-универсальная круговая строка.

Как и его аналог для линейных цепей, задача о k-универсальных круговых строках – это всего лишь частный случай более общей задачи, которая требует, чтобы мы реконструировали круговую строку с учетом ее k-мерного состава. Эта задача моделирует сборку кольцевого генома, содержащего одну хромосому, подобно геномам большинства бактерий. Мы знаем, что можем восстановить круговую строку по ее k-мер-композиции, найдя эйлеров цикл в графе де Брюйна, построенном из этих k-меров. Следовательно, мы можем построить k-универсальную круговую двоичную строку, найдя эйлеров цикл в графе де Брюйна, построенном из набора всех двоичных k-меров (рис. 3.28).

Упражнение. Сколько существует 3-универсальных круговых строк?

Несмотря на то что поиск 20-универсальной круговой строки эквивалентен поиску эйлерова цикла в графе более чем с миллионом ребер, теперь мы имеем быстрый алгоритм для решения этой задачи. Пусть $BinaryStrings_k$ будет набором всех 2^k двоичных k-меров. Единственное, что нам нужно сделать, – это решить задачу k-универсальной круговой строки и найти эйлеров цикл в $Debruijn(BinaryStrings_k)$. Обратите внимание, что узлы этого графа представляют все возможные двоичные $(k-1)$-меры. Направленное ребро соединяет $(k-1)$-мер *Pattern* с $(k-1)$-мер *Pattern′* в этом графе, если существует k-мер, префикс которого – *Pattern*, а суффикс – *Pattern′*.

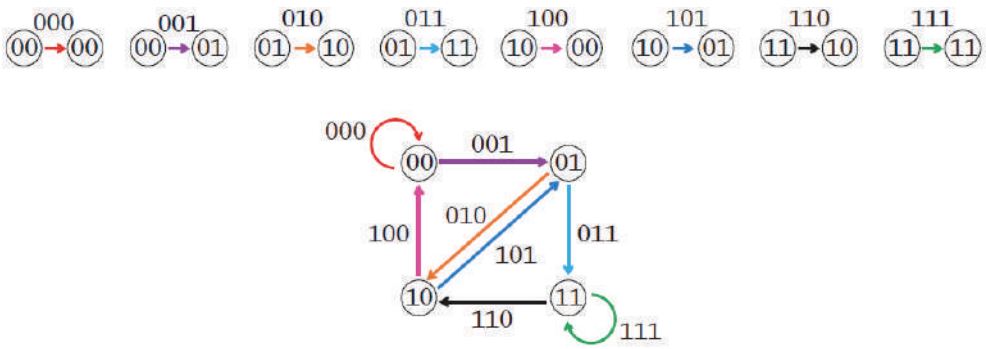


Рис. 3.28 (Вверху) Граф, состоящий из восьми изолированных направленных ребер, по одному для каждого двоичного 3-мера. Узлы каждого ребра соответствуют префиксу и суффиксу 3-мера. (Внизу) Склеивание одинаково помеченных узлов в графе сверху приводит к графу де Брюйна, содержащему четыре узла. Эйлеров цикл через ребра 000 → 001 → 011 → 111 → 110 → 101 → 010 → 100 → 000 дает 3-универсальную круговую строку 00011101

ОСТАНОВИТЕСЬ и задумайтесь. На рис. 3.29 показано, что DeBruijn(*BinaryStrings*$_k$) сбалансирован и сильно связен, следовательно, он является эйлеровым. Можете ли вы доказать, что для любого *k* функция *DeBruijn*(*BinaryStrings*$_k$) является эйлеровой?

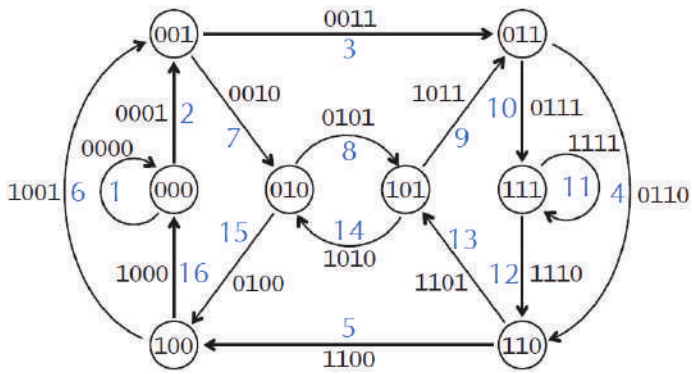


Рис. 3.29 Эйлеров цикл, записывающий циклическую 4-универсальную строку 0000110010111101 в *Debruijn*(*BinaryStrings*$_k$)

Сборка геномов из рид-пар

От ридов к рид-парам

Ранее мы описали идеализированную форму сборки генома, чтобы развить ваше интуитивное представление о графах де Брюйна. В оставшейся части главы мы обсудим ряд практических тем, которые помогут вам оценить передовые методы, используемые современными ассемблерами (программами сборки генома).

Мы уже упоминали, что сборка ридов, выбранных из случайно сгенерированного текста, является тривиальной задачей, поскольку не ожидается, что случайные строки будут иметь длинные повторы. Более того, графы де Брюйна становятся все менее и менее запутанными по мере увеличения длины рида (рис. 3.30). Как только длина рида превышает длину всех повторов в геноме (при условии, что в ридах нет ошибок), граф де Брюйна превращается в путь. Однако, несмотря на многочисленные попытки, биологи так и не придумали, как генерировать *длинные* и *точные* риды. Самые точные технологии секвенирования, доступные сегодня, генерируют риды длиной всего около 300 нуклеотидов, что слишком мало для охвата большинства повторов даже в коротких бактериальных геномах.

Ранее мы видели, что строка **TA**ATGCCATGGGATGTT не может быть однозначно восстановлена из ее 3-мерного состава, поскольку другая строка (**TA**ATGGGATGCCATGTT) имеет такую же 3-мер-композицию.

ОСТАНОВИТЕСЬ и задумайтесь. Какая дополнительная экспериментальная информация позволит вам однозначно восстановить строку **TA**ATGCCATGGGATGTT?

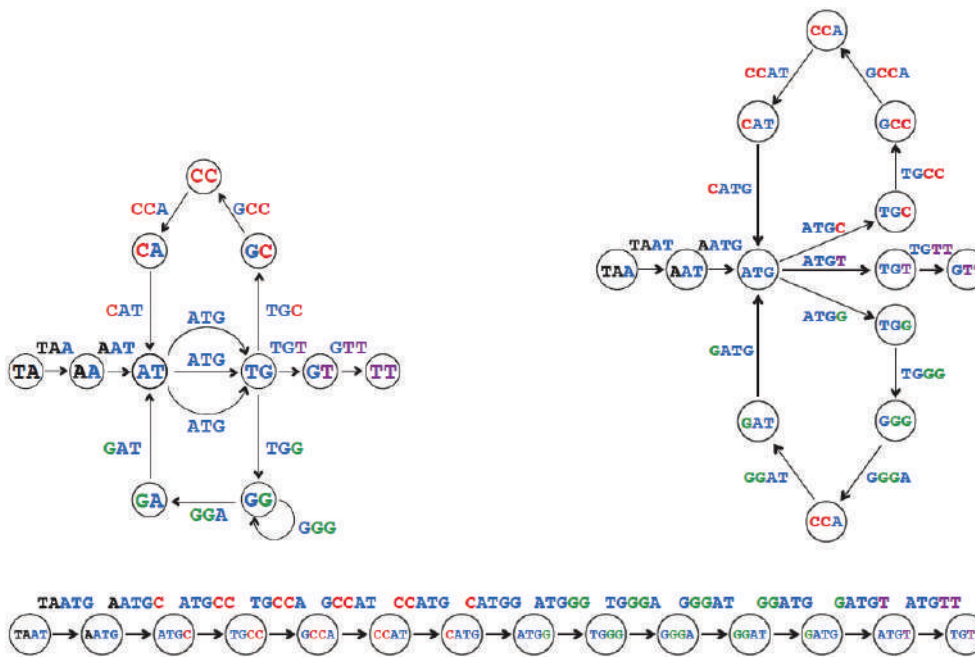


Рис. 3.30 Граф $DeBruijn_4$(**TA**ATGCCATGGGATGTT) (вверху справа) менее запутан, чем граф $DeBruijn_3$(**TA**ATGCCATGGGATGTT) (вверху слева). Граф $DeBruijn_5$(**TA**ATGCCATGGGATGTT) (внизу) представляет собой путь

Увеличение длины рида помогло бы осуществить правильную сборку, но, поскольку оно представляет собой сложную экспериментальную задачу, биологи разработали остроумный экспериментальный метод увеличения длины рида путем создания рид-пар, представляющих собой пары ридов, разделенных фиксированным расстоянием d в геноме (рис. 3.31). Вы можете думать о рид-паре как о длинном риде с промежутком d общей длиной $k + d + k$, первый и последний k-меры которого известны, но чей средний сегмент длины d неизвестен. Тем не менее рид-пары содержат больше информации, чем одни k-меры, и поэтому мы должны иметь возможность использовать их для улучшения наших сборок генома. Если бы вы только могли определить нуклеотиды в среднем сегменте рид-пары, вы бы немедленно увеличили длину рида с k до $2 \cdot k + d$.

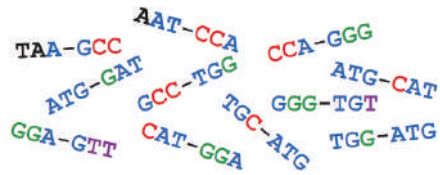


Рис. 3.31 Рид-пары, отобранные из **TA**ATG**CC**ATGGGATG**TT** и образованные ридами длины 3, разделенными промежутком длиной 1. Простой, но неэффективный способ собрать эти пары ридов состоит в построении графа де Брюйна отдельных ридов (3-меров) в пределах рид-пары

Преобразование рид-пар в длинные виртуальные риды

Пусть *Reads* будет совокупностью всех $2N$ ридов k-меров, взятых из N рид-пар. Обратите внимание, что рид-пара, образованная k-мер ридами $Read_1$ и $Read_2$, соответствует двум ребрам в графе де Брюйна $DeBruijn_k(Reads)$. Поскольку эти риды разделены расстоянием d в геноме, должен быть путь длиной $k + d + 1$ в $DeBruijn_k(Reads)$, соединяющий узел в начале ребра, соответствующий $Read_1$, с узлом в конце ребра, соответствующему $Read_2$, как показано на рис. 3.32. Если существует только один путь длины $k + d + 1$, соединяющий эти узлы, или если все такие пути составляют одну и ту же строку, то мы можем преобразовать рид-пару, образованную ридами $Read_1$ и $Read_2$, в виртуальный рид длины $2 \cdot k + d$, который начинается как $Read_1$, выписывает этот путь и заканчивается $Read_2$.

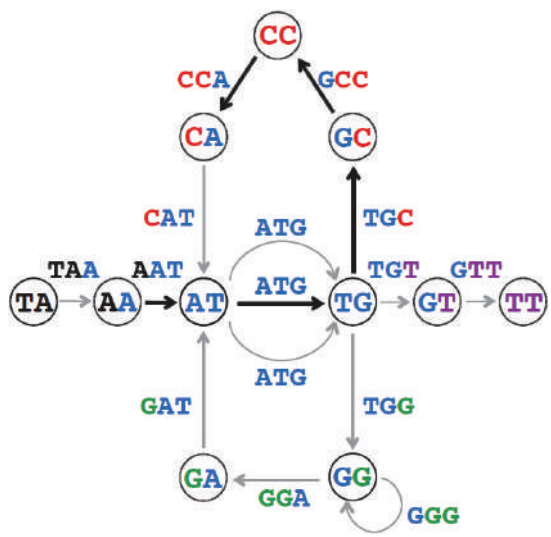


Рис. 3.32 Выделенный путь длиной $k + d + 1 = 3 + 1 + 1 = 5$ (соединяющий узел **AAT** с узлом **CCA**) представляет собой строку **AATGCCA**. (Имеется три таких пути, потому что есть три возможных выбора ребер, помеченных как **ATG**.) Таким образом, рид с промежутком **AAT**–**CCA** может быть преобразован в длинный виртуальный рид **AATGCCA**

Например, рассмотрим граф де Брюйна на рис. 3.32, который создается из всех ридов, присутствующих в рид-парах на рис. 3.31. Существует уникальная строка, написанная в виде пути длины $k + d + 1 = 5$ между ребрами, помеченными **AAT** и **CCA**, в рид-паре, представленной ридом с промежутком **AAT**–**CCA**. Таким образом, из двух коротких ридов длины k мы сгенерировали длинный виртуальный рид длиной $2 \cdot k + d$, достигнув вычислительным путем того, чего исследователи до сих пор не могут достичь экспериментально! После предварительной обработки графа де Брюйна для создания длинных виртуальных ридов мы можем просто построить граф де Брюйна из этих длинных ридов и использовать его для сборки генома.

Хотя идея преобразования рид-пар в длинные виртуальные риды используется во многих программах сборки генома, мы сделали оптимистическое предположение: *«Если есть только один путь длины k + d + 1, соединяющий эти узлы, или если все такие пути выписывают ту же самую строку...»* На практике это предположение ограничивает применение метода длинного виртуального рида к сборке рид-пар, потому что весьма часто повторяющиеся участки генома содержат несколько путей одинаковой длины между двумя ребрами, и эти пути часто представляют собой разные строки (рис. 3.33). Если это так, то мы не можем надежно преобразовать рид-пару в длинный рид. Вместо этого мы опишем альтернативный подход к анализу рид-пар.

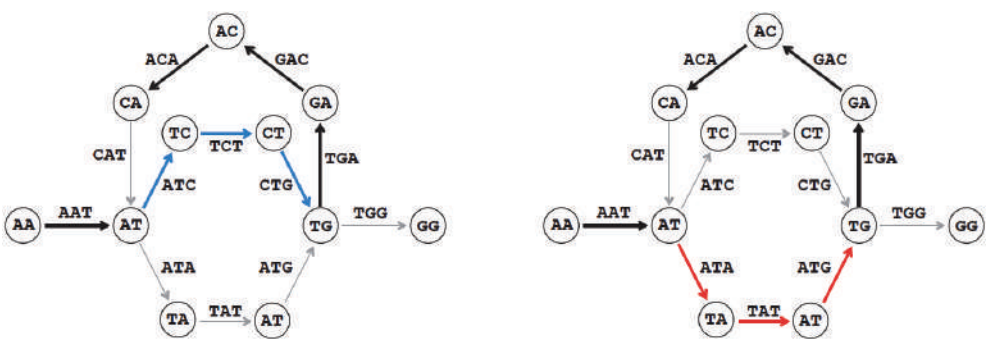


Рис. 3.33 (Слева) Выделенный путь в $DeBruijn_3$(AATCTGACATATGG) описывает длинный виртуальный рид AATCTGACA, который является подстрокой AACTGACATATGG. (Справа) Выделенный путь на том же графе обозначает длинный виртуальный рид AATATGACA, которого нет в AATCTGACATATGG

От композиции к спаренной композиции

Для строки *Text* (k, d)-мер представляет собой пару k-меров в *Text*, разделенных расстоянием d. Мы используем обозначение $(Pattern_1|Pattern_2)$ для обозначения (k, d)-мера, k-мерами которого являются $Pattern_1$ и $Pattern_2$. Например, (**AAT** | **TGG**) представляет собой (3, 4)-мер в **TA**ATGCCATGGGATGTT. Композиция

(*k*, *d*)-меров *Text*, обозначаемая $PairedComposition_{k,d}(Text)$, представляет собой совокупность всех (*k, d*)-меров в *Text* (включая повторяющиеся (*k, d*)-меры). Например, $PairedComposition_{3,1}$(**TAATGCCATGGGATGTT**):

```
TAA GCC
 AAT CCA
  ATG CAT
   TGC ATG
    GCC TGG
     CCA GGG
      CAT GGA
       ATG GAT
        TGG ATG
         GGG TGT
          GGA GTT
TAATGCCATGGGATGTT
```

Упражнение. Сгенерируйте (3, 2)-мер-композицию строки **TAATGCCATGGGATGTT**.

Поскольку порядок (3, 1)-меров в *PairedComposition*(**TAATGCCATGGGATGTT**) неизвестен, мы перечислим их в соответствии с лексикографическим порядком 6-меров, образованных их конкатенированными 3-мерами:

(**AAT**|**CCA**) (**ATG**|**CAT**) (**ATG**|**GAT**) (**CAT**|**GGA**) (**CCA**|**GGG**) (**GCC**|**TGG**)
(**GGA**|**GTT**) (**GGG**|**TGT**) (**TAA**|**GCC**) (**TGC**|**ATG**) (**TGG**|**ATG**)

Заметим, что если в 3-мер-композиции этой строки есть повторяющиеся 3-меры, то в ее парном составе нет повторяющихся (3, 1)-меров. Кроме того, хотя **TAATGCCATGGGATGTT** и **TAATGGGATGCCATGTT** имеют одинаковый состав 3-меров, они имеют разные составы (3, 1)-меров. Таким образом, если мы сможем сгенерировать (3, 1)-мер-композицию этих строк, то сможем их различить. Но как мы можем восстановить строку по ее (*k, d*)-мер-композиции? И можем ли мы адаптировать метод графа де Брюйна для этой цели?

Задача реконструкции строки из рид-пар: *восстановите строку из ее спаренной композиции.*

Input: набор спаренных *k*-меров *PairedReads* и целое число *d*.

Output: строка *Text* с (*k, d*)-мер-композицией, равной *PairedReads* (если такая строка существует).

Парные графы графы де Брюйна

Для данного (k, d)-мера ($a_1 \ldots a_k \mid b_1, \ldots b_k$) мы определяем его **префикс** и **суффикс** ($k - 1$, $d + 1$)-меров следующим образом:

$$Prefix((a_1 \ldots a_k \mid b_1, \ldots b_k)) = (a_1 \ldots a_{k-1} \mid b_1, \ldots b_{k-1})$$
$$Suffix((a_1 \ldots a_k \mid b_1, \ldots b_k)) = (a_2 \ldots a_k \mid b_2, \ldots b_k).$$

Например, *Prefix*((GAC|TCA)) = (GA|TC) и *Suffix*((GAC|TCA)) = (AC|CA).

Обратите внимание, что для последовательных (k, d)-меров, появляющихся в *Text*, суффикс первого (k, d)-мера равен префиксу второго (k, d)-мера. Например, для последовательных (k, d)-меров (**TA**A|**GCC**) и (**AAT**|**CCA**) в **TAATGCCATGGGATGTT**

$$Suffix((\mathbf{TA}A|\mathbf{GCC})) = Prefix((\mathbf{AAT}|\mathbf{CCA})) = (\mathbf{A}A|\mathbf{CC}).$$

По заданной строке *Text* мы строим граф $PathGraph_{k,d}(Text)$, представляющий путь, образованный $|Text| - (k + d + k) + 1$ ребрами, соответствующими всем (k, d)-мерам в *Text*. Мы помечаем ребра на этом пути (k, d)-мерами, а начальный и конечный узлы ребра помечаем его префиксом и суффиксом соответственно (рис. 3.34).

Рис. 3.34 $PathGraph_{3,1}$(**TAATGCCATGGGATGTT**). Каждый (3, 1)-мер для экономии места был отображен как двухстрочное выражение

Парный граф де Брюйна, обозначенный $DeBruijn_{k,d}(Text)$, формируется путем склеивания узлов с одинаковыми метками в $PathGraph_{k,d}(Text)$ (рис. 3.35). Обратите внимание, что парный граф де Брюйна менее запутан, чем граф де Брюйна, построенный из отдельных ридов на рис. 3.14.

ОСТАНОВИТЕСЬ и задумайтесь. Парный граф де Брюйна легко построить из строки *Text*. Но как мы можем построить парный граф де Брюйна из (k, d)-мер-композиции *Text*?

Определим $PairedCompositionGraph_{k,d}(Text)$ как граф, состоящий из $|Text| - (k + d + k) + 1$ изолированных ребер, помеченных (k, d)-мерами в *Text*, чьи узлы помечены префиксами и суффиксами этих меток (рис. 3.36). Как вы можете предположить, склеивание одинаково помеченных узлов в $PairedCompositionGraph_{k,d}(Text)$ приводит к точно такому же графу де Брюйна, что и склеивание одинаково помеченных узлов в $PathGrap_{k,d}(Text)$. Конечно, на практике мы не можем знать оригинальную последовательность *Text*; однако можем

сформировать $PairedCompositionGraph_{k,d}(Text)$ непосредственно из (k, d)-мер-композиции *Text*, и на этапе склеивания получится парный граф де Брюйна этой композиции. Геном можно реконструировать, следуя эйлеровой траектории на этом графе де Брюйна.

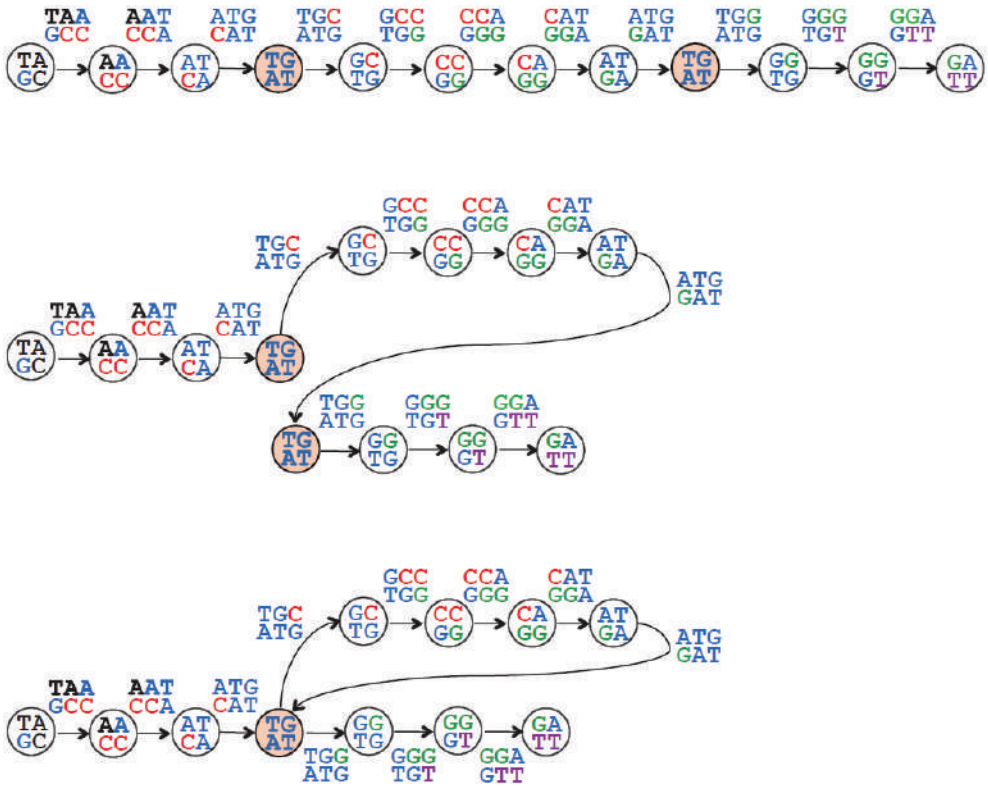


Рис. 3.35 (Вверху) $PathGrap_{3,\ 1}$(**TAATGCCATGGGATGTT**) состоит из 11 ребер и 12 узлов. Только два из этих узлов имеют одинаковую метку (**TG**|**AT**). (Посередине) Приведение двух одинаково помеченных узлов ближе друг к другу при подготовке к склеиванию. (Внизу) Парный граф де Брюйна $DeBruijn_{3,1}$(**TAATGCCATGGGATGTT**) получается из $PathGrap_{3,\ 1}$(**TAATGCCATGGGATGTT**) путем склеивания узлов с общей меткой (**TG**|**AT**). Этот парный граф де Брюйна имеет уникальный эйлеров путь, который выглядит как **TAATGCCATGGGATGTT**

Ловушка парных графов де Брюйна

Ранее мы видели, что каждое решение задачи реконструкции строки соответствует эйлеровому пути в графе де Брюйна, построенном из k-мер-композиции. Так же каждое решение задачи реконструкции строки по рид-парам соответствует эйлеровому пути в парном графе де Брюйна, построенному из (k, d)-мер-композиции.

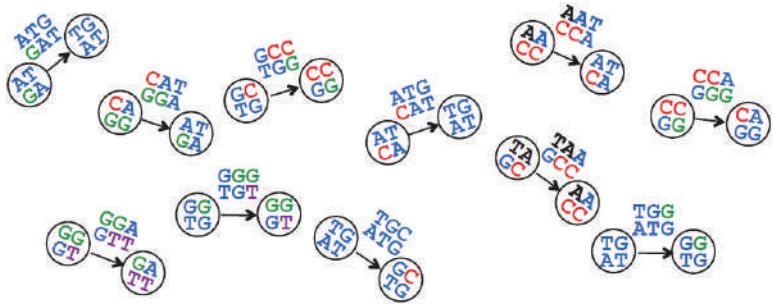


Рис. 3.36 Граф *PairedCompositionGraph*$_{3,\,1}$(**TAATGCCATGGGATGTT**) представляет собой набор изолированных ребер. Каждое ребро помечено (3, 1)-мером в **TAATGCCATGGGATGTT**; начальный узел ребра помечен префиксом (3, 1)-мера ребра, а конечный узел ребра помечен суффиксом этого (3, 1)-мера. Склеивание одинаково помеченных узлов дает парный граф де Брюйна, показанный на рис. 3.35 (внизу)

Упражнение. На парном графе де Брюйна, показанном на рис. 3.37, реконструируйте геном, представленный следующим эйлеровым путем (2, 1)-меров: (AG|AG) → (GC|GC) → (CA|CT) → (AG|TG) → (GC|GC) → (CT|CT) → (TG|TG) → (GC|GC) → (CT|CA).

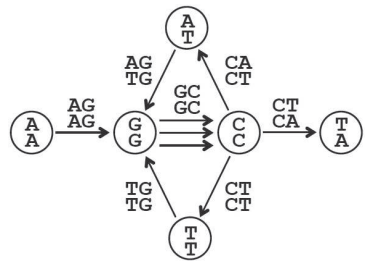


Рис. 3.37 Парный граф де Брюйна, построенный из набора девяти (2, 1)-меров (AG|AG), (AG|TG), (CA|CT), (CT|CA), (CT|CT), (GC|GC), (GC|GC), (GC|GC) и (TG|TG)

Мы также видели, что каждый эйлеров путь в графе де Брюйна, построенный из композиции k-меров, представляет собой решение задачи реконструкции строки. Но верно ли это для парного графа де Брюйна?

ОСТАНОВИТЕСЬ и задумайтесь. Граф, показанный на рис. 3.37, имеет еще один эйлеров путь: (AG|AG) → (GC|GC) → (CT|CT) → (TG|TG) → (GC|GC) → (CA|CT) → (AG|TG) → (GC|GC) → (CT|CA). Можете ли вы реконструировать геном, написанный этим путем?

Если вы попытались ответить на предыдущий вопрос, то вы знаете, что не каждый эйлеров путь в парном графе де Брюйна, построенном из (*k, d*)-мер-композиции, дает решение задачи реконструкции строки из рид-пары.

Упражнение представило следующий эйлеров путь, образованный девятью ребрами в нашем графе.

AG–AG → GC–GC → CA–CT → AG–TG → GC–GC →
CT–CT → TG–TG → GC–GC → CT–CA

Мы можем упорядочить (2,1)-меры этого пути в девяти строках, показанных ниже, определив строку AGCAGCTGCTGCA, написанную этим путем:

```
AG–AG
 GC–GC
   CA–CT
   AG–TG
     GC–GC
     CT–CT
       TG–TG
       GC–GC
         CT–CA
AGCAGCTGCTGCA
```

Теперь снова взгляните на тот же график и рассмотрите эйлеров путь, представленный вопросом «Остановитесь и задумайтесь»:

AG–AG → GC–GC → CT–CT → TG–TG → GC–GC →
CA–CT → AG–TG → GC–GC → CT–CA

Попытка собрать эти (2,1)-меры показывает, что не каждый столбец имеет один и тот же нуклеотид (см. два столбца, показанные ниже красным). Этот пример показывает, что не каждый эйлеров путь в спаренном графе де Брюйна, построенном из (*k*, *d*)-мер-композиции, дает решение задачи реконструкции строки по прочитанным парам.

```
AG–AG
GC–GC
  CT–CT
  TG–TG
    GC–GC
    CA–CT
      AG–TG
      GC–GC
        CT–CA
AGC?GC?GCTGCA
```

Теперь вы готовы реконструировать строку из рид-пар и стать экспертом по сборке генома.

ЗАРЯДНАЯ СТАНЦИЯ: Генерация всех эйлеровых циклов. Вы знаете, как построить один эйлеров цикл в графе, но остается неясным, как найти все возможные эйлеровы циклы, что будет полезно при решении задачи реконструкции строки из рид-пар. Изучите эту зарядную станцию, чтобы увидеть, как сгенерировать все эйлеровы циклы в графе.

ЗАРЯДНАЯ СТАНЦИЯ: Реконструкция строки, написанной по пути в парном графе де Брюйна. Чтобы решить задачу реконструкции строки из рид-пар, вам нужно будет восстановить строку по ее пути в парном графе де Брюйна. Изучите эту зарядную станцию, чтобы увидеть пример того, как это можно сделать.

Эпилог. Сборка генома – работа с реальными данными секвенирования

Наше обсуждение сборки генома до сих пор основывалось на различных предположениях. Соответственно, применение графов де Брюйна к реальным данным секвенирования не является простой процедурой. Ниже мы описываем практические задачи, связанные с особенностями современных технологий секвенирования, и некоторые вычислительные методы, разработанные для решения этих задач. В этом разделе в качестве упрощения мы сначала предположим, что риды генерируются как непрерывные подстроки генома, а не рид-пары.

Разбиваем риды на k-меры

Для заданной *k*-мер подстроки генома мы определяем ее **покрытие** как количество ридов, содержащих этот *k*-мер. Мы приняли как должное, что ассемблер может генерировать все *k*-меры, присутствующие в геноме, но это предположение об «идеальном *k*-мере покрытия» на практике не выполняется. Например, популярная технология секвенирования Illumina генерирует риды длиной примерно 300 нуклеотидов, но эта технология по-прежнему пропускает многие 300-меры, присутствующие в геноме (даже если средний охват очень высок), и почти все риды, которые он генерирует, содержат ошибки секвенирования.

ОСТАНОВИТЕСЬ и задумайтесь. При заданном наборе ридов с неполным покрытием k-меров можете ли вы найти параметр $l < k$, чтобы те же риды имели идеальное покрытие l-меров? Какое максимальное значение этого параметра?

На рис. 3.38 (слева) показаны четыре 10-меров ридов, которые охватывают некоторые, но не все 10-меры в геноме примера. Однако если мы предпримем нелогичный шаг и разобьем эти риды на *более короткие* 5-меры (рис. 3.38, справа), то эти 5-меры будут иметь идеальное покрытие. Этот **метод фрагментации рида**, при котором мы разбиваем рид на более короткие k-меры, используется многими современными ассемблерами.

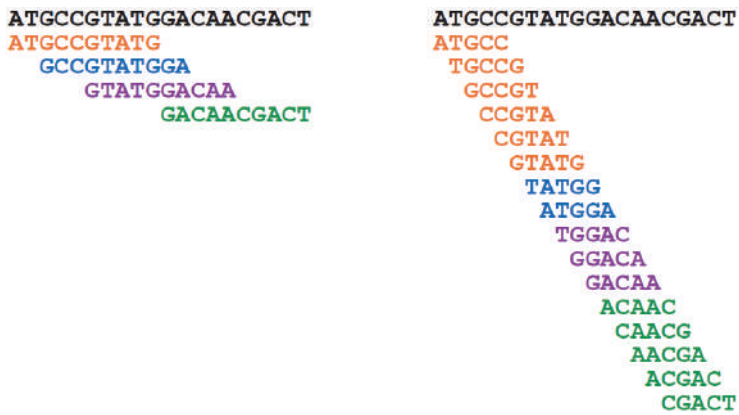


Рис. 3.38 Разбиение 10-мер ридов (слева) на 5-меры приводит к идеальному покрытию генома 5-мерами (справа)

Фрагментация рида должна иметь практический компромисс. С одной стороны, чем меньше значение k, тем больше вероятность того, что k-мер-покрытие будет идеальным. С другой стороны, меньшие значения k приводят к более запутанному графу де Брюйна, что затрудняет вычисление генома из этого графа.

Фрагментация генома на контиги

Даже после разрыва рида большинство сборок генома по-прежнему имеет пробелы в покрытии k-меров, что приводит к тому, что в графе де Брюйна отсутствуют ребра, и поэтому поиск эйлерова пути терпит неудачу. В этом случае биологи часто останавливаются на сборке **контигов** (длинных смежных сегментов генома), а не целых хромосом. Например, типичный проект по секвенированию бактерий может привести к получению около сотни контигов длиной от нескольких тысяч до нескольких сотен тысяч нуклеотидов. Для

большинства геномов порядок этих контигов в геноме остается неизвестным. Излишне говорить, что биологи предпочли бы иметь всю геномную последовательность, но затраты на упорядочивание контигов в окончательную сборку и закрытие пробелов с использованием более дорогих экспериментальных методов часто непомерно высоки.

К счастью, мы можем вывести контиги из графа де Брюйна. Путь в графе называется **неветвящимся**, если $In(v) = Out(v) = 1$ для каждого промежуточного узла v этого пути, т. е. для каждого узла, кроме, быть может, начального и конечного узлов пути. Максимальный неветвящийся путь – это неветвящийся путь, который нельзя расширить до более длинного неветвящегося пути. Нас интересуют эти пути, потому что цепи нуклеотидов, которые они составляют, должны присутствовать в любой сборке с данным составом k-меров. По этой причине контиги соответствуют строкам, состоящим из максимальных неветвящихся путей в графе де Брюйна. Например, граф де Брюйна на рис. 3.39, построенный для 3-мер-композиции **TA**ATGCCATGGGATGTT, имеет девять максимальных неветвящихся путей, описывающих контиги **TA**AT, TGTT, TGCCAT, ATG, ATG, ATG, TGG, GGG, и GGAT. На практике у биологов нет иного выбора, кроме как разбивать геномы на контиги, даже в случае полного покрытия (как на рис. 3.39), поскольку их повторы не позволяют извлечь единственный эйлеров путь.

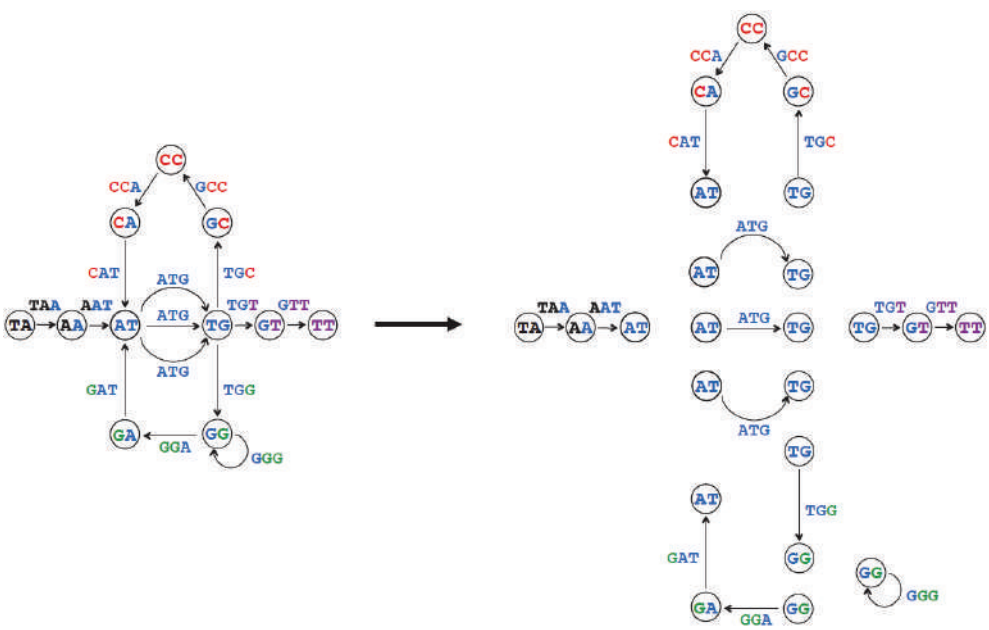


Рис. 3.39 Разбиение графа $DeBruijn_3$(**TA**ATGCCATGGGATGTT) на девять максимальных неветвящихся путей, представляющих контиги **TA**AT, TGTT, TGCCAT, ATG, ATG, ATG, TGG, GGG и GGAT

Сборка ридов с возможными ошибками

Риды с возможными ошибками представляют собой еще одно препятствие для реальных проектов секвенирования. Добавление ошибочного рида CGTA**C**GGACA (с единственной ошибкой, которая неправильно интерпретирует T как **C**) к набору ридов на рис. 3.38 приводит к ошибочным 5-мерам CGTA**C**, GTA**C**G, TA**C**GG, A**C**GGA и **C**GGAC после разрыва рида. Эти 5-меры приводят к ошибочному пути от узла CGTA к узлу GGAC в графе де Брюйна (рис. 3.40 (вверху)), а это означает, что если также сгенерирован правильный рид CGTATGGACA, то у нас будет два пути, соединяющих CGTA с GGAC в графе де Брюйна. Эта структура называется **пузырьком**, который мы определяем как два коротких непересекающихся пути (например, короче некоторой пороговой длины), соединяющих одну и ту же пару узлов в графе де Брюйна.

ОСТАНОВИТЕСЬ и задумайтесь. Каков размер пузырька (в графе де Брюйна, построенного из *k*-меров), который возникает из-за одной ошибки при чтении?

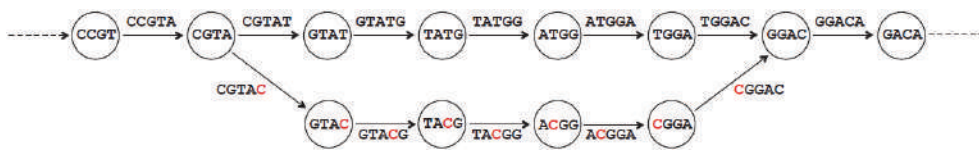


Рис. 3.40 (Вверху) Правильный путь CGTA → GTAT → TATG → ATGG → TGGA → GGAC вместе с неправильным путем CGTA → GTA**C** → TA**C**G → A**C**GG → **C**GGA → GGAC образуют пузырек на графе де Брюйна, что затрудняет определение правильного пути

ОСТАНОВИТЕСЬ и задумайтесь. Разработайте алгоритм для обнаружения пузырьков в графах де Брюйна. После обнаружения пузырька вы должны решить, какой из двух путей в пузырьке удалить. Какое решение вы примете?

Существующие ассемблеры удаляют пузырьки из графов де Брюйна. Практическая задача заключается в том, что, поскольку почти все операции рида содержат ошибки, графы де Брюйна имеют миллионы пузырьков (рис. 3.40 (внизу)). Однако при удалении пузырьков иногда удаляется правильный путь, тем самым вместо исправления вносится добавочная ошибка. Что еще хуже, в геноме, имеющем **неточные повторы**, где повторяющиеся области отличаются одним нуклеотидом или какой-либо другой небольшой вариацией, считывание двух повторяющихся копий также будет генерировать пузырьки в графе де Брюйна, потому что одна из копий может показаться ошибочной

версией другой. Удаление пузырьков в этих областях приводит к ошибкам сборки, делая повторы более похожими, чем они есть на самом деле. Таким образом, современные сборщики генома пытаются отличить пузырьки, вызванные ошибками секвенирования (которые следует удалить), от пузырьков, вызванных вариациями (которые следует сохранить).

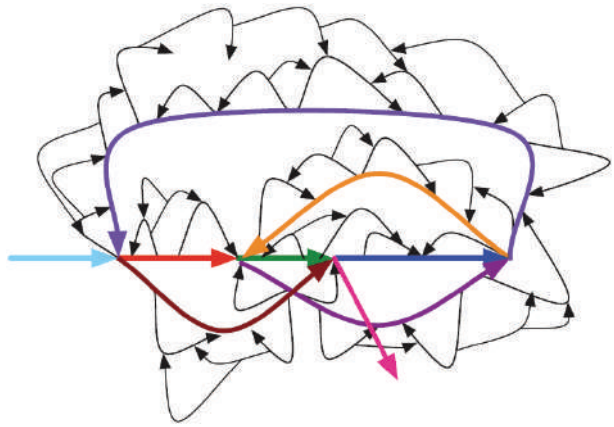


Рис. 3.41 Иллюстрация графа де Брюйна с множеством пузырьков. Удаление пузырьков должно оставить только цветные пути

Определение кратности ребер в графах де Брюйна

Хотя структура графа де Брюйна требует, чтобы мы знали **кратность** каждого *k*-мера в геноме (т. е. сколько раз встречается этот *k*-мер), эту информацию не всегда можно получить из ридов. Однако кратность *k*-мера в геноме часто можно оценить по его покрытию. В самом деле, ожидается, что *k*-меры, которые появляются в геноме *t* раз, будут иметь приблизительно в *t* раз большее покрытие, чем *k*-меры, которые появляются только один раз. Излишне говорить, что охват генома варьируется, и это условие часто нарушается. В результате существующие ассемблеры часто собирают повторяющиеся области в геномах, не зная точного количества раз, когда каждый *k*-мер из этой области встречается в геноме.

Мы познакомили вас с некоторыми практическими аспектами секвенирования генома, но некоторые из них все еще остаются; например, как нам быть с тем фактом, что геномы двухцепочечные? (Смотрите **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Подводные камни сборки двойной цепи ДНК**.) На практике в результате экспериментов по секвенированию генома исследователи в конечном итоге получат набор контигов, но разделение этих контигов на две группы (соответствующие каждой цепи) представляет собой сложную задачу, с которой сталкиваются современные ассемблеры и которую еще предстоит окончательно решить. Например, если ассемблер выводит пять контигов, может оказаться, что контиги 1 и 3 принадлежат одной цепи, а контиги 2, 4 и 5 принадлежат комплементарной ей цепи.

Тем не менее мы зададим вам по-настоящему трудную задачу, в которой вы не столкнетесь с этими проблемами. Почему? Разработка алгоритмов сборки для больших геномов – серьезная задача, потому что даже кажущаяся простой задача построения графа де Брюйна из набора всех *k*-меров, присутствующих в миллионах ридов, нетривиальна. Чтобы облегчить вам жизнь, мы предоставим небольшой бактериальный геном для вашего первого набора данных сборки.

Заключительная задача. *Carsonella ruddii* – это бактерия, которая симбиотически живет внутри некоторых насекомых. Ее защищенная от внешней среды жизнь позволила ей сократить свой геном всего до 160 000 пар оснований. Имея всего около 200 генов, ей не хватает некоторых генов, необходимых для выживания, но эти гены предоставляются ему его насекомым-хозяином. На самом деле у *Carsonella* настолько маленький геном, что биологи предположили, что он теряет свою «бактериальную идентичность» и превращается в **органеллу**, являющуюся частью организма и генома хозяина. Этот переход от бактерии к органелле происходил много раз в истории эволюции; в том числе митохондрия, ответственная за производство энергии в клетках человека и других животных, когда-то была свободно бродящей бактерией, которую мы ассимилировали в далеком прошлом.

Имея набор смоделированных безошибочных рид-пар, используйте парный граф де Брюйна для реконструкции генома *Carsonella ruddii*. Сравните эту сборку со сборкой, полученной из классического графа де Брюйна (т. е. когда все, что мы знаем, это сами риды, но не знаем расстояния между рид-парами), чтобы лучше оценить преимущества рид-пар. Какое минимальное значение *d* необходимо для каждого *k*, чтобы можно было реконструировать весь геном *Carsonella ruddii* по ее (*k, d*)-мер-композиции?

☞ **Загрузить данные 3.1**

Упражнение. Кстати, еще один момент... каков был заголовок газеты «Нью-Йорк Таймс» от 27 июня 2000 года?

Зарядные станции

Влияние склейки на матрицу смежности

На рис. 3.42 используется *Text* = **TA**ATG**CC**ATG**GG**ATG**TT**, чтобы проиллюстрировать, как склеивание преобразует матрицу смежности $PathGraph_3(Text)$ в матрицу смежности $DeBruijn_3(Text)$.

	TA	AA	AT_1	TG_1	GC	CC	CA	AT_2	TG_2	GG_1	GG_2	GA	AT_3	TG_3	GT	TT
TA	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AA	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
AT_1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
TG_1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
GC	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
CC	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
CA	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
AT_2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
TG_2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
GG_1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
GG_2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
GA	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
AT_3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
TG_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
GT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
TT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	TA	AA	AT	TG	GC	CC	CA	GG	GA	GT	TT
TA	0	1	0	0	0	0	0	0	0	0	0
AA	0	0	1	0	0	0	0	0	0	0	0
AT	0	0	0	3	0	0	0	0	0	0	0
TG	0	0	0	0	1	0	0	1	0	1	0
GC	0	0	0	0	0	1	0	0	0	0	0
CC	0	0	0	0	0	0	1	0	0	0	0
CA	0	0	1	0	0	0	0	0	0	0	0
GG	0	0	0	0	0	0	0	1	1	0	0
GA	0	0	1	0	0	0	0	0	0	0	0
GT	0	0	0	0	0	0	0	0	0	0	1
TT	0	0	0	0	0	0	0	0	0	0	0

Рис. 3.42 Матрицы смежности. (Вверху) Матрица смежности 16×16 $PathGraph_3$(**TA**ATG**CC**ATGGGATG**TT**). Обратите внимание, что мы использовали индексирование для различения множественных вхождений AT, TG и GG в $PathGraph_3$(**TA**ATG**CC**ATGGGATG**TT**). (Внизу) Матрица смежности 11×11 $DeBruijn_3$(**TA**ATG**CC**ATGGGATG**TT**), полученная после склеивания узлов, помеченных идентичными 2-мерами. Обратите внимание, что, если есть m ребер, соединяющих вершины i и j, (i, j)-й элемент матрицы смежности равен m

Генерация всех эйлеровых циклов

Главная сложность создания всех эйлеровых циклов в графе заключается в отслеживании потенциально многих различных альтернатив в любом заданном узле. На противоположном конце спектра **простой ориентированный граф**,

связный граф, в котором каждый узел имеет степени входа и выхода, равные 1, он предлагает тривиальный случай, поскольку существует только один эйлеров цикл.

Наша идея состоит в том, чтобы преобразовать один помеченный ориентированный граф *Graph*, содержащий $n \geqslant 1$ эйлеровых циклов, в n различных простых ориентированных графов, каждый из которых содержит один эйлеров цикл. Это преобразование обладает тем свойством, что оно легко обратимо, т. е. по единственному эйлерову циклу в одном из простых ориентированных графов мы можем легко восстановить исходный эйлеров цикл в *Graph*.

Для узла v в *Graph* (степени вхождения больше 1) с входящим ребром (u, v) и исходящим ребром (v, ω) мы построим «более простой» (u, v, ω) **граф обхода**, в котором удалим ребра (u, v) и (v, ω) из *Graph* и добавим новый узел x вместе с ребрами (u, x) и (x, ω) (рис. 3.43 (вверху)). Новые ребра (u, x) и (x, ω) обходного графа наследуют метки удаленных ребер (u, v) и (v, ω) соответственно. Критическое свойство этого графа раскрывается с помощью следующего упражнения.

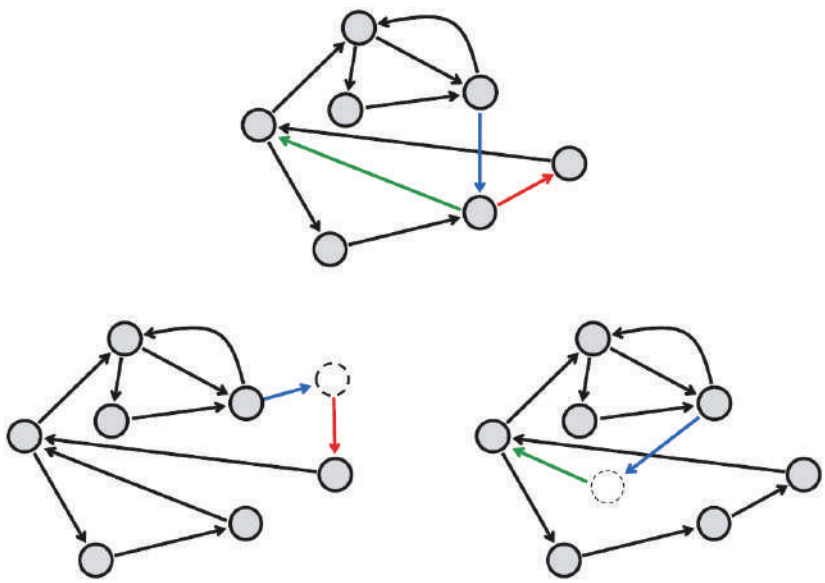


Рис. 3.43 (Вверху) Эйлеров граф. (Внизу слева) Граф обхода (u, v, ω) (справа), построенный для синего и красного ребер. (Внизу справа) Другой граф обхода, построенный для синего и зеленого ребер

Упражнение. Покажите, что любой эйлеров цикл в графе, проходящий через (u, v), а затем (v, ω), соответствует эйлеровому циклу (с теми же метками ребер) в (u, v, ω)-обходном графе, проходящем через (u, x), а затем (x, ω).

В общем, для входящего ребра (u, v) в v вместе с k исходящими ребрами (v, ω_1), ..., (v, ω_k) из v можно построить k различных обходных графов (рис. 3.43 (внизу)). Обратите внимание, что, поскольку никакие два графа обхода не имеют одного и того же эйлерова цикла, общее количество эйлеровых циклов во всех k этих обходных графов равно количеству эйлеровых циклов в исходном графе.

Наша идея, грубо говоря, состоит в том, чтобы итеративно построить все возможные графы обхода для *Graph*, пока мы не получим большое семейство простых ориентированных графов; каждый из этих графов будет соответствовать отдельному эйлеровому циклу в *Graph*. Эта идея реализована псевдокодом, приведенным ниже.

```
AllEulerianCycles(Graph)
    AllGraphs ← набор, состоящий из единственного графа Graph
    while существует граф G в AllGraphs, не являющийся простым
        v ← узел со степенью большей 1 в графе G
        for каждого ребра (u, v) входящего в v
            for каждого ребра (v,w) исходящего из v
                NewGraph ← (u, v, w)-граф обхода G
                if NewGraph является связным
                    добавить NewGraph в AllGraphs
        удалить G из AllGraphs
    for каждого графа G в AllGraphs
        output (единственный) эйлеров цикл в графе G
```

Существует более элегантный метод построения всех эйлеровых циклов в эйлеровом графе, основанный на теореме, к доказательству которой приложил руку де Брюйн. Чтобы узнать об этой теореме, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: «Лучшая» теорема**.

Реконструкция строки, записанной как путь в парном графе де Брюйна

Рассмотрим следующий эйлеров путь, образованный девятью ребрами в парном графе де Брюйна с рис. 3.37.

AG–AG → GC–GC → CA–CT → AG–TG → GC–GC → CT–CT → TG–TG → GC–GC → CT–CA

Мы можем расположить (2, 1)-меры этого пути в девять строк, показанных ниже, открывая строку **AGCAGCTGCTGCA**, написанную следующим путем:

```
AG-AG
   GC-GC
      CA-CT
         AG-TG
            GC-GC
               CT-CT
                  TG-TG
                     GC-GC
                        CT-CA
AGCAGCTGCTGCA
```

Теперь рассмотрим другой эйлеров путь в парном графе де Брюйна с рис. 3.37:

AG–AG → GC–GC → CT–CT → TG–TG → GC–GC → CA–CT → AG–TG → GC–GC → CT–CA

Попытка собрать эти (2, 1)-меры показывает, что не каждый столбец содержит один и тот же нуклеотид (см. два столбца, показанные ниже красным цветом). Этот пример показывает, что не все эйлеровы пути в спаренном графе де Брюйна представляют собой решения задачи реконструкции строки из рид-пар.

```
AG-AG
   GC-GC
      CT-CT
         TG-TG
            GC-GC
               CA-CT
                  AG-TG
                     GC-GC
                        CT-CA
AGC?GC?GCTGCA
```

Задача строки, написанной с пробелом в пути генома:
восстановите строку (k, d)-меров, соответствующую пути в парном графе де Брюйна.

Input: последовательность (*k, d*)-меров $(a_1|b_1), \ldots, (a_n|b_n)$ таких, что $Suffix((a_i|b_i) = Prefix((a_{i+1}|b_{i+1}))$ для $1 \leqslant i \leqslant n - 1$.

Output: строка *Text* длины $k + d + k + n - 1$ такая, что *i*-й (*k, d*)-мер *Text* равен $(a_i|b_i)$ для $1 \leqslant i \leqslant n$ (если такая строка существует).

Наш метод решения этой задачи будет состоять в том, чтобы расщеплять заданные (*k, d*)-меры ($a_1|b_1$), …, ($a_n|b_n$) на их начальные *k*-меры, *FirstPatterns* = (a_1, … , a_n), и их конечные *k*-меры, *SecondPatterns* = (b_1, …, b_n). Предполагая, что мы реализовали алгоритм, решающий задачу строки, записанной путем генома (обозначенной как **StringSpelledByPatterns**), мы можем собрать *FirstPatterns* и *SecondPatterns* в строки *PrefixString* и *SuffixString* соответственно.

Для первого приведенного выше примера у нас есть *PrefixString* = AGCAGCTGCT и *SuffixString* = AGCTGCTGCA. Эти строки полностью перекрываются, начиная с четвертого нуклеотида *PrefixString*:

```
PrefixString = AGCAGCTGCT
SuffixString =    AGCTGCTGCA
Genome       = AGCAGCTGCTGCA
```

Однако для второго приведенного выше примера идеального перекрытия не существует:

```
PrefixString = AGCTGCAGCT
SuffixString =    AGCTGCTGCA
Genome       = AGC?GC?GCTGCA
```

Следующий алгоритм, **StringSpelledByGappedPatterns**, обобщает этот метод для произвольной строки *GappedPatterns* (*k, d*)-меров. Он создает строки *PrefixString* и *SuffixString*, как описано выше, и проверяет, имеют ли они полное перекрытие (т. е. формируют префикс и суффикс реконструированной строки). Также предполагается, что количество (*k, d*)-меров в *GappedPatterns* не менее *d*; в противном случае невозможно восстановить непрерывную строку.

```
StringSpelledByGappedPatterns(GappedPatterns, k, d)
    FirstPatterns ← последовательность начальных k-меров в GappedPatterns
    SecondPatterns ← последовательность конечных k-меров в GappedPatterns
    PrefixString ← StringSpelledByGappedPatterns(FirstPatterns, k)
    SuffixString ← StringSpelledByGappedPatterns(SecondPatterns, k)
    for i = k + d + 1 до |PrefixString|
        if i-й символ в PrefixString ≠ (i − k − d)-му символу в SuffixString
            return «не существует строки, написанной строками с пробелом»
    return строка PrefixString, склеенная с k + d символами строки SuffixString.
```

Максимальные неветвящиеся пути в графе

Узел *v* в ориентированном графе *Graph* называется узлом **FIFO** (**1-in-1-out node**), если его степени входа и выхода равны 1. Мы можем перефразировать определение «максимального неветвящегося пути» из основного текста как путь, внутренние узлы которого являются узлами FIFO, а начальные и конеч-

ные узлы не являются узлами FIFO. Также обратите внимание, что определение из основного текста книги не охватывает случай, когда *Graph* имеет связанный компонент, представляющий собой изолированный цикл, в котором все узлы являются узлами FIFO (вспомните рис. 3.39).

Псевдокод **MaximalNonBranchingPaths**, показанный ниже, перебирает все узлы графа, которые не являются узлами FIFO, и генерирует все неветвящиеся пути, начинающиеся в каждом таком узле. На последнем этапе он находит все изолированные циклы в графе.

```
MaximalNonBranchingPaths(Graph)
    Paths ← пустой список
    for каждого узла v в графе Graph
        if v не является узлом FIFO
            if Out(v) > 0
                for каждого исходящего ребра (v, ω) из v
                    NonBranchingPath ← путь, состоящий из единственного ребра (v, ω)
                    while ω является узлом FIFO
                        продолжить NonBranchingPath исходящим ребром (ω, u) из ω
                        ω ← u
                    добавить NonBranchingPath к набору Paths
    for каждого изолированного цикла Cycle в графе Graph
        добавить Cycle к путям Paths
    return Paths
```

Сопутствующие материалы

Краткая история технологий секвенирования ДНК

В 1988 году Радое Дрманак, Андрей Мирзабеков и Эдвин Саузерн одновременно и независимо друг от друга предложили футуристический и в то время совершенно невероятный метод **ДНК-чипов** (DNA-arrays) для секвенирования ДНК («ДНК-чип», или «ДНК-микрочип», – серия фрагментов ДНК, прикрепленных к физической подложке, например к пластику или стеклу. – *Прим. ред.*). Никто из этих трех биологов не знал о работах Эйлера, Гамильтона и де Брюйна; никто и представить себе не мог, что последствия его собственных экспериментальных исследований в конечном итоге столкнут его лицом к лицу с этими математическими гигантами.

Десять лет назад Фредерик Сенгер секвенировал крошечный геном вируса φX174 длиной 5386 нуклеотидов. К концу 1980-х биологи регулярно секвенировали вирусы, содержащие сотни тысяч нуклеотидов, но идея секвенирования бактериальных (не говоря уже о человеческих) геномов оставалась недостижимой как в экспериментальном, так и в вычислительном смысле. Действительно, в конце 1980-х годов считывание одного рида стоило более доллара,

а всего генома млекопитающих оценивалось в миллиарды. Поэтому ДНК-чипы были изобретены с целью дешевого получения состава *k*-меров генома, хотя и с меньшей длиной считывания *k*, чем исходная технология секвенирования ДНК. Например, в то время как дорогостоящий метод секвенирования Сенгера в 1988 году генерировал риды длиной 500 нуклеотидов, изобретатели ДНК-чипов изначально стремились получить риды длиной всего 10.

ДНК-чипы работают следующим образом. Сначала мы синтезируем все 4^k возможных *k*-меров ДНК и присоединяем их к ДНК-чипу, который представляет собой сетку, где каждый *k*-мер получает уникальную позицию. Затем мы метим фрагмент одноцепочечной ДНК (с неизвестной последовательностью нуклеотидов) флуоресцентной меткой и наносим раствор, содержащий эту меченую ДНК, на ДНК-чип. *k*-меры во фрагменте ДНК будут гибридизоваться (связываться) со своими обратными комплементарными *k*-мерами на матрице. Все, что нам нужно сделать, – это использовать спектроскопию, чтобы проанализировать, какие участки на этом массиве дают флуоресцентный ответ на освещение; поэтому обратное дополнение *k*-меров, соответствующих этим сайтам, должно принадлежать (неизвестному) фрагменту ДНК. Таким образом, набор *k*-меров на матрице, дающий флуоресцентный отклик, показывает состав фрагмента ДНК (рис. 3.44).

AAA	AGA	CAA	CGA	GAA	GGA	TAA	TGA
AAC	AGC	CAC	CGC	GAC	GGC	TAC	TGC
AAG	AGG	CAG	CGG	GAG	GGG	TAG	TGG
AAT	AGT	CAT	CGT	GAT	GGT	TAT	TGT
ACA	ATA	CCA	CTA	GCA	GTA	TCA	TTA
ACC	ATC	CCC	CTC	GCC	GTC	TCC	TTC
ACG	ATG	CCG	CTG	GCG	GTG	TCG	TTG
ACT	ATT	CCT	CTT	GCT	GTT	TCT	TTT

Рис. 3.44 Матрица миниатюрного ДНК-чипа, содержащая все возможные 3-меры. Взятие обратных комплементов флуоресцентно меченных 3-меров дает набор {ACC, ACG, CAC, CCG, CGC, CGT, GCA, GTT, TAC, TTA}, который представляет собой 3-мер-композицию строки из 12 нуклеотидов CGCACGTTACCG. Обратите внимание, что этот ДНК-чип не дает информации о кратности 3-меров

Поначалу мало кто верил, что ДНК-чипы будут работать, потому что и биохимическая задача синтеза миллионов коротких фрагментов ДНК, и алгоритмическая задача реконструкции последовательности казались слишком сложными. В 1988 году журнал *Science* писал, что, учитывая объем работы, необходимой для синтеза массива ДНК, «использование ДНК-чипов для секвенирования означало бы просто замену одной ужасной задачи на другую». Оказалось, что *Science* был прав только наполовину. В середине 1990-х годов ряд компаний усовершенствовали технологии создания больших ДНК-чипов, но ДНК-чипы в конечном итоге не смогли реализовать мечту, которая мотивировала их изобретателей, потому что точность гибридизации ДНК с чипом была слишком низкой, а значение k было слишком маленьким.

И все же провал технологии ДНК-чипов впоследствии дал замечательный результат; в то время как первоначальная цель (секвенирование ДНК) оставалась недостижимой, появились два новых неожиданных применения ДНК-чипов. Сегодня они используются для измерения экспрессии генов, а также для анализа генетических вариаций. Эти неожиданные приложения превратили ДНК-чипы в многомиллиардную индустрию, в которую вошла компания Hyseq, основанная Радое Дрманаком, одним из первых изобретателей ДНК-чипов.

После основания фирмы Hyseq Дрманак не отказался от своей мечты об изобретении альтернативной технологии секвенирования ДНК. В 2005 году он основал Complete Genomics, одну из первых компаний, занимающихся секвенированием нового поколения (NGS). Complete Genomics, Illumina, Life Technologies и другие компании NGS впоследствии разработали технологию, позволяющую очень недорого читать почти все k-меры из генома, что наконец сделало возможным метод эйлеровой сборки. Хотя эти технологии сильно отличаются от технологии ДНК-чипов, предложенной в 1988 году, все еще можно признать интеллектуальное наследие ДНК-чипов в методах NGS, что свидетельствует о том, что хорошие идеи никогда не умирают, даже если они поначалу терпят неудачу.

Подобно ДНК-чипам, технологии NGS изначально генерировали миллионы довольно коротких, подверженных ошибкам ридов (длиной около 20 нуклеотидов), когда в 2005 году началась революция NGS. Однако всего за несколько лет компании NGS смогли увеличить длину ридов и повысить их точность на порядок. Более того, компании Pacific Biosciences и Oxford Nanopore Technologies уже генерируют безошибочные риды, содержащие тысячи нуклеотидов. Возможно, ваш собственный стартап найдет способ сгенерировать один рид, охватывающий весь геном, что сделает эту главу сноской в истории секвенирования генома. Что бы ни принесло будущее, недавние разработки в области NGS уже произвели революцию в геномике, и биологи готовятся собрать геномы всех видов млекопитающих на Земле, опираясь при этом на простую идею, выдвинутую Леонардом Эйлером в 1735 году.

Повторы в геноме человека

Транспозон – это фрагмент ДНК, который может менять свое положение в геноме, что часто приводит к дупликации (повтору). (Транспозоны также

известны под названием «прыгающие гены». – *Прим. ред.*) Транспозон, внедряющийся в ген, скорее всего, отключит этот ген. Заболевания, вызываемые транспозонами, включают гемофилию, порфирию, мышечную дистрофию Дюшенна и многие другие. Транспозоны составляют большую часть генома человека и делятся на два класса в соответствии с их механизмом транспозиции, которые можно описать как **ретротранспозоны** или как **ДНК-транспозоны**.

Ретротранспозоны копируются в два этапа: сначала они транскрибируются с ДНК на РНК, а затем полученная РНК **обратно транскрибируется** в ДНК с помощью **обратной транскриптазы**. Затем этот скопированный фрагмент ДНК вставляется в новое положение в геноме. Транспозоны ДНК не включают промежуточную РНК, а вместо этого катализируются **транспозазами**. Транспозаза вырезает ДНК-транспозон, который затем вставляется в новый участок генома, что приводит к повтору.

Первые транспозоны были обнаружены в кукурузе Барбарой МакКлинток, за что она была удостоена Нобелевской премии в 1983 году. Около 85 % генома кукурузы и 50 % генома человека состоят из транспозонов. Наиболее распространенным транспозоном у людей является последовательность Alu, длина которой составляет примерно 300 оснований и на которую приходится примерно миллион повторов (с мутациями) в геноме человека. **Транспозон Mariner –** еще один транспозон в геноме человека, содержащий около 14 000 повторов, что составляет 2,6 млн пар оснований. Транспозоны Mariner существуют у многих видов и даже могут передаваться от одного вида к другому. Транспозон Mariner был использован для разработки **системы транспозонов «Спящая красавица»**, синтетического транспозона ДНК, созданного для выведения новых признаков у животных и для генной терапии.

Графы

Использование слова «граф» в этой книге отличается от термина «график» в математике средней школы; мы не имеем в виду диаграмму данных. Вы можете думать о графе как о карте, показывающей города, соединенные дорогами.

Первая картинка на рис. 3.45 показывает шахматную доску 4×4 с удаленными угловыми квадратами. Конь может сделать два шага в любом из четырех направлений (влево, вправо, вверх и вниз), а затем один шаг в перпендикулярном направлении. Например, конь на поле 1 может пойти на поле 7 (два вниз и один влево), поле 9 (два вниз и один вправо) или поле 6 (два вправо и один вниз).

ОСТАНОВИТЕСЬ и задумайтесь. Может ли конь обойти эту доску, пройдя через каждую клетку ровно один раз и вернуться на ту же клетку, с которой он начал?

Вторая картинка на рис. 3.45 представляет каждую из 12 клеток шахматной доски в виде узла. Два узла соединены ребром, если конь может пройти между ними за один шаг. Например, узел 1 соединен с узлами 6, 7 и 9. Соединение узлов таким образом дает «граф коня», состоящий из 12 узлов и 16 ребер.

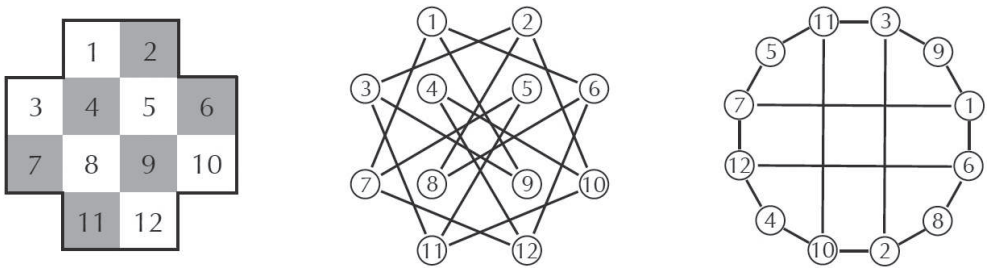


Рис. 3.45 (Слева) Воображаемая шахматная доска. (Посередине) Граф коня представляет каждую клетку узлом и соединяет два узла ребром, если конь может перемещаться между соответствующими клетками за один ход. (Справа) Эквивалентное представление «графа коня»

Мы можем описать граф его набором узлов и ребер, где каждое ребро записывается как пара узлов, которые оно соединяет. Граф на второй картинке рис. 3.45 описывается набором узлов

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$$

и следующим набором ребер:

$$\begin{array}{cccccccc} 1–6 & 1–7 & 1–9 & 2–3 & 2–8 & 2–10 & 3–9 & 3–11 \\ 4–10 & 4–12 & 5–7 & 5–11 & 6–8 & 6–12 & 7–12 & 10–11. \end{array}$$

Путь в графе – это последовательность ребер, где каждое последующее ребро начинается в узле, заканчивающем предыдущее ребро. Например, путь $8 \to 6 \to 1 \to 9$ на рис. 3.45 начинается в узле 8, заканчивается в узле 9 и состоит из 3 ребер. Пути, которые начинаются и заканчиваются в одном и том же узле, называются циклами. Цикл $3 \to 2 \to 10 \to 11 \to 3$ начинается и заканчивается в узле 3 и состоит из 4 ребер.

Способ построения графа не имеет значения; два графа с одним и тем же набором узлов и ребер эквивалентны, даже если конкретные изображения, представляющие граф, различны. Единственное, что важно, – это какие узлы соединены, а какие нет. Таким образом, граф на второй картинке рис. 3.45 идентичен графу на третьей. Этот граф показывает цикл, который посещает каждый узел графа коня один раз, и описывает последовательность ходов коня, которые посещают каждую клетку ровно один раз.

Упражнение. Сколько всего существует ходов конем на шахматной доске, изображенной на рис. 3.45?

Количество ребер, соединенных с данным узлом v, называется **степенью** v. Например, узел 1 в графе коня имеет степень 3, а узел 5 – степень 2. Сумма степеней всех 12 узлов в этом случае равна 32 (восемь узлов имеют степень 3 и четыре узла имеют степень 2), что в два раза превышает количество ребер в графе.

ОСТАНОВИТЕСЬ и задумайтесь. Сможете ли вы соединить семь телефонов таким образом, чтобы каждый из них был подключен ровно к трем другим?

Многие задачи биоинформатики анализируют направленные, или **ориентированные**, **графы**, в которых каждое ребро направлено от одного узла к другому, как показано стрелками на рис. 3.46. Вы можете думать об ориентированном графе как о карте, показывающей города, соединенные дорогами с односторонним движением. Каждый узел в ориентированном графе характеризуется «степенью вхождения», **indegree** (количество входящих ребер), и «степенью исхождения», **outdegree** (количество исходящих ребер).

ОСТАНОВИТЕСЬ и задумайтесь. Докажите, что для любого ориентированного графа сумма входящих степеней всех узлов равна сумме исходящих степеней всех узлов.

Неориентированный граф является **связным**, если каждые две вершины имеют соединяющий их путь. Несвязные графы можно разбить на непересекающиеся компоненты связности.

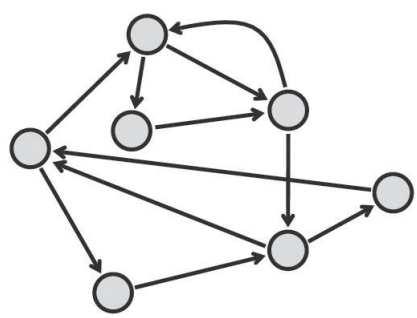


Рис. 3.46 Ориентированный граф

Игра «Икосиан»

Совершим исторический экскурс в Дублин, где в 1857 году ирландский математик Уильям Гамильтон создал **игру «Икосиан»** (Она также называется игрой «Вокруг света», или «Гамильтоновой игрой». – *Прим. ред.*). Эта игра, представляющая из себя коммерческий флоп, состоит из деревянной доски с 20 отверстиями для колышков и несколькими линиями, соединяющими отверстия, а также 20 пронумерованных колышков (рис. 3.47 (слева)). Задача заключается в том, чтобы поместить пронумерованные колышки в отверстия таким образом, чтобы колышек 1 соединился линией на доске с колышком 2, который в свою очередь соединился бы линией с колышком 3 и т. д., пока, наконец, колышек 20 будет соединен линией, возвращающейся к колышку 1. Другими словами, если мы будем двигаться по линиям на доске от колышка к колышку в порядке возрастания, мы достигнем каждого колышка ровно один раз, а затем вернемся к исходному.

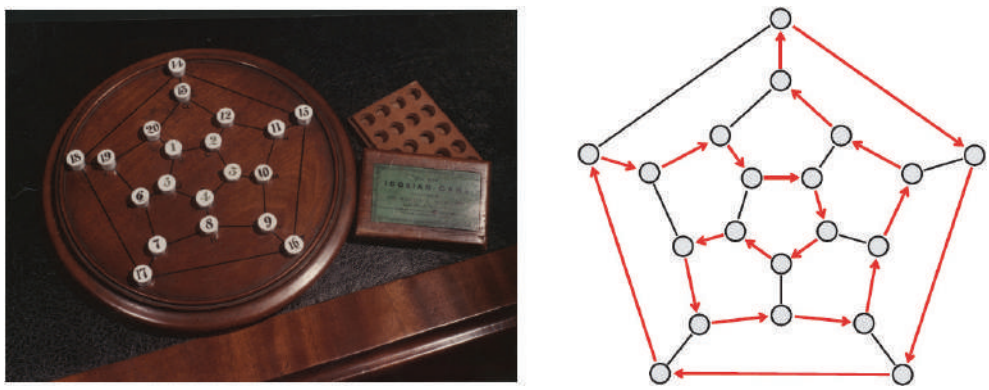


Рис. 3.47 (Слева) Игра «Икосиан», или игра Гамильтона, с показанным выигрышным размещением колышков. (Справа) Выигрышное размещение колышков можно представить в виде гамильтонова цикла на графе. Каждый узел в этом графе представляет собой отверстие для штифта на доске, а ребро соединяет два отверстия для штифта, которые соединены отрезком линии на доске

Мы можем смоделировать икосианскую игру с помощью графа, если представим каждое отверстие для колышков узлом, а затем преобразуем линии, соединяющие отверстия для колышков, в ребра, соединяющие соответствующие узлы. Этот граф имеет гамильтонов цикл, решающий игру «Икосиан»; один из них показан на рис. 3.47 (справа). Хотя грубый подход к задаче гамильтонова цикла прекрасно работает для этих маленьких графов, для больших графов он быстро становится практически неосуществимым.

Разрешимые и неразрешимые задачи

Вдохновленные теоремой Эйлера, вы, вероятно, задаетесь вопросом, существует ли такой простой результат, ведущий к быстрому алгоритму для задачи гамильтонова цикла. Основная проблема заключается в том, что, хотя мы руководствуемся теоремой Эйлера при решении задачи эйлерова цикла, аналогичное простое условие для задачи гамильтонова цикла остается неизвестным. Конечно, вы всегда можете перебрать все обходы графа и сообщить, если найдете гамильтонов цикл. Проблема с этим методом грубой силы заключается в том, что для графа, состоящего всего из тысячи узлов, проходов по графу может быть больше, чем атомов во Вселенной!

В течение многих лет задача гамильтонова цикла ускользала от всех попыток некоторых самых блестящих исследователей мира ее решить. После многих лет бесплодных усилий ученые-компьютерщики начали задаваться вопросом, является ли эта задача **разрешимой** (или нет!), т. е. что их неспособность найти алгоритм с полиномиальным временем не была связана с отсутствием глубокого понимания вопроса, а скорее с тем, что такого алгоритма для решения задачи гамильтонова цикла просто не существует. В 1970-х ученые-компьютерщики обнаружили еще тысячи алгоритмических задач с той же судьбой, что и задача гамильтонова цикла. Хотя эти задачи могут показаться простыми, никто не смог найти быстрые алгоритмы для их решения. Большое подмножество этих задач, включая задачу гамильтонова цикла, теперь вместе известны как ***NP*-полные задачи**. (NP означает «Non-deterministic Polynomial». – *Прим. ред.*) Формальное определение *NP*-полноты выходит за рамки этого текста.

Все *NP*-полные задачи *эквивалентны* друг другу: любая *NP*-полная задача может быть преобразована в любую другую *NP*-полную задачу за полиномиальное время. Таким образом, если вы найдете быстрый алгоритм для одной *NP*-полной задачи, вы автоматически сможете использовать этот алгоритм для разработки быстрого алгоритма для любой другой NP-полной задачи! Проблема эффективного решения *NP*-полных задач или окончательного доказательства их неразрешимости настолько фундаментальна, что Математический институт Клэя в 2000 году назвал ее одной из семи задач тысячелетия. Найдите эффективный алгоритм для любой *NP*-полной задачи или покажите, что одна из этих задач неразрешима, и институт Клэя наградит вас премией в один миллион долларов.

Подумайте дважды, прежде чем приступать к решению *NP*-полной задачи. Пока решена только одна из семи задач тысячелетия; в 2003 году Григорий Перельман доказал гипотезу Пуанкаре. Истинный математик, Перельман впоследствии отказался от приза в миллион долларов, полагая, что чистота и красота математики превыше любого материального вознаграждения.

NP-полные задачи относятся к более широкому классу сложных вычислительных задач. Задача *A* называется ***NP*-трудной задачей**, если существует некоторая *NP*-полная задача, которая может быть сведена к *A* за полиноми-

альное время. Поскольку *NP*-полные задачи можно свести друг к другу за полиномиальное время, все *NP*-полные задачи являются *NP*-трудными. Однако не каждая *NP*-трудная задача является *NP*-полной (это означает, что первые «по крайней мере так же сложно» решить, как и вторые). Одним из примеров *NP*-сложной задачи, которая не является *NP*-полной, является **Задача о коммивояжере**, в которой нам дан граф со взвешенными ребрами и мы должны создать гамильтонов цикл минимального общего веса.

От Эйлера до Гамильтона и де Брюйна

Эйлер представил свое решение задачи кенигсбергских мостов Императорской Российской академии наук в Санкт-Петербурге в 1735 году. На рис. 3.48 представлен рисунок Эйлера для задачи о семи мостах Кенигсберга.

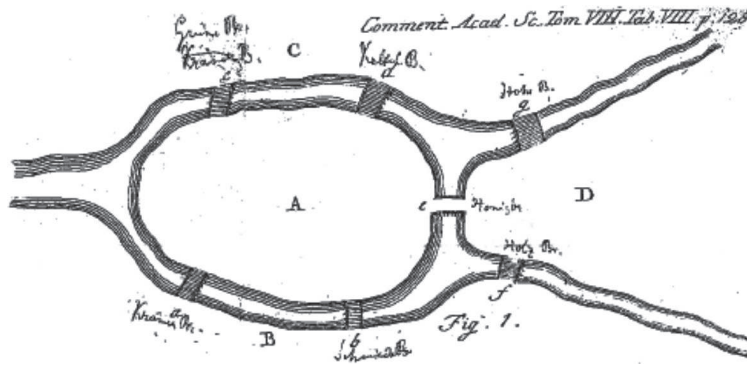


Рис. 3.48 Рисунок Кенигсберга, представленный Эйлером, который показывает каждую из четырех частей города, обозначенных A, B, C и D, а также семь мостов, перекинутых через рукава реки Прегель

Эйлер был самым плодовитым математиком всех времен; помимо теории графов, он впервые ввел обозначение $f(x)$ для представления функции, i для квадратного корня из -1 и π для обозначения отношения длины окружности к ее диаметру. Усердно работая на протяжении всей своей жизни, он ослеп на правый глаз в 1735 году. И продолжил работать. В 1766 году он потерял способность пользоваться и левым глазом и прокомментировал это так: *«Теперь у меня будет меньше отвлекающих внимание помех»*. Несмотря на это, он продолжил работу! Даже полностью ослепнув, он опубликовал сотни статей.

В этой главе мы познакомились с тремя математиками трех разных столетий – Эйлером, Гамильтоном и де Брюйном, жившими в разное время в разных частях Европы (рис. 3.49). Мы чувствуем причастность к ним, их работе и в том, как она сходится к этой единственной точке в современной биологии, к которой пришли мы. Однако первые биологи, занимавшиеся секвенировани-

ем ДНК, понятия не имели, как можно применить к своему предмету теорию графов. Более того, первая статья, в которой идеи трех математиков применялись к сборке генома, была опубликована спустя много лет после смерти Эйлера и Гамильтона, когда де Брюйну было уже за семьдесят. Так что, возможно, мы могли бы думать об этих героях не как об искателях приключений, а как об одиноких странниках. Как это часто бывает с проклятием математиков, каждый из них страстно занимался абстрактными вопросами, не имея ни малейшего представления, куда однажды могут привести эти темы, когда их уже не будет.

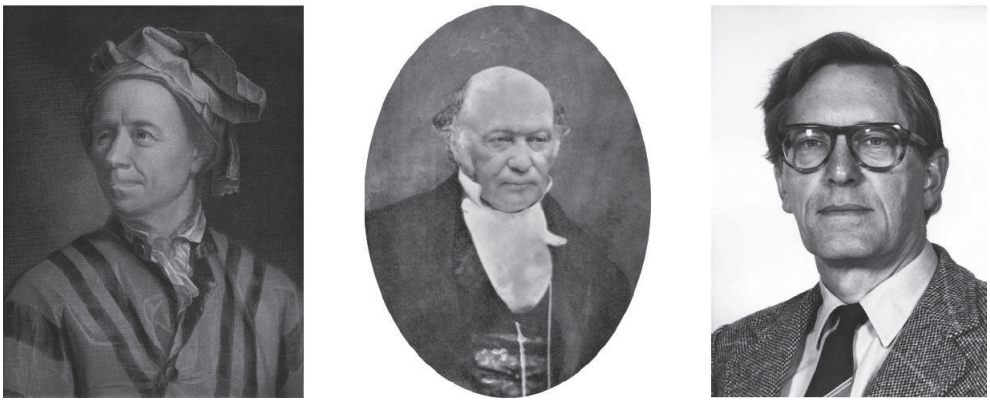


Рис. 3.49 Леонард Эйлер (слева), Уильям Гамильтон (в центре) и Николаас де Брюйн (справа)

Семь мостов Калининграда

Кенигсберг был сильно разрушен во время Второй мировой войны; его руины были захвачены Советской армией. Город был переименован в Калининград в 1946 году в честь советского революционера и государственного деятеля Михаила Калинина.

С XVIII века в планировке Кенигсберга многое изменилось, и так уж получилось, что нарисованный сегодня граф моста для города Калининград до сих пор не содержит эйлерова цикла. Однако этот граф содержит эйлеров путь, а это означает, что жители Калининграда могут пройти по каждому мосту ровно один раз, но при этом не могут вернуться к тому месту, с которого начали. Таким образом, калининградцы наконец-то добились хотя бы небольшой части цели, поставленной кенигсбергцами (правда, домой им придется ехать на такси). Тем не менее прогуливаться по Калининграду не так приятно, как в 1735 году, поскольку прекрасный старый Кенигсберг был разрушен бомбардировками союзников (Авиацией Великобритании и США. – *Прим. ред.*) в 1944 году и ужасной советской архитектурой в последующие после Второй мировой войны годы. (В этом с автором книги трудно поспорить. – *Прим. ред.*)

Подводные камни сборки двухцепочечной ДНК

Чтобы собрать настоящие геномы, биоинформатики должны обрабатывать риды обеих цепей ДНК, не зная заранее, из какой цепи происходит каждый рид. Чтобы решить эту задачу, они сначала добавляют комплемент каждого рида к коллекции ридов, эффективно удваивая количество ридов. В идеальном мире граф де Брюйна, сформированный из всех этих ридов, состоял бы из двух (топологически идентичных, но по-разному помеченных) связанных компонентов, по одному для каждой цепи ДНК (рис. 3.50).

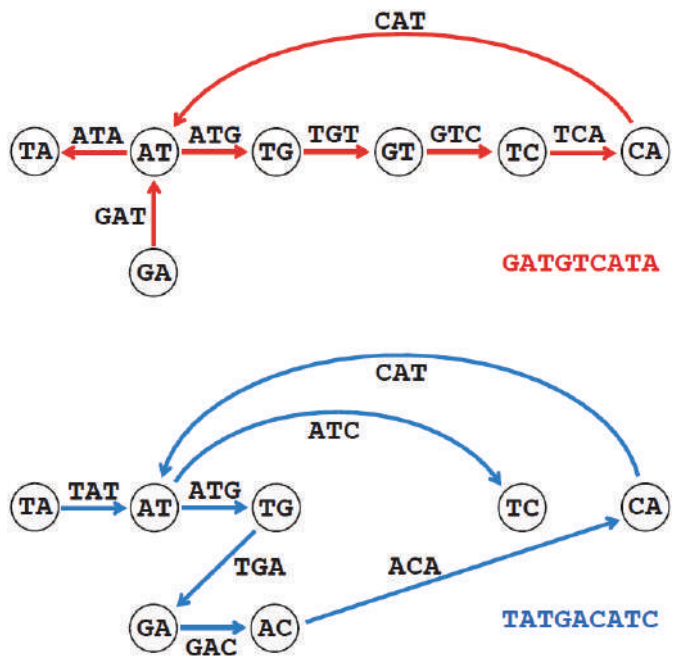


Рис. 3.50 Реконструкция геномов **GATGTCATA** и **TATGACATC** из их графов $DeBruijn_3$(**GATGTCATA**) (вверху) и $DeBruijn_3$(**TATGACATC**) соответственно (внизу) может быть выполнена легко, поскольку существует только один способ обхода каждого графа

В действительности эти два компонента будут склеены между собой, поскольку внутри одной цепи генома имеется множество комплементарных *k*-меров. Действительно, кроме прямых повторов (как ATG в **GATGTATGA**), геномы имеют много комплементарных повторов, в которых одна подстрока является комплементом другой (например, ATG/CAT в **GATGTATGA**). В результате, в то время как одиночная цепь **GATGTATGA** не имеет повторяющихся 3-меров, что делает ее сборку тривиальной, комплементарные цепи **GATGTATGA** и **TATGACATC** имеют повторяющиеся 3-меры. На рис. 3.51 пока-

зан граф де Брюйна для комплементарных цепей GATGTATGA и TATGACATC, который требует склеивания узлов из двух разных цепей, что усложняет сборку.

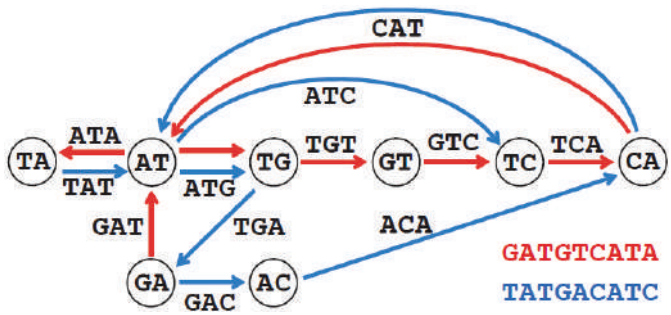
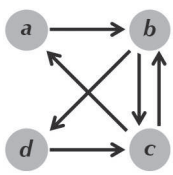


Рис. 3.51 Граф де Брюйна, образованный объединением 3-меров из GATGTATGA и его комплемента TATGACATC. Реконструкция исходного генома теперь является нетривиальной задачей

«ЛУЧШАЯ» теорема

Имея матрицу смежности $A(G)$ ориентированного эйлерова графа G, определим матрицу $A^*(G)$, заменив i-й диагональный элемент $-A(G)$ на $Indegree(i)$ для каждого узла i в G (рис. 3.52).



$A(G)$

	a	**b**	**c**	**d**
a	0	1	0	0
b	0	0	1	1
c	1	1	0	0
d	0	0	1	0

$A^*(G)$

	a	**b**	**c**	**d**
a	1	−1	0	0
b	0	2	−1	−1
c	−1	−1	2	0
d	0	0	−1	1

Рис. 3.52 (слева) Граф G с двумя эйлеровыми циклами. (В центре) Матрица смежности $A(G)$ группы G. (Справа) Матрица $A^*(G)$. Каждый i-кофактор $A^*(G)$ равен 2. Таким образом, «ЛУЧШАЯ» теорема вычисляет количество эйлеровых циклов как $2 \cdot 0! \cdot 1! \cdot 1! \cdot 0! = 2$

Так называемый ***i*-кофактор** матрицы M – это определитель матрицы, полученной из M удалением i-й строки и i-го столбца. Можно показать, что для данного эйлерова графа G все i-кофакторы $A^*(G)$ имеют одно и то же значение, которое мы обозначаем как $c(G)$.

Следующая теорема дает формулу, вычисляющую количество эйлеровых циклов в графе. Ее название (**BEST**, «лучшая») является аббревиатурой ее первооткрывателей: de **B**ruijn, van Aardenne-**E**hrenfest, **S**mith, and **T**utte.

«ЛУЧШАЯ» теорема. Количество эйлеровых циклов в эйлеровом графе равно

$$c(G) \cdot \prod_{\text{all nodes } v \text{ in graph } G} (Indegree(v) - 1)!$$

«ЛУЧШАЯ» теорема, доказательство которой выходит за рамки этого текста, предоставляет нам альтернативный способ построения всех эйлеровых циклов в эйлеровом ориентированном графе.

Библиографические примечания

После работы Эйлера по задаче кенигсбергских мостов (Eiler, 1758[1]) теория графов была забыта более чем на сто лет, но возродилась во второй половине XIX века. Теория графов процветала в XX веке, когда она стала важной областью математики со многими практическими приложениями. Граф де Брюйна был введен независимо Николаасом де Брюйном (de Bruijn, 1946[2]) и И. Дж. Гудом (Good, 1946[3]).

Методы секвенирования ДНК были изобретены независимо и одновременно в 1977 году группами под руководством Фредерика Сенгера (Sanger, Nicklen, and Coulson, 1977[4]) и Уолтера Гилберта (Maxam and Gilbert, 1977[5]). Годом ранее Уолтер Флайерс и его коллеги секвенировали более короткий вирус под названием MS2, но метод Сенгера был адаптирован для более длинных геномов. ДНК-чипы были предложены одновременно и независимо друг от друга в 1988 году Радое Дрманаком (Drmanac et al., 1989[6]), Андреем Мирзабековым (Lysov et al., 1988[7]) и Эдвином Саузерн (Southern, 1988[8]). Эйлеровский подход к ДНК-чипам был описан в 1989 году (Pevzner, 1989[9]).

---

1 https://scholarlycommons.pacific.edu/cgi/viewcontent.cgi?article=1052&context=euler-works.
2 https://pure.tue.nl/ws/files/4442708/597473.pdf.
3 https://academic.oup.com/jlms/article/s1-21/3/167/845002.
4 https://www.ncbi.nlm.nih.gov/pubmed/271968.
5 https://www.ncbi.nlm.nih.gov/pubmed/265521.
6 https://www.sciencedirect.com/science/article/pii/0888754389902905.
7 https://www.tandfonline.com/doi/abs/10.1080/07391102.1994.10508033?journalCode=tbsd20.
8 https://patents.google.com/patent/US20040259119.
9 https://www.ncbi.nlm.nih.gov/pubmed/2684223.

Эйлеровский подход к секвенированию ДНК был описан Idury and Waterman, 1995[1], и далее развит Pevzner, Tang, and Waterman, 2001[2]. Чтобы решить проблему сборки из коротких ридов, полученных с помощью технологий секвенирования следующего поколения, был разработан ряд инструментов сборки, основанных на графах де Брюйна (Zerbino and Birney, 2008[3], Butler et al., 2008[4]). Парные графы де Брюйна были введены Medvedev et al., 2011[5].

Система транспозонов «Спящая красавица» была разработана Ivics et al., 1997[6].

[1] https://www.ncbi.nlm.nih.gov/pubmed/7497130.
[2] https://www.ncbi.nlm.nih.gov/pubmed/11504945.
[3] https://www.ncbi.nlm.nih.gov/pubmed/18349386.
[4] https://www.ncbi.nlm.nih.gov/pubmed/18340039.
[5] https://www.ncbi.nlm.nih.gov/pubmed/21999285.
[6] https://www.ncbi.nlm.nih.gov/pubmed/9390559.


Глава 4
Как мы секвенируем антибиотики?
Алгоритмы грубой силы
• 147 • 119 • 128 •
Два человека и одна обезьяна
в самом фантастическом
приключении их жизней...

Открытие антибиотиков

В августе 1928 года перед отъездом в отпуск шотландский микробиолог Александр Флеминг поместил свои культуры стафилококков, вызывающих инфекцию, на лабораторный стол. Вернувшись к работе несколько недель спустя, Флеминг заметил, что одна культура была заражена грибком *Penicillium*, а окружающая ее колония стафилококка уничтожена! Флеминг назвал вещество, убивающее бактерии, **пенициллином** и предположил, что его можно использовать для лечения бактериальных инфекций у людей.

Когда Флеминг опубликовал свое открытие в 1929 году, его статья не вызвала немедленного отклика. Последующие эксперименты изо всех сил пытались выделить **антибиотик** (т. е. химическое соединение, которое убивало бактерии) из самого грибка. В результате Флеминг в конце концов пришел к выводу, что пенициллин нельзя практически применять для лечения бактериальных инфекций, и отказался от своих исследований антибиотиков.

После начала Второй мировой войны в поисках новых лекарств для лечения раненых солдат американское и британское правительства активизировали поиск антибиотиков; однако проблема их массового производства оставалась. В марте 1942 года половина всего запаса пенициллина, принадлежащего фармацевтическому гиганту Merck, была использована для лечения всего лишь одного инфицированного пациента.

В том же 1942 году русские биологи Георгий Гаузе и Мария Бражникова заметили, что бактерия *Bacillus brevis* убивает патогенную бактерию *Staphylococcus aureus*. В отличие от усилий Флеминга с пенициллином они успешно выделили антибиотическое соединение из *Bacillus brevis* и назвали его «советским грамицидином» (*Gramicidin Soviet*). В течение года этот антибиотик был распределен по советским военным госпиталям.

Тем временем американские ученые прочесывали различные продовольственные рынки в поисках испорченных продуктов и наконец нашли в Иллинойсе заплесневелую дыню с высокой концентрацией пенициллина. Это открытие позволило Соединенным Штатам произвести 2 млн доз пенициллина ко времени вторжения союзников в Нормандию в 1944 году, и это спасло жизни тысяч раненых солдат.

Гаузе продолжил свои исследования советского грамицидина после Второй мировой войны, но не смог выяснить его химическую структуру. Приняв эстафету от Гаузе, английский биохимик Ричард Синг изучил советский грамицидин и широкий спектр других антибиотиков, продуцируемых *Bacillus brevis*. Через несколько лет после окончания Второй мировой войны он продемонстрировал, что они представляют собой короткие цепи аминокислот (т. е. мини-белки), называемые пептидами. Гаузе получил Сталинскую премию в 1946 году, а Синг – Нобелевскую премию в 1952 году. Сталинская премия оказалась очень ценной, поскольку защитила Гаузе от репрессий в период «лысенковщины», кампании против «буржуазных» генетиков, усилившейся в послевоенное время. Подробнее см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Гаузе и «лысенковщина»**.

Массовое производство антибиотиков положило начало эволюционной гонке вооружений между фармацевтическими компаниями и патогенными бактериями. Первые работали над созданием новых антибиотиков, а вторые приобретали устойчивость к этим препаратам. Хотя современная медицина побеждала во всех битвах на протяжении шести десятилетий, за последние десять лет наблюдается тревожный рост устойчивых к антибиотикам бактериальных инфекций, которые не поддаются лечению даже самыми сильными антибиотиками. В частности, бактерия *Staphylococcus aureus*, которую Гаузе изучал в 1942 году, мутировала в устойчивый штамм, известный как **устойчивый к метициллину *Staphylococcus aureus* (MRSA)**. В настоящее время MRSA является основной причиной смерти от инфекций в больницах; уровень смертности даже превысил уровень смертности от СПИДа в Соединенных Штатах.

С появлением MRSA разработка новых антибиотиков представляет собой центральную проблему для современной медицины. Трудной проблемой в исследованиях антибиотиков является **секвенирование** недавно открытых антибиотиков или определение порядка аминокислот, составляющих пептид антибиотика.

Как бактерии производят антибиотики?

Как пептиды кодируются геномом

Начнем с рассмотрения **тироцидина B1**, одного из многих антибиотиков, продуцируемых *Bacillus brevis*. Тироцидин B1 определяется цепью из 10 аминокислот, показанной ниже (с использованием как однобуквенных, так и трехбуквенных обозначений аминокислот). Наша цель в этом разделе – выяснить, как *Bacillus brevis* могла создать этот антибиотик.

```
Val-Lys-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr
 V   K   L   F   P   W   F   N   Q   Y
```

Рис. 4.1 Тироцидин B1

Центральная догма молекулярной биологии гласит, что «ДНК производит РНК, РНК делает белок». Согласно центральной догме, ген из генома сначала **транскрибируется** в цепь РНК, состоящую из четырех **рибонуклеотидов**: аденина, гуанина, цитозина и урацила. Нить РНК может быть представлена как цепь РНК, состоящая из четырехбуквенного алфавита {A, C, G, U}. Вы можете думать о геноме как о большой поваренной книге, и в этом случае ген и его РНК-транскрипт образуют рецепт в этой поваренной книге. Затем транскрипт РНК **транслируется** в аминокислотную последовательность белка.

Химический механизм, лежащий в основе транскрипции и трансляции, очень похож на репликацию, но с вычислительной точки зрения оба процесса более просты. Транскрипция просто преобразует цепь ДНК в цепь РНК, заменяя все вхождения T на U. Полученная цепь РНК транслируется в аминокислотную последовательность следующим образом. Во время трансляции цепь РНК разделяется на неперекрывающиеся 3-меры, называемые **кодонами**. Затем

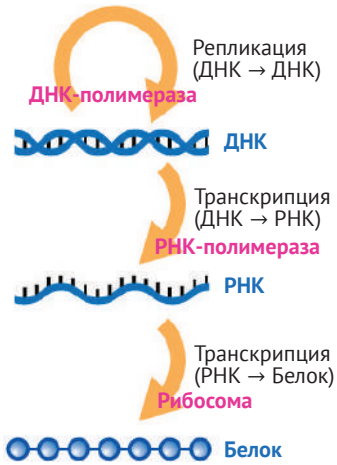


Рис. 4.2 Репликация, транскрипция, трансляция

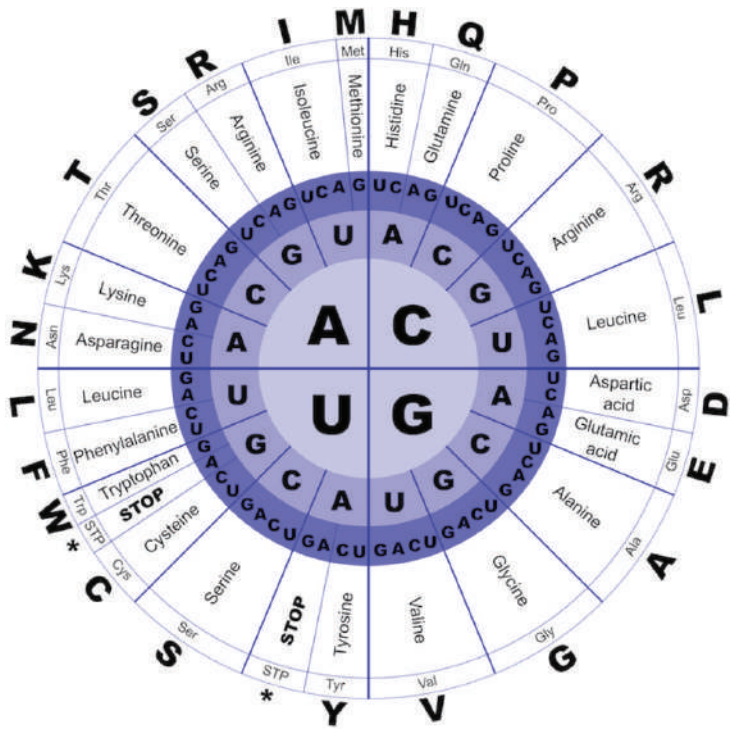


Рис. 4.3 Генетический код определяет трансляцию 3-мера РНК (кодона) в одну из 20 различных аминокислот. Первые три круга, идущие изнутри наружу, представляют собой 1-й, 2-й и 3-й нуклеотиды кодона. 4-й, 5-й и 6-й круги определяют переведенную аминокислоту тремя способами: полное название аминокислоты, ее трехбуквенное сокращение и ее однобуквенное сокращение. Три из 64 кодонов РНК являются стоп-кодонами, которые останавливают трансляцию. Воспроизведено из Open Clip Art

каждый кодон преобразуется в одну из 20 аминокислот посредством **генетического кода**; результирующая последовательность может быть представлена в виде **цепи аминокислот**, состоящей из 20-буквенного алфавита. Как показано на рисунке ниже, каждый из 64 кодонов РНК кодирует свою собственную аминокислоту (некоторые кодоны кодируют одну и ту же аминокислоту), за исключением трех **стоп-кодонов**, которые не транслируются в аминокислоты и служат для остановки трансляции. (Для получения дополнительной информации см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Открытие кодонов** и **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Разделение генов**.) Например, цепь ДНК **TATACGAAA** транскрибируется в цепь РНК **UAUACGAAA**, которая, в свою очередь, транслируется в аминокислотную цепь Y-T-K.

Мы будем представлять генетический код в виде массива *GeneticCode*, содержащего 64 элемента, как показано ниже.

0	AAA	K	16	CAA	Q	32	GAA	E	48	UAA	*
1	AAC	N	17	CAC	H	33	GAC	D	49	UAC	Y
2	AAG	K	18	CAG	Q	34	GAG	E	50	UAG	*
3	AAU	N	19	CAU	H	35	GAU	D	51	UAU	Y
4	ACA	T	20	CCA	P	36	GCA	A	52	UCA	S
5	ACC	T	21	CCC	P	37	GCC	A	53	UCC	S
6	ACG	T	22	CCG	P	38	GCG	A	54	UCG	S
7	ACU	T	23	CCU	P	39	GCU	A	55	UCU	S
8	AGA	R	24	CGA	R	40	GGA	G	56	UGA	*
9	AGC	S	25	CGC	R	41	GGC	G	57	UGC	C
10	AGG	R	26	CGG	R	42	GGG	G	58	UGG	W
11	AGU	S	27	CGU	R	43	GGU	G	59	UGU	C
12	AUA	I	28	CUA	L	44	GUA	V	60	UUA	L
13	AUC	I	29	CUC	L	45	GUC	V	61	UUC	F
14	AUG	M	30	CUG	L	46	GUG	V	62	UUG	L
15	AUU	I	31	CUU	L	47	GUU	V	63	UUU	F

Рис. 4.4 *Массив GeneticCode* содержит 64 элемента, каждый из которых представляет собой аминокислоту или стоп-кодон (обозначается *)

В следующей задаче вам нужно найти трансляцию строки РНК в строку аминокислот.

Задача трансляции белка: *транслировать строку РНК в строку аминокислот.*

Input: паттерн строки РНК и массив *GeneticCode*.

Output: трансляция *Pattern* в аминокислотную строку *Peptide*.

Упражнение. Сколько цепей ДНК длиной 30 транскрибируется и транслируется в тироцидин B1 (не считая стоп-кодон)?

Напомним, что аминокислотная последовательность тироцидина B1 представляет собой Val-Lys-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr (рис. 4.3).

Где в геноме Bacillus brevis закодирован тироцидин?

Тысячи различных 30-меров ДНК могут кодировать тироцидин B1, и мы хотели бы знать, какой из них присутствует в геноме *Bacillus brevis*. Есть три разных способа разделить цепь ДНК на кодоны для трансляции по одному, начиная с каждой из первых трех начальных позиций цепи. Эти различные способы разделения цепи ДНК на кодоны называются **рамками считывания**. Поскольку ДНК двухцепочечная, геном имеет шесть рамок считывания (по три на каждой цепи), как показано на рисунке ниже.

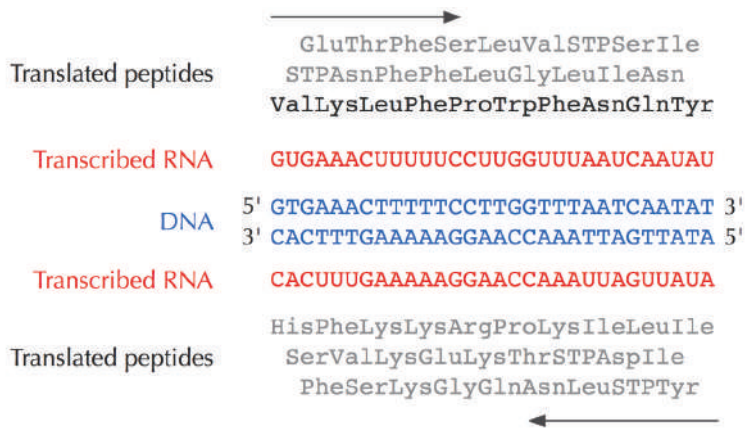


Рис. 4.5 Шесть рамок считывания дают шесть различных способов транскрипции и трансляции одного и того же фрагмента ДНК (по три из каждой цепи). Три верхние строки аминокислот читаются слева направо, тогда как нижние три строки читаются справа налево. Выделенная строка аминокислот указывает последовательность тироцидина B1. Стоп-кодоны обозначены как STP

Мы говорим, что последовательность *Pattern* цепи ДНК **кодирует** пептид цепи аминокислот, если цепь РНК, транскрибируемая либо из *Pattern*, либо из его обратного комплемента, транслируется в пептид. Например, строка ДНК GAAACT транскрибируется в GAAACU и транслируется в ET. Реверсный комплемент этой цепи ДНК, AGTTTC, транскрибируется в AGUUUC и транслируется в SF. Таким образом, GAAACT кодирует как ET, так и SF.

Задача кодирования пептидов: *найти подстроки генома, кодирующие заданную аминокислотную последовательность.*

Input: строка ДНК Text, строка аминокислот *Peptide* и массив *GeneticCode*.

Output: все подстроки *Text* содержащие *Peptide* (если такие подстроки существуют).

Упражнение. Решить задачу кодирования пептидов для *Bacillus brevis* и тироцидина B1 (Val-Lys-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr). Сколько стартовых позиций в *Bacillus brevis* кодирует этот пептид (рис. 4.3)?

☞ **Загрузите данные 4.1. Геном *Bacillus brevis***

После решения проблемы кодирования пептида для тироцидина B1 мы должны найти 30-мер в геноме *Bacillus brevis*, кодирующий тироцидин B1, но такого 30-мера не существует!

ОСТАНОВИТЕСЬ и задумайтесь. Как бактерия могла сделать пептид, который не кодируется бактериальным геномом?

От линейных к циклическим пептидам

Ни Гауз, ни Синг не знали об этом, но тироцидины и грамицидины на самом деле являются циклическими пептидами; циклическое представление тироцидина B1 показано слева на рис. 4.6. Таким образом, тироцидин B1 имеет десять различных линейных представлений, и мы должны запустить задачу кодирования пептидов для каждой из этих последовательностей, чтобы найти потенциальные 30-меры, кодирующие тироцидин B1. Тем не менее, когда мы решаем задачу кодирования пептидов для каждой из десяти строк справа на рисунке ниже, мы не находим 30-мерного генома *Bacillus brevis*, кодирующего тироцидин B1!

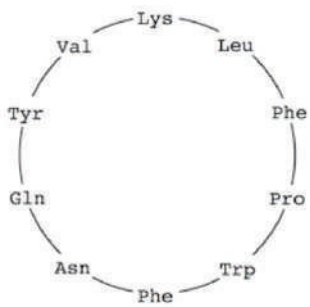


Рис. 4.6 Тироцидин B1 представляет собой циклический пептид (слева), поэтому он имеет десять различных линейных представлений (справа)

Уклоняясь от центральной догмы молекулярной биологии

Надеюсь, вы озадачены, потому что центральная догма молекулярной биологии подразумевает, что все пептиды должны кодироваться геномом. Лауреат Нобелевской премии Эдвард Тейтум был так же сбит с толку и в 1963 году придумал остроумный эксперимент. Трансляция белка осуществляется молекулярной машиной, называемой **рибосомой**, и поэтому Татум рассудил, что, если он заблокирует рибосому, все производство белка в *Bacillus brevis* должно прекратиться. К его изумлению, производство почти всех белков действительно остановилось, но тироцидинов и грамицидинов – продолжилось! Этот эксперимент привел Татума к гипотезе о том, что эти пептиды должны собираться каким-то еще неизвестным, нерибосомным молекулярным механизмом.

В 1969 году Фриц Липманн (еще один лауреат Нобелевской премии) продемонстрировал, что тироцидины и грамицидины являются **нерибосомными пептидами** (**NRPs**), синтезируемыми не рибосомой, а гигантским белком, называемым **NRP-синтетазой**. Этот фермент объединяет пептиды антибиотиков без какой-либо зависимости от РНК или генетического кода! Теперь мы знаем, что каждая NRP-синтетаза собирает пептиды, выращивая их по одной аминокислоте за раз, как показано на рисунке ниже.

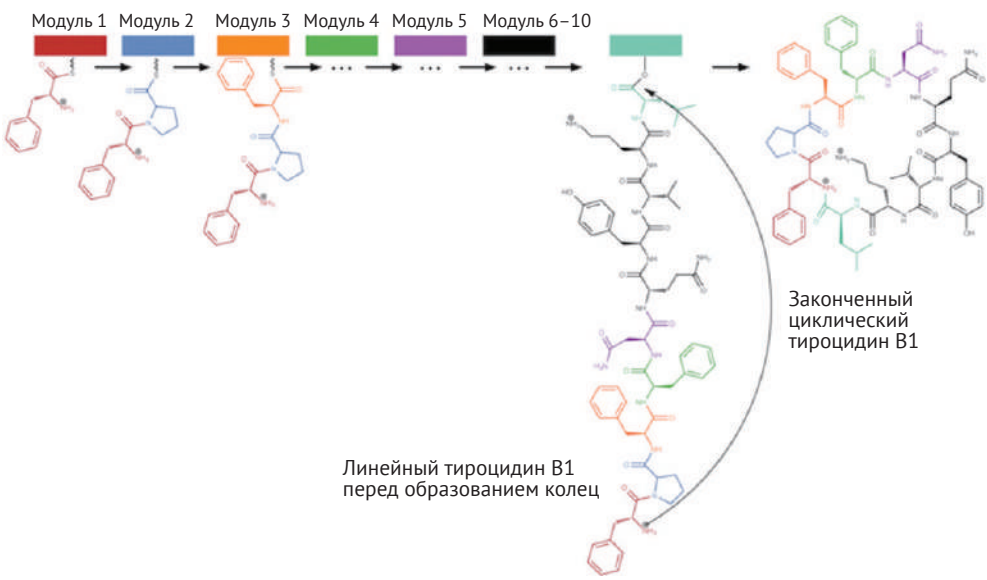


Рис. 4.7 NRP-синтетаза представляет собой гигантский многомодульный белок, который поэтапно собирает циклический пептид, по одной аминокислоте за раз. Каждый из десяти различных модулей (показанных разными цветами) добавляет одну аминокислоту к пептиду, который на рисунке является одним из многих тироцидинов, продуцируемых *Bacilus brevis*. На заключительном этапе пептидное кольцо цикла замыкается

Причина, по которой многие NRP применяются в фармацевтике, заключается в том, что они были оптимизированы за эоны эволюции как «молекулярные пули», которые бактерии и грибы используют для уничтожения своих врагов. Если эти враги окажутся патогенами, исследователи будут стремиться использовать эти пули в качестве антибактериальных препаратов. Однако NRP не ограничиваются антибиотиками: многие из них представляют собой противоопухолевые агенты и иммунодепрессанты, а другие используются бактериями для связи с другими клетками. Дополнительные сведения см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Определение кворума**.

Секвенирование антибиотиков путем их дробления на части

Введение в масс-спектрометрию

Поскольку NRP не придерживаются центральной догмы, мы не можем вывести их из генома, что возвращает нас к тому, с чего мы начали. Что делает секвенирование этих пептидов еще более трудным, так это то, что многие NRP (включая тироцидины и грамицидины) являются циклическими. Таким образом, стандартные инструменты для секвенирования линейных пептидов, которые мы опишем в следующей главе, неприменимы к анализу NRP.

Рабочей лошадкой секвенирования пептидов является применение масс-спектрометра, дорогостоящих молекулярных весов, которые разбивают молекулы на части, а затем определяют массы полученных фрагментов. Масс-спектрометр измеряет массу молекулы в дальтонах (Да); 1 Да приблизительно равен массе одного нуклона (протона или нейтрона).

Мы аппроксимируем массу молекулы, просто добавляя количество протонов и нейтронов, находящихся в составных атомах молекулы, что дает целочисленную массу молекулы. Например, аминокислота глицин, имеющая химическую формулу C2H3ON, имеет целочисленную массу 57, поскольку $2 \cdot 12 + 3 \cdot 1 + 1 \cdot 16 + 1 \cdot 14 = 57$. Однако 1 Да не совсем равен массе протона/нейтрона, и нам может понадобиться учитывать различные встречающиеся в природе изотопы каждого атома при взвешивании молекулы (подробнее см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Молекулярная масса.**). В результате аминокислоты обычно имеют нецелые массы (например, глицин имеет общую целочисленную массу, равную примерно 57,02 Да); однако для простоты мы будем работать с таблицей целочисленных масс, приведенной ниже.

G	A	S	P	V	T	C	I	L	N	D	K	Q	E	M	H	F	R	Y	W
57	71	87	97	99	101	103	113	113	114	115	128	128	129	131	137	147	156	163	186

Тироцидин B1, представленный последовательностью VKLFPWFNQY, имеет общую массу 1322 Да (99 + 128 + 113 + 147 + 97 + 186 + 147 + 114 + 128 + 163 = 1322).

Масс-спектрометр может разбить каждую молекулу тироцидина B1 на два линейных фрагмента и анализирует образцы, которые могут содержать миллиарды идентичных копий пептида, причем каждая копия распадается по-своему. Одна копия может разбиваться на LFP и WFNQYVK (с соответствующими массами 357 и 965), а другая может разбиваться на PWFN и QYVKLF. Наша цель – использовать массу этих и других фрагментов для секвенирования пептида. Совокупность масс всех осколков, сгенерированных масс-спектрометром, называется экспериментальным спектром.

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем использовать экспериментальный спектр для секвенирования пептида?

Задача секвенирования циклопептидов

Пока для простоты будем считать, что масс-спектрометр разрывает копии циклического пептида по любым возможным связям, так что результирующий экспериментальный спектр содержит массы всех возможных линейных фрагментов пептида, называемых **субпептидами**. Например, циклический пептид NQEL имеет 12 возможных субпептидов: N, Q, E, L, NQ, QE, EL, LN, NQE, QEL, ELN и LNQ. Мы также предполагаем, что субпептиды могут встречаться более одного раза, если аминокислота встречается в пептиде несколько раз (например, ELEL также имеет 12 субпептидов: E, L, E, L, EL, LE, EL, LE, ELE, LEL, ELE и LEL).

Упражнение. Сколько субпептидов имеет циклический пептид длины n?

Теоретический спектр циклического пептида *Peptide*, обозначаемого *Cyclospectrum*(*Peptide*), представляет собой совокупность всех масс его субпептидов в дополнение к массе 0 и массе всего пептида, причем массы упорядочены от наименьшей к наибольшей. Будем считать, что теоретический спектр может содержать повторяющиеся элементы, как в случае NQEL (показан ниже), где **NQ** и **EL** имеют одинаковую массу.

	L	N	Q	E	LN	**NQ**	**EL**	QE	LNQ	ELN	QEL	NQE	NQEL
0	113	114	128	129	227	**242**	**242**	257	355	356	370	371	484

Задача генерации теоретического спектра: *сгенерировать теоретический спектр циклического пептида.*

Input: аминокислотная строка *Peptide*.

Output: *Cyclospectrum*(*Peptide*).

Создать теоретический спектр известного пептида несложно, но наша цель – решить обратную задачу восстановления неизвестного пептида по его *экспериментальному* спектру. Мы начнем с предположения, что биологу посчастливилось создать **идеальный спектр**, совпадающий с теоретическим спектром пептида.

Задача секвенирования циклопептида: *при заданном идеальном спектре найти циклический пептид, теоретический спектр которого соответствует экспериментальному спектру.*

Input: набор (возможно, повторяющихся) целых чисел *Spectrum*, соответствующих экспериментальному спектру.

Output: аминокислотная строка *Peptide* такая, что *Cyclospectrum*(*Peptide*) = *Spectrum* (если такая строка существует).

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите теоретический спектр тироцидина B1, показанный ниже; если бы в результате эксперимента был получен такой спектр, как бы вы реконструировали аминокислотную последовательность тироцидина B1?

0	97	99	113	114	128	128	147	147	163	186	227
241	242	244	260	261	262	283	291	333	340	357	388
389	390	390	405	430	430	447	485	487	503	504	518
543	544	552	575	577	584	631	632	650	651	671	672
690	691	738	745	747	770	778	779	804	818	819	835
837	875	892	892	917	932	932	933	934	965	982	989
1031	1039	1060	1061	1062	1078	1080	1081	1095	1136	1159	1175
1175	1194	1194	1208	1209	1223	1225	1322				

С этого момента мы будем иногда работать непосредственно с массами аминокислот, позволив себе представлять пептид последовательностью целых чисел, обозначающих массы аминокислот, входящих в состав пептида. Например, мы представляем NQEL как 114-128-129-113 и тироцидин B1 (VKLFPWFNQY) как 99-128-113-147-97-186-147-114-128-163. Поэтому мы заменили алфавит из 20 аминокислот на алфавит только из 18 целых чисел, потому что две пары аминокислот имеют одинаковую целочисленную массу:

G	A	S	P	V	T	C	**I/L**	N	D	**K/Q**	E	M	H	F	R	Y	W
57	71	87	97	99	101	103	**113**	114	115	**128**	129	131	137	147	156	163	186

Обратите внимание, что в общем случае (когда мы не ограничены алфавитом аминокислот) задача секвенирования циклопептидов может иметь несколько решений. Например, пептиды 1-1-3-3 и 1-2-1-4 имеют одинаковый теоретический спектр {1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7 }.

ОСТАНОВИТЕСЬ и задумайтесь. Мы не знаем, существуют ли разные пептиды (в алфавите из 18 аминокислотных масс) с одинаковыми теоретическими спектрами, – сможете ли вы найти такие пептиды? **Примечание.** На этот вопрос может быть трудно ответить.

Алгоритм грубой силы для секвенирования циклопептидов

Впервые мы столкнулись с алгоритмами грубой силы в задаче поиска мотивов. В этой главе мы обсудим, как ускорить алгоритмы грубой силы для секвенирования пептидов, чтобы сделать их практически применимыми.

Давайте разработаем простой алгоритм грубой силы для задачи секвенирования циклопептидов. Мы обозначаем общую массу *Peptide* цепи аминокислот как *Mass*(*Peptide*). В экспериментах по масс-спектрометрии, хотя пептид, сгенерировавший *Spectrum*, неизвестен, масса (пептид) может быть выведена из *Spectrum* и обозначается *ParentMass*(*Spectrum*). Для простоты будем считать, что для всех экспериментальных спектров *ParentMass*(*Spectrum*) равна наибольшей массе в *Spectrum*.

Алгоритм грубой силы секвенирования циклолептидов **BFCyclopeptideSequencing** генерирует все возможные пептиды, масса которых равна *ParentMass*(*Spectrum*), а затем проверяет, какие из этих пептидов имеют теоретические спектры, соответствующие *Spectrum*.

```
BFCyclopeptideSequencing (Spectrum)
    for каждого пептида с массой Mass(Peptide), равной ParentMass(Spectrum)
        if Spectrum = Cyclospectrum(Peptide)
            output Peptide
```

Не должно быть никаких сомнений в том, что **BFCyclopeptideSequencing** решит задачу секвенирования циклопептидов. Однако нас должно беспокоить время его работы: сколько существует пептидов с массой, равной *ParentMass*(*Spectrum*)?

Задача подсчета пептидов с заданной массой: *вычислить количество пептидов с заданной массой.*

Input: целое число m.

Output: количество линейных пептидов, имеющих целую массу m.

Упражнение. Решить задачу о подсчете пептидов с заданной массой. Таблица целочисленных масс воспроизводится ниже; напомним, что мы предполагаем, что пептиды образуются только из 18 масс аминокислот. То есть I/L считаются одинаковыми, и K/Q считаются одинаковыми, поэтому мы будем считать AIKD и ALQD просто одним пептидом, а не двумя.

G	A	S	P	V	T	C	I	L	N	D	K	Q	E	M	H	F	R	Y	W
57	71	87	97	99	101	103	113	113	114	115	128	128	129	131	137	147	156	163	186

Предложение. Это упражнение не является обязательным, поскольку у вас могут возникнуть трудности с решением данной задачи; если это так, пожалуйста, вернитесь к ней после того, как вы узнаете больше об алгоритмах динамического программирования в следующем разделе.

Оказывается, существуют *триллионы* пептидов с такой же целочисленной массой (1322), что и тироцидин B1 (рисунок ниже). Поэтому алгоритм **BFCyclopeptideSequencing** совершенно непрактичен, и мы даже не будем просить вас его разработать.

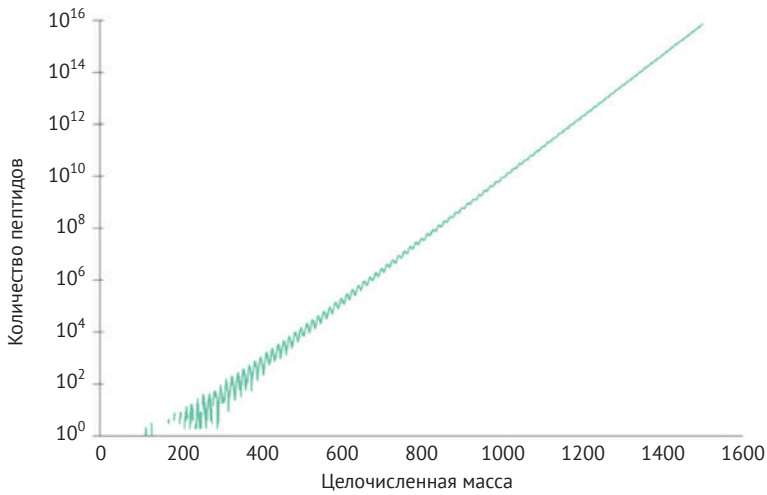


Рис. 4.8 Количество пептидов с заданной целочисленной массой растет экспоненциально

Упражнение. Этот рисунок предполагает, что для больших m количество пептидов с данной целочисленной массой m может быть аппроксимировано как $k \cdot C^m$, где k и C являются константами. Если вы решили задачу, заданную выше, используйте свое решение, чтобы найти C. (Дайте ответ в виде десятичной дроби; допустимая ошибка 0,002.)

Алгоритм ветвей и границ для секвенирования циклопептидов

Тот факт, что алгоритм из предыдущего раздела с треском провалился, не означает, что все методы грубой силы обречены. Можем ли мы разработать более быстрый алгоритм грубой силы, основанный на другой идее?

Вместо проверки всех циклических пептидов с заданной массой наш новый метод решения проблемы секвенирования циклопептидов будет «выращивать» линейные пептиды-кандидаты, чьи теоретические спектры «согласуются» с экспериментальным спектром.

ОСТАНОВИТЕСЬ и задумайтесь. Что должно означать, что линейный пептид согласуется с экспериментальным спектром? Могли бы вы классифицировать VKF как соответствующий спектру тироцидина B1, воспроизведенному ниже? Что у нас с VKY?

0	97	99	113	114	128	128	147	147	163	186	227
241	242	244	260	261	262	283	291	333	349	357	388
389	390	390	405	430	430	447	485	487	503	504	518
543	544	552	575	577	584	631	632	650	651	671	672
690	691	738	745	747	770	778	779	804	818	819	835
1031	1039	1060	1061	1062	1078	1080	1081	1095	1136	1159	1175
1175	1194	1194	1208	1209	1223	1225	1322				

Имея экспериментальный спектр *Spectrum*, сформируем набор линейных пептидов-кандидатов, первоначально состоящий из **пустого пептида**, который представляет собой просто пустую строку (обозначенную как «») с массой 0. На следующем шаге мы дополним *Peptides*, чтобы включить все линейные пептиды длины 1. Мы продолжаем этот процесс, создавая 18 новых пептидов длины $k + 1$ для каждой аминокислотной строки *Peptide*, присоединяя каждую возможную массу аминокислот к концу *Peptide*.

Чтобы количество пептидов-кандидатов не увеличивалось в геометрической прогрессии, каждый раз, когда мы расширяем *Peptides*, мы урезаем

его, оставляя только те линейные пептиды, которые соответствуют экспериментальному спектру. Затем проверяем, имеет ли какой-либо из этих новых линейных пептидов массу, равную *Mass*(*Spectrum*). Если это так, мы закольцовываем этот пептид и проверяем, обеспечивает ли он решение проблемы секвенирования циклопептидов.

В более общем смысле алгоритмы грубой силы, которые перебирают все решения-кандидаты, но отбрасывают большие подмножества безнадежных кандидатов, используя различные условия согласованности, известны как **алгоритмы ветвей и границ**. Каждый такой алгоритм состоит из **шага ветвления** для увеличения числа возможных решений, за которым следует **шаг ограничения** для удаления безнадежных кандидатов. В нашем алгоритме ветвей и границ для задачи секвенирования циклопептидов шаг ветвления будет расширять каждый пептид-кандидат длины k на 18 пептидов длины $k + 1$, а шаг ограничения удаляет из рассмотрения несогласованные пептиды.

Обратите внимание, что спектр *линейного* пептида не содержит столько масс, сколько спектр *циклического* пептида с той же аминокислотной последовательностью. Например, теоретический спектр циклического пептида NQEL содержит 14 масс (соответствует «», N, Q, E, L, LN, NQ, QE, EL, ELN, LNQ, NQE, QEL и NQEL). Однако приведенный ниже теоретический спектр линейного пептида NQEL не содержит масс, соответствующих LN, LNQ или ELN, поскольку эти субпептиды «обвивают» конец линейного пептида.

0	113	114	128	129	242	242	257	370	371	484
""	L	N	Q	E	NQ	EL	QE	QEL	NQE	NQEL

Упражнение. Сколько субпептидов имеет линейный пептид заданной длины n? (Включите пустой пептид и весь пептид целиком.)

Input: целое число n.

Output: количество субпептидов линейного пептида длины n.

Для экспериментального спектра *Spectrum* циклического пептида линейный пептид **согласуется** с *Spectrum*, если каждая масса в его теоретическом спектре содержится в *Spectrum*. Если масса появляется в теоретическом спектре линейного пептида более одного раза, то она должна появляться как минимум столько раз в *Spectrum*, чтобы линейный пептид соответствовал *Spectrum*. Например, линейный пептид может по-прежнему соответствовать теоретическому спектру NQEL, если спектр пептида дважды содержит 242. Но он не может быть согласованным с теоретическим спектром NQEL, если его спектр дважды содержит 113.

Ключом к нашему новому алгоритму является то, что каждый линейный субпептид циклического пептида *Peptide* согласуется с *Cyclospectrum*(*Peptide*).

Таким образом, чтобы решить задачу секвенирования циклопептидов для *Spectrum*, мы можем безопасно запретить все пептиды, несовместимые со *Spectrum*, из растущего набора *Peptide*, что усиливает шаг ограничения, описанный нами выше.

Например, линейный пептид VKF (со спектром {0, 99, 128, 147, 227, 275, 374}) будет запрещен, поскольку он не соответствует спектру тироцидина B1, воспроизведенному выше. Но линейный пептид VKY не будет запрещен, потому что каждая масса в его теоретическом спектре ({0, 99, 128, 163, 227, 291, 390}) присутствует в спектре тироцидина B1.

Как насчет шага ветвления? Имея текущую коллекцию линейных пептидов *Peptides*, определите *Expand(Peptides)* как новый набор, содержащий все возможные расширения пептидов в Peptides на одну массу аминокислоты. Теперь мы можем предоставить псевдокод для алгоритма ветвей и границ, называемого **CyclopeptideSequencing**.

```
CyclopeptideSequencing(Spectrum)
    CandidatePeptides ← набор, содержащий только пустой пептид FinalPeptides ←
    пустой набор строк
        while CandidatePeptides не пустой
            CandidatePeptides ← Expand(CandidatePeptides)
            for каждого пептида Peptide в CandidatePeptides
                if Mass(Peptide) = ParentMass(Spectrum)
                    if Cyclospectrum(Peptide) = Spectrum и Peptide не присутствует
                    в FinalPeptides
                        добавить Peptide к FinalPeptides
                    удалить Peptide из CandidatePeptides
                else if Peptide не согласуется со Spectrum
                    удалить Peptide из CandidatePeptides
    return FinalPeptides
```

Примечание После провала первого алгоритма грубой силы, который мы рассмотрели, вы, возможно, не решитесь реализовать алгоритм **CyclopeptideSequencing**, опасаясь, что его время выполнения будет непомерно высоким. Потенциальная задача циклопептидного секвенирования заключается в том, что на промежуточных стадиях могут быть получены неправильные *k*-меры (т. е. *k*-меры, которые не являются субпептидами правильного решения). Однако на практике это не вызывает беспокойства. См. раздел **ЗАРЯДНАЯ СТАНЦИЯ: Насколько быстро выполняется циклопептидное секвенирование?**.

Трудно представить наихудший сценарий, при котором **CyclopeptideSequencing** выполняется долго, но никто не может гарантировать, что этот алгоритм не будет генерировать огромное количество неверных *k*-меров на промежуточных этапах. **BFCyclopeptideSequencing** является экспоненци-

альным, и, хотя на практике **CyclopeptideSequencing** намного быстрее, полиномиальность этого алгоритма не доказана. Таким образом, с точки зрения вводного курса алгоритмов, посвященного теоретической информатике, практический алгоритм **CyclopeptideSequencing** столь же неэффективен, как и **BFCyclopeptideSequencing**, поскольку время выполнения ни одного из этих алгоритмов не может быть ограничено полиномом.

Масс-спектрометрия и гольф

От теоретических к реальным спектрам

Хотя **CyclopeptideSequencing** успешно реконструировал тироцидин B1, этот алгоритм работает только в случае идеального спектра, т. е. когда экспериментальный спектр пептида точно совпадает с его теоретическим спектром. Эта негибкость **CyclopeptideSequencing** представляет собой практический барьер, поскольку масс-спектрометры генерируют «зашумленные» спектры, далекие от идеальных, – они характеризуются наличием **ложных масс** и **отсутствующих масс**. Ложная масса присутствует в экспериментальном спектре, но отсутствует в теоретическом спектре; отсутствующая масса присутствует в теоретическом спектре, но ее нет в экспериментальном спектре.

Например, сравните следующие теоретические и (смоделированные) экспериментальные спектры циклического пептида NQEL. Массы, отсутствующие в экспериментальном спектре, показаны синим цветом, а ложные массы в экспериментальном спектре – зеленым.

Теоретический:	0		113	114	128	**129**	227	**242**	**242**	257		355	356	370	371	484
Экспериментальный:	0	**99**	113	114	128		227			257	**299**	355	356	370	371	484

Что особенно беспокоит в этом примере, так это то, что масса аминокислоты E (129) отсутствует, а масса аминокислоты V (99) неверна; в результате первый шаг **CyclopeptideSequencing** установил бы {V, L, N, Q} в качестве аминокислотного состава наших пептидов-кандидатов, что неверно. Фактически *любая* ложная или отсутствующая масса приведет к тому, что **CyclopeptideSequencing** отбросит правильный пептид, потому что его теоретический спектр отличается от экспериментального.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы переформулировали задачу секвенирования циклопептидов, чтобы обрабатывать экспериментальные спектры с ошибками?

Адаптация секвенирования циклопептидов для спектров с ошибками

Чтобы обобщить задачу секвенирования циклопептидов для работы с зашумленными спектрами, нам нужно ослабить требование, согласно которому теоретический спектр пептида-кандидата должен *точно* совпадать с экспериментальным спектром, и вместо этого включить **функцию оценки**, которая будет выбирать пептид, чей теоретический спектр соответствует заданному экспериментальному спектру *наиболее близко*. Имея циклический пептид *Peptide* и спектр *Spectrum*, мы определяем *Score*(*Peptide, Spectrum*) как число масс, общих между *Cyclospectrum*(*Peptide*) и *Spectrum*. Вспомним наш предыдущий пример (см. таблицу выше).

Если *Spectrum* является экспериментальным спектром в нижней строке этой таблицы, то *Score*(NQEL, *Spectrum*) = 11 (количество столбцов, общих для теоретического и экспериментального спектров, показано черным цветом).

Функция оценки должна учитывать **кратность** общих масс, т. е. сколько раз они встречаются в каждом спектре. Например, предположим, что *Spectrum* – это теоретический спектр NQEL; для этого спектра масса 242 имеет кратность 2. Если 242 имеет кратность 1 в экспериментальном спектре пептида, то 242 вносит 1 в *Score*(*Peptide, Spectrum*). Если 242 имеет кратность 2 или более в экспериментальном спектре *Peptide*, то 242 вносит 2 в *Score*(*Peptide, Spectrum*).

Задача оценки циклопептидов: *подсчитайте оценку циклического пептида по спектру.*

Input: аминокислотная строка *Peptide* и набор целых чисел *Spectrum*.

Output: счет пептида по сравнению со спектром, *Score*(*Peptide, Spectrum*).

Теперь мы можем переопределить задачу секвенирования циклопептидов для зашумленных спектров.

Задача секвенирования циклопептидов (для спектров с ошибками): *найти циклический пептид с максимальным счетом по сравнению с экспериментальным спектром.*

Input: набор целых чисел *Spectrum*.

Output: циклический пептид *Peptide*, максимизирующий *Score*(*Peptide, Spectrum*), по сравнению со всеми пептидами *Peptide* с массой, равной *ParentMass*(*Spectrum*).

Наша цель – адаптировать алгоритм **CyclopeptideSequencing** для поиска пептида с максимальной кратностью. Помните, что этот алгоритм имел строгий ограничивающий шаг, на котором отбрасывались все линейные пептиды-кандидаты, имеющие несогласованные спектры. Например, мы видели, что линейный пептид VKF не соответствует теоретическому спектру циклического пептида тироцидина B1. Однако, пожалуй, не следует запрещать VKF в случае экспериментальных спектров, так как они могут иметь недостающие массы. Таким образом, нам, возможно, следует пересмотреть этот шаг, чтобы включить больше линейных пептидов-кандидатов, при этом гарантируя, что количество рассматриваемых нами пептидов не выйдет из-под контроля.

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем ограничить рост набора линейных пептидов-кандидатов в случае экспериментальных спектров?

Чтобы ограничить количество рассматриваемых линейных пептидов-кандидатов, мы заменим *Peptides* таблицей лидеров *Leaderboard*, которая содержит *N* пептидов-кандидатов с наивысшей кратностью для дальнейшего пополнения. На каждом шаге мы будем расширять охват пептидов-кандидатов, найденных в *Leaderboard*, а затем исключать те пептиды, чьи недавно рассчитанные кратности недостаточно высоки, чтобы сохранить их в *Leaderboard*. Эта идея похожа на понятие «кат» в турнире по гольфу («отсечение»); после нее только *N* лучших игроков в гольф могут играть в следующем раунде, поскольку это игроки, у которых наибольшие разумные шансы на победу.

Чтобы быть справедливым, кат (это сокращение набора) должен включать всех, кто равен по очкам с участником, занявшим *N*-е место. Таким образом, набор *Leaderboard* должен быть сокращен до «*N* линейных пептидов с наивысшей кратностью, включая связи», которые могут включать более *N* пептидов. Имея набор *Leaderboard*, спектр *Spectrum* и целое число *N*, определите *Trim*(*Leaderboard, Spectrum, N*) как набор *N* линейных пептидов с наибольшей кратностью в *Leaderboard* (включая связи) по отношению к *Spectrum*.

ОСТАНОВИТЕСЬ и задумайтесь. Наша оценочная функция пептидов в настоящее время предполагает, что пептиды являются кольцевыми, но технически пептиды должны считаться как линейные пептиды до последнего шага. Как бы вы оценили линейный пептид в сравнении со спектром?

Обратите внимание, что *Score*(*Peptide, Spectrum*) оценивает *Peptide* по сравнению со *Spectrum* только в том случае, если *Peptide* является циклическим. Однако, чтобы обобщить эту функцию оценки, когда *Peptide* является линейным, мы просто исключаем те субпептиды *Peptide*, которые охватывают конец строки, в результате чего получается функция *LinearScore*(*Peptide, Spectrum*). Напри-

мер, если *Spectrum* является экспериментальным спектром NQEL, то вы можете убедиться, что *LinearScore*(NQEL, *Spectrum*) = 8.

Теперь мы представляем алгоритм **LeaderboardCyclopeptideSequencing**.

```
LeaderboardCyclopeptideSequencing(Spectrum, N)
    Leaderboard ← набор, содержащий только пустые пептиды
    LeaderPeptide ← пустой пептид
    while Leaderboard не пустой
        Leaderboard ← Expand(Leaderboard)
        for каждый Peptide в Leaderboard
            if Mass(Peptide) = ParentMass(Spectrum)
                if Score(Peptide, Spectrum) > Score(LeaderPeptide, Spectrum)
                    LeaderPeptide ← Peptide
            else if Mass(Peptide) > ParentMass(Spectrum)
                удалить Peptide из Leaderboard
        Leaderboard ← Trim(Leaderboard, Spectrum, N)
    output LeaderPeptide
```

Сложный аспект реализации **LeaderboardCyclopeptideSequencing** заключается в том, чтобы убедиться, что функция *Trim* реализована правильно. Ознакомьтесь с разделом **ЗАРЯДНАЯ СТАНЦИЯ: Сокращение таблицы лидеров пептидов**, чтобы лучше понять тему сокращения таблицы лидеров.

Мы отмечаем, что, поскольку линейный пептид, дающий начало циклическому пептиду с наивысшим счетом, может быть исключен из таблицы лидеров на раннем этапе, **LeaderboardCyclopeptideSequencing** является эвристикой, которая не гарантирует правильного решения проблемы секвенирования циклопептидов. Когда мы разрабатываем эвристику, мы должны спросить себя: насколько она точна? Рассмотрим смоделированный спектр $Spectrum_{10}$ тироцидина B1 (показано ниже), который имеет приблизительно 10 % **отсутствующих/ложных** масс. Обратите внимание, что синих масс на самом деле нет в спектре, но мы показываем их так, чтобы было понятно, каких масс не хватает.

0	97	99	**113**	114	**128**	128	147	147	163	186	227
241	242	244	260	261	262	283	291	333	340	357	385
388	389	390	390	405	430	430	447	485	487	503	504
518	543	544	552	575	577	584	**631**	632	650	651	671
672	690	691	738	745	747	770	778	779	804	818	819
820	835	837	875	892	**892**	917	932	932	933	934	965
982	989	**1030**	**1031**	1039	1060	1061	1062	1078	1080	1081	1095
1136	1159	1175	1175	1194	1194	1208	1209	1223	1225	1322	

Применение **LeaderboardCyclopeptideSequencing** к этому спектру (с $N = 1000$) дает правильный циклический пептид VKLFPWFNQY, который имеет счет 86.

До сих пор алгоритм **LeaderboardCyclopeptideSequencing** работал хорошо, но по мере увеличения количества ошибок возрастает и вероятность того, что этот алгоритм даст неправильный пептид на выходе. Давайте посмотрим, как этот алгоритм работает с более зашумленным смоделированным спектром; ниже мы показываем $Spectrum_{25}$ для тироцидина B1, который имеет 25 % **отсутствующих/ложных** масс.

0	97	99	113	114	**115**	128	128	147	147	163	186
227	241	242	**244**	244	**256**	260	261	262	283	291	**309**
330	333	340	**347**	**357**	**385**	388	389	390	390	405	**430**
430	435	447	485	487	503	504	518	**543**	544	552	575
577	584	**599**	**608**	631	632	650	651	**653**	**671**	672	690
691	**717**	738	745	**747**	770	**778**	779	804	818	819	**827**
835	837	875	892	892	917	932	932	933	934	965	982
989	**1031**	1039	1060	**1061**	1062	1078	1080	1081	1095	1136	1159
1175	1175	1194	1194	1208	1209	1223	**1225**	1322			

Упражнение. Запустите **LeaderboardCyclopeptideSequencing** на $Spectrum_{25}$ с $N = 1000$. Вы должны найти 38 линейных пептидов с максимальной кратностью 83 (соответствует 15 различным циклическим пептидам). Что это за пептиды? (Выведите ваши пептиды в целочисленном формате, где каждый пептид разделен одним пробелом, например 113-147-71-129 71-147-129-113.)

Примечание. Можно найти больше пептидов с оценкой 83, если использовать большее значение N; например, используйте $N = 1000$.

При запуске на $Spectrum_{25}$ **LeaderboardCyclopeptideSequencing** (с $N = 1000$) идентифицирует VKLFPADFNQY (кратность: 83) как циклический пептид с наивысшим счетом вместо правильного пептида VKLFPWFNQY (кратность: 82). Эти два пептида подобны благодаря тому факту, что общая масса **A** (71) и **D** (115) равны массе W (186).

ОСТАНОВИТЕСЬ и задумайтесь. Как мы могли исключить неправильный пептид VKLFP**AD**FNQY из рассмотрения для $Spectrum_{25}$?

Обратите внимание, что, хотя правильные и неправильные пептиды похожи, их аминокислотный состав различается. Если бы мы могли вычислить аминокислотный состав тироцидина B1 только по его спектру и провести секвени-

рование **LeaderboardCyclopeptideSequencing** на этом меньшем алфавите (а не на алфавите всех аминокислот), то мы могли бы исключить из рассмотрения неверный пептид VKLFP**AD**FNQY.

От 20 до более чем 100 аминокислот

До сих пор мы предполагали, что только 20 аминокислот образуют строительные блоки белков; эти строительные блоки называются **протеиногенными аминокислотами**. На самом деле существуют две дополнительные протеиногенные аминокислоты – **селеноцистеин** и **пирролизин**, которые внедряются в белки с помощью специальных биосинтетических механизмов (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Селеноцистеин и пирролизин**). Однако, помимо 22 протеиногенных аминокислот, NRP содержат **непротеиногенные аминокислоты**, которые увеличивают количество возможных строительных блоков для пептидов-антибиотиков с 20 до числа, превышающего 100.

Расширение алфавита аминокислот создает проблемы для нашего нынешнего метода секвенирования циклопептидов. Действительно, правильный пептид теперь должен «конкурировать» со многими другими неправильными за место в таблице лидеров, увеличивая шанс того, что правильный пептид будет отсечен в процессе выполнения алгоритма.

Например, хотя тироцидин B1 содержит только протеиногенные аминокислоты, его близкий родственник тироцидин B (Val-**Orn**-Leu-Phe-Pro-Trp-Phe-Asn-Gln-Tyr) содержит непротеиногенную аминокислоту под названием **орнитин** (**Orn**). Поскольку существует так много непротеиногенных аминокислот, биоинформатики часто предполагают, что любое целое число от 57 до 200 может представлять массу аминокислоты; самая «легкая» аминокислота, глицин (Gly), имеет массу 57 Да, а большинство аминокислот имеет массу менее 200 Да.

Упражнение. Применение **LeaderboardCyclopeptideSequencing** к расширенному аминокислотному алфавиту (т. е. к каждому целому числу от 57 до 200 включительно) к $Spectrum_{10}$ с N = 1000 выдает 34 различных линейных пептида с максимальной оценкой. Что это за пептиды? (Дайте ответ в целочисленном формате, разделенном пробелом, например 113-147-71-129 199-200-61.)

Примечание. Это упражнение сложное, поскольку его выполнение может занять много времени, если ваше решение неэффективно.

Когда мы применяем **LeaderboardCyclopeptideSequencing** для расширенного алфавита к $Spectrum_{10}$, одним из самых результативных пептидов являет-

ся VKLFPWFNQ**XZ**, где X имеет массу 98, а Z – 65. По-видимому, нестандартные аминокислоты успешно конкурировали со стандартными аминокислотами за ограниченное количество позиций в таблице лидеров, в результате чего VKLFPWFNQ**XZ** победил правильный пептид VKLFPWFNQY. Поскольку **LeaderboardCyclopeptideSequencing** не может идентифицировать правильный пептид даже с 10 % ложных и отсутствующих масс, наша заявленная цель из предыдущего раздела теперь еще более важна. Мы должны определить аминокислотный состав пептида по его спектру, чтобы можно было запустить **LeaderboardCyclopeptideSequencing** для этого меньшего алфавита аминокислот.

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем определить, какие аминокислоты присутствуют в неизвестном пептиде, используя только экспериментальный спектр?

Спектральная свертка спасает положение

Один из способов определить аминокислотный состав пептида по его экспериментальному спектру – взять наименьшие массы, присутствующие в спектре (между 57 и 200 Да). Однако, если отсутствует только одна аминокислотная масса, этот метод не сможет восстановить аминокислотный состав пептида.

Давайте применим другой метод. Скажем, наш экспериментальный спектр содержит массу субпептидов NQ**E** и NQ. Разница этих масс дает массу **E**, даже если ее не было в экспериментальном спектре! Если лежащим в основе пептидом является NQEL, то мы также можем найти массу **E**, вычитая массы QE и Q или NQEL и LNQ.

Следуя этому примеру, мы определяем **свертку** (Свертка, или конволюция, – математическая операция, которая при применении к двум функциям возвращает третью функцию, соответствующую взаимнокорреляционной функции. – *Прим. ред.*) спектра, взяв все положительные разности масс в спектре. В таблицах далее показаны свертки теоретических и смоделированных спектров NQEL, с которыми мы сталкивались ранее (спектры воспроизведены ниже).

Теоретический:	0		113	114	128	**129**	227	**242**	**242**	257		355	356	370	371	484
Экспериментальный:	0	**99**	113	114	128		227			257	**299**	355	356	370	371	484

Как и предполагалось, некоторые значения в этих таблицах появляются чаще, чем другие. Например, **113** (масса **L**) имеет кратность 8 в таблице выше; мы говорим, что число **113** имеет **кратность** 8. Шесть из восьми вхождений числа **113** в приведенной выше таблице соответствуют парам субпептидов, отличающихся **L**: **L** и «»; **L**N и N; E**L** и E; **L**NQ и NQ; QE**L** и QE; NQE**L** и NQE.

	""	L	N	Q	E	LN	NQ	EL	QE	LNQ	ELN	QEL	NQE
	0	113	114	128	129	227	242	242	257	355	356	370	371
0													
113	**113**												
114	**114**	1											
128	**128**	15	14										
129	**129**	16	15	1									
227	227	**114**	**113**	99	98								
242	242	**129**	**128**	**114**	**113**	15							
242	242	**129**	**128**	**114**	**113**	15							
257	257	144	143	**129**	**128**	30	15	15					
355	355	242	241	227	226	**128**	**113**	**113**	98				
356	356	243	242	228	227	**129**	**114**	**114**	99	1			
370	370	257	256	242	241	143	**128**	**128**	**113**	15	14		
371	371	258	257	243	242	144	**129**	**129**	**114**	16	15	1	
484	484	371	370	356	355	257	242	242	227	**129**	**128**	**114**	**113**

Рис. 4.9 Спектральная свертка для теоретического спектра NQEL. Наиболее часто встречающиеся элементы в свертке между 57 и 200 (кратность в скобках): 113 (8), 114 (8), 128 (8), 129 (8)

	""	false	L	N	Q	LN	QE	false	LNQ	ELN	QEL	NQE
	0	99	113	114	128	227	257	299	355	356	370	371
0												
99	**99**											
113	**113**	14										
114	**114**	15	1									
128	**128**	29	15	14								
227	227	**128**	**114**	**113**	**99**							
257	257	158	144	143	**129**	30						
299	299	200	186	185	171	72	42					
355	355	256	242	241	227	**128**	98	56				
356	356	257	243	242	228	**129**	**99**	57	1			
370	370	271	257	256	242	143	**113**	71	15	14		
371	371	272	258	257	243	144	**114**	72	16	15	1	
484	484	385	371	370	356	257	227	185	**129**	**128**	**114**	**113**

Рис. 4.10 Спектральная свертка для смоделированного спектра NQEL. Наиболее часто в свертке между 57 и 200 встречаются элементы (кратности в скобках): 113 (4), 114 (4), 128 (4), 99 (3), 129 (3)

Интересно, что **129** (масса **E**) появляется три раза в приведенной выше свертке смоделированного спектра, хотя **129** отсутствовало в самом спектре.

Теперь мы должны чувствовать себя уверенно, используя наиболее часто встречающиеся целые числа в свертке в качестве предположения об аминокислотном составе неизвестного пептида. В нашем смоделированном спектре для NQEL наиболее частыми элементами свертки в диапазоне от 57 до 200 являются (кратности в скобках):

113 (4), **114** (4), **128** (4), **99** (3), **129** (3).

Обратите внимание, что эти наиболее часто встречающиеся элементы захватывают все четыре аминокислоты в NQEL.

Задача спектральной свертки: *вычислите свертку спектра.*

Input: набор целых чисел *Spectrum*.

Output: список элементов свертки *Spectrum*. Если элемент имеет кратность *k*, он должен встречаться ровно *k* раз; вы можете выдать элементы в любом порядке.

Напомним, что алгоритм **LeaderboardCyclopeptideSequencing** не смог реконструировать тироцидин B1 из $Spectrum_{10}$ при использовании расширенного алфавита аминокислот. Десять наиболее часто встречающихся элементов его спектральной свертки в диапазоне от 57 до 200 (с кратностями в скобках):

147 (35)	128 (31)	97 (28)	113 (28)	114 (26)
186 (23)	57 (21)	163 (21)	99 (18)	145 (18)

Каждая масса в этом списке, кроме 57 и 145, захватывает аминокислоту в тироцидине B1! На следующем шаге показана свертка Spectrum10.

Теперь у нас есть схема нового алгоритма секвенирования циклопептидов. Имея экспериментальный спектр, мы сначала вычисляем свертку экспериментального спектра. Затем выбираем *M* наиболее часто встречающихся элементов между 57 и 200 в свертке, чтобы сформировать расширенный алфавит возможных масс аминокислот. Чтобы быть справедливым, мы должны включить верхние *M* элементов свертки «со связями». Наконец, мы запускаем алгоритм **LeaderboardCyclopeptideSequencing**, в котором массы аминокислот ограничены этим алфавитом. Мы называем этот алгоритм **ConvolutionCyclopeptideSequencing**.

Упражнение. Запустите **ConvolutionCyclopeptideSequencing** на $Spectrum_{25}$ (воспроизведенном ниже) с $N = 1000$ и $M = 20$. Определите 86 пептидов с наивысшей оценкой, используя функцию циклической оценки. (Верните пептиды в целочисленном формате, разделенные одним пробелом, например 123-57-200-143 199-143-121-60.)

ConvolutionCyclopeptideSequencing (с $N = 1000$ и $M = 20$) теперь правильно реконструирует тироцидин B1 из $Spectrum_{10}$. Настоящая проверка этого алгоритма заключается в том, будет ли он работать на более зашумленном спектре. Напомним, что наш предыдущий алгоритм не смог определить правильный пептид для $Spectrum_{25}$. Напротив, **ConvolutionCyclopeptideSequencing** (с $N = 1000$ и $M = 20$) теперь правильно идентифицирует тироцидин B1 из этого спектра!

Рис.4.11 Спектральная свертка экспериментального спектра $Spectrum_{10}$ для тироцидина B1. Для каждого элемента спектральной свертки (показанного в виде числа в рамке) его координата *y* представляет количество раз, которое элемент появляется в спектральной свертке. Зеленые прямоугольники представляют массы аминокислот из тироцидина B1

0	97	99	113	114	**115**	128	128	147	147	163	186
227	241	242	**244**	244	**256**	260	261	262	283	291	**309**
330	333	340	**347**	**357**	**385**	388	389	390	390	405	**430**
430	**435**	447	485	487	503	504	518	**543**	544	552	575
577	584	**599**	**608**	631	632	650	651	**653**	**671**	672	690
691	**717**	738	745	**747**	770	**778**	779	804	818	819	**827**
835	837	875	892	892	917	932	932	933	934	965	982
989	**1031**	1039	1060	**1061**	1062	1078	1080	1081	1095	1136	1159
1175	1175	1194	1194	1208	1209	1223	**1225**	1322			

Рис. 4.12 $Spectrum_{25}$

Эпилог. От смоделированных спектров – к реальным

В этой главе мы оградили вас от ужасных реалий масс-спектрометрии, предоставив смоделированные спектры, которые относительно легко секвенировать (даже с ложными или отсутствующими массами). Мы совершили грех упрощения, назвав масс-спектрометр «весами» и предположив, что эта сложная машина просто взвешивает крошечные пептидные фрагменты по одному. На самом деле масс-спектрометр сначала превращает субпептиды в ионы (т. е. заряженные частицы). Ионизация частиц помогает масс-спектрометру сортировать ионы с помощью электрического и магнитного полей; ионы разделяются не по массе, а по **соотношению масса/заряд**. Если ион NQY (целочисленная масса: 114 + 128 + 163 = 405) имеет заряд +1, то он содержит один дополнительный протон, в результате чего общая целочисленная масса равна 406, а отношение массы к заряду равно 406/1 = 406. Если быть более точным, моноизотопная масса NQY составляет примерно 114,043 + 128,058 + 163,063 = 405,164, а масса протона составляет 1,007 Да, что делает отношение массы к заряду более близким к (405,164 + 1,007) / 1 = 406,171.

Масс-спектрометр выводит набор **пиков**, которые показаны ниже для реального спектра тироцидина B1. Координата x каждого пика представляет отношение массы иона к заряду, а его высота представляет **интенсивность** (т. е. относительную численность) ионов, имеющих соответственное отношение массы к заряду. Например, в экспериментальном спектре тироцидина B1, показанном ниже, вы найдете небольшой (почти невидимый) пик с отношением массы к заряду 406,30, который соответствует фрагментному иону NQY, имеющему отношение массы к заряду 406,171, с ошибкой примерно 0,13 Да.

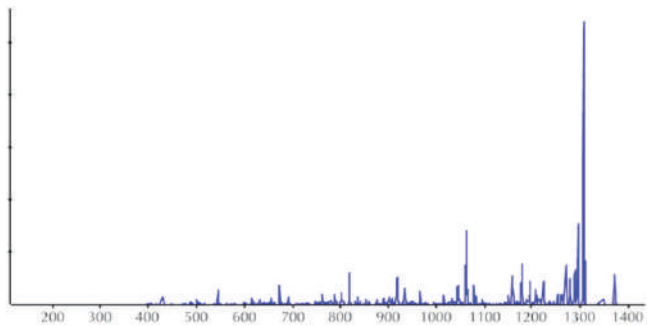


Рис. 4.13 Масс-спектр тироцидина B1

Как вы понимаете, мы должны преодолеть несколько практических барьеров, чтобы анализировать реальные спектры. Во-первых, заряд каждого пика неизвестен, что часто вынуждает исследователей пробовать все возможные заряды от 1 до некоторого параметра *maxCharge*, где конкретный выбор *maxCharge* зависит от используемой технологии фрагментации. Эта процедура

генерирует массы *maxCharge* для каждого пика, так что чем больше значение *maxCharge*, тем больше ложных масс в спектре.

Во-вторых, реальный спектр, показанный выше, имеет около 1000 пиков, большинство из которых являются **ложными пиками**, что означает, что их отношение масса/заряд не соответствует отношению массы/заряда какого-либо субпептида (при любом значении заряда). К счастью, ложные пики обычно имеют низкую интенсивность, что требует этапа предварительной обработки, который удаляет пики низкой интенсивности перед применением алгоритма. Ниже приведен список 95 соотношений масса/заряд для пиков, которые «выжили» на этом этапе предварительной обработки для спектра тироцидина B1. Тем не менее их интенсивность может различаться на 2-3 порядка; например, интенсивность пика с отношением масса/заряд 372,2 в 300 раз меньше, чем интенсивность пика с отношением масса/заряд 1306,5.

372.2	397.2	402.0	**406.3**	415.1	**431.2**	**448.3**	449.3	452.2
471.3	**486.3**	**488.2**	500.5	**505.3**	516.1	536.1	**544.2**	**545.3**
562.5	571.3	599.2	614.4	615.4	616.4	618.2	**632.0**	655.5
656.3	**672.5**	**673.3**	677.3	**691.4**	**692.4**	712.1	722.3	**746.5**
760.4	761.6	762.5	**771.6**	788.4	802.3	803.3	818.5	**819.4**
831.4	**836.3**	853.3	875.5	**876.5**	901.5	915.9	916.5	917.8
918.4	**933.4**	**934.7**	**935.5**	949.4	**966.2**	995.4	1015.6	1027.5
1029.5	1031.5	1044.5	1046.5	**1061.5**	**1063.4**	**1079.2**	1083.7	
1088.4	1093.5	**1096.5**	1098.4	1158.5	1159.5	**1176.6**	1177.7	
1178.6	1192.7	**1195.4**	1207.5	**1210.4**	**1224.6**	1252.5	1270.5	
1271.5	1278.6	1279.6	1295.6	1305.6	1306.5	1307.5	1309.6	

Рис. 4.14 95 соотношений масса/заряд, имеющих наибольшую интенсивность. Значения, выделенные жирным шрифтом, соответствуют массам субпептидов тироцидина B1, если мы допускаем расхождение масс до 0,3 Да (рис. 4.13)

Только 31 из этих 95 соотношений масса/заряд (выделены жирным шрифтом на рисунках выше и ниже) могут быть сопоставлены с субпептидами тироцидина B1 (с *maxCharge* = 1 и максимально допустимым расхождением масс 0,3 Да):

Масса	Субпептид	Масса	Субпептид	Масса	Субпептид
406.2	NQY	431.2	FPW	448.2	WFN
486.2	KLFP	488.2	VKLF	505.2	NQYV
544.2	LFPW	545.2	PWFN	632.3	QYVKL
672.3	KLFPW	673.3	PWFNQ	691.3	LFPWF
692.3	FPWFN	746.3	NQYVKL	771.3	VKLFPW
819.4	KLFPWF	836.4	PWFNQY	876.4	QYVKLFP
918.4	VKLFPWF	933.4	LFPWFNQ	934.4	YVKLFPW
935.4	PWFNQYV	966.4	WFNQYVK	1061.5	KLFPWFNQ
1063.5	PWFNQYVK	1079.5	WFNQYVKL	1096.5	LFPWFNQY
1176.5	NQYVKLFPW	1195.6	LFPWFNQYV	1210.6	FPWFNQYVK
1224.6	KLFPWFNQY				

Рис. 4.15 31 соотношений масса/заряд, соответствующих субпептидам тироцидина B1

Теперь вы можете видеть, что секвенирование тироцидина B1 из реального спектра, для которого две трети всех масс являются ложными, представляет собой гораздо более сложную задачу, чем секвенирование этого пептида из смоделированного $Spectrum_{25}$. Попробуйте развить методы, которые мы изучили в этой главе, для анализа реального спектра.

Заключительная задача. Тироцидин B1 – лишь один из многих известных NRP, продуцируемых *Bacillus brevis*. Один вид бактерий может продуцировать десятки различных антибиотиков, и даже после 70 лет исследований, вероятно, существуют неоткрытые антибиотики, продуцируемые *Bacillus brevis*. Попробуйте секвенировать тироцидин, соответствующий реальному экспериментальному спектру ниже. Поскольку технология фрагментации, используемая для генерации спектра, имеет тенденцию производить ионы с зарядом +1, вы можете с уверенностью предположить, что все заряды равны +1. Выдайте пептид как набор целочисленных масс, разделенных пробелами.

371.5	375.4	390.4	392.2	409.0	420.2	427.2	443.3	446.4	461.3
471.4	477.4	491.3	505.3	506.4	519.2	536.1	546.5	553.3	562.3
588.2	600.3	616.2	617.4	618.3	633.4	634.4	636.2	651.5	652.4
702.5	703.4	712.5	718.3	721.0	730.3	749.4	762.6	763.4	764.4
779.6	780.4	781.4	782.4	797.3	862.4	876.4	877.4	878.6	879.4
893.4	894.4	895.4	896.5	927.4	944.4	975.5	976.5	977.4	979.4
1005.5	1007.5	1022.5	1023.7	1024.5	1039.5	1040.3	1042.5	1043.4	1057.5
1119.6	1120.6	1137.6	1138.6	1139.5	1156.5	1157.6	1168.6	1171.6	1185.4
1220.6	1222.5	1223.6	1239.6	1240.6	1250.5	1256.5	1266.5	1267.5	1268.6

Рис. 4.16 Экспериментальный спектр

Зарядные станции

Создание теоретического спектра пептида

Имея аминокислотную строку *Peptide*, мы начнем с предположения, что она представляет собой *линейный* пептид. Наш метод построения его теоретического спектра основан на предположении, что масса любого субпептида равна разности масс двух префиксов пептида. Мы можем вычислить массив *PrefixMass*, в котором хранятся массы каждого префикса *Peptide* в порядке возрастания, например для *Peptide* = NQEL, *PrefixMass* = (0, 114, 242, 371, 484). Затем масса субпептида *Peptide*, начиная с позиции $i + 1$ и заканчивая в позиции j, может быть вычислена как $PrefixMass(j) - PrefixMass(i)$. Например, когда *Peptide* = NQEL,

$$Mass(\text{QE}) = PrefixMass(3) - PrefixMass(1) = 371 - 114 = 257.$$

Псевдокод, показанный ниже, реализует эту идею. Он также представляет алфавит из 20 аминокислот и их целочисленные массы в виде пары массивов из 20 элементов *AminoAcid* и *AminoAcidMass*, соответствующих верхней и нижней строкам следующей таблицы целочисленных масс соответственно.

G	A	S	P	V	T	C	I	L	N	D	K	Q	E	M	H	F	R	Y	W
57	71	87	97	99	101	103	113	113	114	115	128	128	129	131	137	147	156	163	186

Рис. 4.17 Пары массивов из 20 элементов *AminoAcid* и *AminoAcidMass*

В следующем псевдокоде предполагается, что у нас есть строка или список символов *Alphabet*, содержащий 20 символов аминокислотного алфавита, и словарь *AminoAcidMass*, ключами которого являются символы *Alphabet*, а значениями являются целые массы каждого символа.

```
LinearSpectrum(Peptide, Alphabet, AminoAcidMass)
  PrefixMass(0) ← 0
    for i ← от 1 до |Peptide|
      for каждого символа s в Alphabet
        if s = i-я аминокислота в Peptide
          PrefixMass(i) ← PrefixMass(i − 1) + AminoAcidMass[s]
    LinearSpectrum ← список, состоящий из одного целого числа 0
    for i ← от 0 до |Peptide| − 1
      for j ← i + 1 до |Peptide|
        добавить PrefixMass(j) − PrefixMass(i) в LinearSpectrum
    return отсортированный список LinearSpectrum
```

Если вместо этого *Peptide* представляет собой *циклический* пептид, то массы в его теоретическом спектре можно разделить на массы, найденные **LinearSpectrum**, и массы, соответствующие субпептидам, обернутым вокруг конца *Peptide*. Кроме того, каждый такой субпептид имеет массу, равную разнице между *Mass*(*Peptide*) и массой субпептида, идентифицированной **LinearSpectrum**. Например, когда *Peptide* = NQEL,

$$Mass(\mathbf{LN}) = Mass(\mathbf{NQEL}) - Mass(\mathbf{QE}) = \mathbf{484} - \mathbf{257} = \mathbf{227}.$$

Таким образом, мы можем сгенерировать циклический спектр, внеся лишь небольшую модификацию в псевдокод **LinearSpectrum**.

```
CyclicSpectrum(Peptide, Alphabet, AminoAcidMass)
  PrefixMass(0) ← 0
  for i ← 1 to |Peptide|
    for каждого символа s в Alphabet
      if s = i-я аминокислота в Peptide
        PrefixMass(i) ← PrefixMass(i − 1) + AminoAcidMass[s]
```

```
peptideMass ← PrefixMass(|Peptide|)
CyclicSpectrum ← список, состоящий из единственного целого числа 0
for i ← 0 до |Peptide| − 1
    for j ← i + 1 до |Peptide|
        добавить PrefixMass(j) − PrefixMass(i) к CyclicSpectrum
        if i > 0 и j < |Peptide|
            добавить peptideMass − (PrefixMass(j) − PrefixMass(i)) к CyclicSpectrum
return отсортированный список CyclicSpectrum
```

Насколько быстро выполняется алгоритм CyclopeptideSequencing?

Давайте запустим **CyclopeptideSequencing** на следующем спектре:

0 97 97 99 101 103 196 198 198 200 202
295 297 299 299 301 394 396 398 400 400 497

CyclopeptideSequencing сначала расширяет набор *Peptides* до набора всех 1-меров, совместимых со *Spectrum*:

97	**99**	**101**	**103**
P	V	T	C

Затем алгоритм добавляет каждую из 18 аминокислотных масс к каждому из 1-меров, указанных выше. Результирующий набор пептидов, содержащий 4·18 = 72 пептида длины 2, затем обрезается, чтобы оставить только десять пептидов, соответствующих спектру:

97-99	97-101	97-103	99-97	99-101
PV	PT	PC	VP	VT
99-103	101-97	101-99	103-97	103-99
VC	TP	TV	CP	CV

После расширения и обрезки на следующей итерации набор *Peptides* содержит 15 последовательных 3-меров:

97-99-103	**97-99-101**	97-101-97	**97-101-99**	97-103-99
PVC	**PVT**	PTP	**PTV**	PCV
99-97-103	99-97-101	**99-101-97**	99-103-97	101-97-99
VPC	VPT	**VTP**	VCP	TPV
101-97-103	**101-99-97**	103-97-101	**103-97-99**	103-99-97
TPC	**TVP**	CPT	**CPV**	CVP

Рис. 4.18 Набор пептидов после итерации

С еще одной итерацией набор *Peptides* содержит десять последовательных 4-меров. Обратите внимание, что шесть 3-меров, выделенных красным выше, не смогли расшириться в какие-либо 4-меры ниже, и поэтому теперь мы знаем, что **CyclopeptideSequencing** может генерировать некоторые неправильные k-меры на промежуточных итерациях.

97-99-103-97 PVCP	97-101-97-99 PTPV	97-101-97-103 PTPC	97-103-99-97 PCVP
99-97-101-97 VPTP	99-103-97-101 VCPT	101-97-99-103 TPVC	101-97-103-99 TPCV
	103-97-101-97 CPTP	103-99-97-101 CVPT	

Рис. 4.19 Набор пептидов после следующей итерации

На последней итерации мы генерируем 10 согласованных 5-меров:

97-99-103-97-101 PVCPT	97-101-97-99-103 PTPVC	97-101-97-103-99 PTPCV
97-103-99-97-101 PCVPT	99-97-101-97-103 VPTPC	99-103-97-101-97 VCPTP
101-97-99-103-97 TPVCP	101-97-103-99-97 TPCVP	103-97-101-97-99 CPTPV
	109-99-97-101-97 CVPTP	

Рис. 4.20 Набор 5-меров после последней итерации

Все эти линейные пептиды соответствуют одному и тому же циклическому пептиду PVCPT, что решает задачу секвенирования циклопептидов.

Сокращение списка пептидов Leaderboard

> **Примечание** В этой зарядной станции используются некоторые обозначения из раздела **ЗАРЯДНАЯ СТАНЦИЯ: Создание теоретического спектра пептида**.

Чтобы реализовать функцию *Trim* в **LeaderboardCyclopeptideSequencing**, мы сначала сгенерируем теоретические спектры всех линейных пептидов из *Leaderboard*. Затем вычислим значения каждого теоретического спектра по сравнению с экспериментальным спектром *Spectrum*. Это требует реализации *LinearScore*(*Peptide, Spectrum*).

Ниже показана *Leaderboard* из десяти линейных пептидов, представленная в виде массива *Leaderboard* (вверху), а также десятиэлементного массива *LinearScores* (внизу), содержащего их значения.

LEADERBOARD	PVT	PTP	PTV	PCP	VPC	VTP	VCP	TPV	TPC	TVP
LINEARSCORES	6	2	4	6	5	2	5	4	4	3

Рис. 4.21 *Leaderboard* и *LinearScores*

Алгоритм **Trim**, показанный ниже, сортирует все пептиды в *Leaderboard* в соответствии с их значениями, в результате чего получается отсортированная таблица лидеров *Leaderboard* (рисунок ниже). Затем **Trim** сохраняет пептиды с наибольшим значением, включая совпадения (например, для $N = 5$ семь пептидов с наибольшим значением, показанных синим цветом, будут сохранены), и удаляет все остальные пептиды из *Leaderboard*.

LEADERBOARD	PVT	PCP	VPC	VCP	PTV	TPV	TPC	TVP	PTP	VTP
LINEARSCORES	6	6	5	5	4	4	4	3	2	2

Рис. 4.22 Отсортированная таблица лидеров *Leaderboard*

```
Trim(Leaderboard, Spectrum, N, Alphabet, AminoAcidMass)
    for j ← 1 до |Leaderboard|
        Peptide ← j-й пептид в Leaderboard
        LinearScores(j) ← LinearScore(Peptide, Spectrum, Alphabet, AminoAcidMass)
    отсортировать Leaderboard в порядке уменьшения значений согласно
LinearScores
    отсортировать LinearScores в порядке уменьшения
    for j ← N + 1 до |Leaderboard|
        if LinearScores(j) < LinearScores(N)
            удалить все пептиды, начинающиеся с j-го пептида из Leaderboard
            return Leaderboard
    return Leaderboard
```

Сопутствующие материалы

Гаузе и лысенковщина

Термин «лысенковщина» относится ко времени политических репрессий против генетики в Советском Союзе, которые начались в конце 1920-х годов и продолжались в течение трех десятилетий, до смерти Сталина. Лысенковщина строилась на теориях наследования по приобретенным признакам, противоречащим законам Менделя.

В 1928 году Трофим Лысенко (на фото ниже), сын украинских крестьян, заявил, что нашел способ значительно увеличить урожайность пшеницы. Во вре-

мя правления Сталина советская пропаганда сосредоточилась на вдохновляющих историях граждан из рабочего класса и изображала Лысенко гением, хотя свои экспериментальные данные он сфабриковал. Воодушевленный своим внезапным статусом героя, Лысенко осудил генетику и начал продвигать свои собственные псевдонаучные взгляды. Он назвал генетиков «любителями мух и ненавистниками людей» и заявил, что они пытаются подорвать поступательное движение советского сельского хозяйства.

Рис. 4.23 Трофим Лысенко

Гаузе оказался в числе немногих советских биологов, не побоявшихся публично осудить Лысенко. К 1935 году Лысенко объявил, что, выступая против его теорий, генетики прямо противостоят учению марксизма. Сталин, находившийся в зале, первым зааплодировал, выкрикивая: «Браво, товарищ Лысенко!» Это событие дало Лысенко полную свободу клеветать на любых генетиков, выступивших против него; многие противники лысенковщины были приговорены к длительным срокам заключения или даже расстреляны.

После Второй мировой войны Лысенко не забыл критику Гаузе: сторонники Лысенко требовали исключения Гаузе из Российской академии наук. Лысенковцы предпринимали различные попытки предложить Гаузе осудить генетику и принять их лженауку. Гаузе был, вероятно, единственным советским биологом в то время, который мог игнорировать такие «приглашения», другими современными противниками лысенковщины были ведущие советские физики-ядерщики. Однако Сталин оставил Гаузе и физиков в покое; по мнению

Сталина, разработка антибиотиков и атомной бомбы были слишком важны. В 1949 году, когда директор российской тайной полиции (Лаврентий Берия) рассказал Сталину об ученых-диссидентах, Сталин ответил: «Убедитесь, что у наших ученых есть все необходимое для выполнения их работы, – добавив, – а расстрелять мы их всегда успеем».

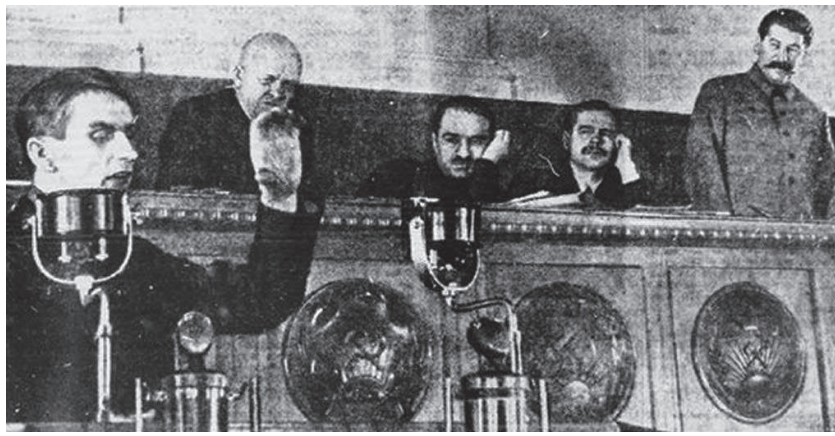


Рис. 4.24 Лысенко (крайний слева) выступает в Кремле в 1935 году, в президиуме Иосиф Сталин (крайний справа)

Открытие кодонов

В 1961 году Сидней Бреннер и Фрэнсис Крик установили правило «один кодон, одна аминокислота» в процессе трансляции белка. Они заметили, что удаление одного нуклеотида или двух последовательных нуклеотидов в гене резко изменило белок в результате. Как это ни парадоксально, удаление *трех* последовательных нуклеотидов привело лишь к незначительным изменениям в белке. Например, фраза

THE · SLY · FOX · AND · THE · SHY · DOG

превращается в тарабарщину после удаления одной буквы:

THE · SYF · OXA · NDT · HES · HYD · OG

или после удаления двух букв:

THE · SFO · XAN · DTH · ESH · YDO · G

но оставляет некоторый смысл после удаления трех букв:

THE · FOX · AND · THE · SHY · DOG

В 1964 году Чарльз Янофски продемонстрировал, что ген и белок, который он производит, коллинеарны, а это означает, что первый кодон кодирует первую аминокислоту в белке, второй кодон кодирует вторую аминокислоту и т. д. В течение следующих 13 лет биологи считали, что белок кодируется длинной последовательностью смежных триплетов нуклеотидов. Однако открытие **расщепленных генов** в 1977 году доказало обратное и потребовало вычислительной задачи предсказания местоположения генов с использованием только геномной последовательности.

Чувство кворума

Традиционное представление о том, что бактерии действуют как одиночки и редко взаимодействуют с остальной частью своей колонии, было опровергнуто открытием метода коммуникации, называемого **чувством кворума**. Это открытие показало, что бактерии способны к координированной деятельности, когда мигрируют к лучшему снабжению питательными веществами или принимают формирование **биопленки** для защиты от враждебной среды. «Язык», используемый для определения кворума, часто основан на обмене пептидами (а также другими молекулами), называемыми **бактериальными феромонами**. Характер связей между бактериями может быть дружеским или враждебным.

Когда одна бактерия выделяет феромоны в окружающую среду, их концентрация часто слишком мала, чтобы ее можно было обнаружить; однако, как только плотность популяции увеличивается, концентрации феромонов достигают порогового уровня, который позволяет бактериям в ответ активировать определенные гены.

Например, *Burkholderia cepacia* является патогеном, поражающим людей с **муковисцидозом**. Большинство пациентов, колонизированных *B. cepacia*, коинфицированы *Pseudomonas aeruginosa*. Корреляция двух штаммов у этих пациентов привела биологов к предположению, что межвидовая коммуникация с *P. aeruginosa* может помочь *B. cepacia* усилить собственную патогенность. Действительно, добавление *P. aeruginosa* к клонам *B. cepacia* приводит к значительному увеличению синтеза **протеаз** (т. е. ферментов, расщепляющих белки), что свидетельствует о наличии чувства кворума, – *B. cepacia* может извлекать выгоду из феромонов, производимых другим видом, чтобы повысить свои шансы на выживание.

Молекулярная масса

Дальтон (сокращенно **Да**) – это единица измерения атомной массы в молекулярном масштабе. Один дальтон эквивалентен одной двенадцатой части массы углерода-12 и имеет значение примерно $1{,}66 \cdot 10^{-27}$ кг. **Моноизотопная масса** молекулы равна сумме масс атомов в этой молекуле с использованием массы наиболее распространенного **изотопа** для каждого элемента (см. таблицу ниже).

Аминокислота	Трехбуквенный код	Молекулярная формула	Масса (Да)
Alanine	Ala	C_3H_5NO	71,03711
Cysteine	Cys	C_3H_5NOS	103,00919
Aspartic acid	Asp	$C_4H_5NO_3$	115,02694
Glutamic acid	Glu	$C_5H_7NO_3$	129,04259
Phenylalanine	Phe	C_9H_9NO	147,06841
Glycine	Gly	C_2H_3NO	57,02146
Histidine	His	$C_6H_7N_3O$	137,05891
Isoleucine	Ile	$C_6H_{11}NO$	113,08406
Lysine	Lys	$C_6H_{12}N_2O$	128,09496
Leucine	Leu	$C_6H_{11}NO$	113,08406
Methionine	Met	C_5H_9NOS	131,04049
Asparagine	Asn	$C_4H_6N_2O_2$	114,04293
Proline	Pro	C_5H_7NO	97,05276
Glutamine	Gln	$C_5H_8N_2O$	128,05858
Arginine	Arg	$C_6H_{12}N_4O$	156,10111
Serine	Ser	$C_3H_5NO_2$	87,03203
Threonine	Thr	$C_4H_7NO_2$	101,04768
Valine	Val	C_5H_9NO	99,06841
Tryptophan	Trp	$C_{11}H_{10}N_2O$	186,07931
Tyrosine	Tyr	$C_9H_9NO_2$	163,06333

Сленоцистеин и пирролизин

Селеноцистеин – это протеиногенная аминокислота, которая существует во всех царствах жизни как строительный блок особого класса, называемый селенопротеинами. В отличие от других аминокислот селеноцистеин напрямую не закодирован в генетическом коде. Вместо этого он особым образом кодируется кодоном UGA, который обычно является стоп-кодоном в механизме, известном как **трансляционное перекодирование**.

Пирролизин представляет собой протеиногенную аминокислоту, которая содержится в некоторых археях и метан-продуцирующих бактериях. В организмах, содержащих пирролизин, эта аминокислота кодируется UAG, который также обычно действует как стоп-кодон.

Псевдополиномиальный алгоритм для Теоремы магистрали

Если $A = (a_1 = 0, a_2, \ldots, a_n)$ – множество из n точек на отрезке в порядке возрастания $(a_1 < a_2 < \cdots < a_n)$, то ΔA обозначает совокупность всех попарных разностей между точек в A. Например, если $A = (0, 2, 4, 7,)$, то

$$\Delta A = (-7, -5, -4, -3, -2, -2, 0, 0, 0, 0, 2, 2, 3, 4, 5, 7).$$

Задача магистрали требует от нас восстановить A по ΔA.

Задача магистрали: *зная все попарные расстояния между точками на отрезке* ΔA*, восстановите положение точек* A*.*

Input: набор целых чисел L.

Output: набор целых чисел A таких, что $\Delta A = L$.

Теперь мы наметим подход к решению Теоремы магистрали, полиномиальный на отрезке сегмента. Для набора целых чисел $A = (a_1 < a_2 < \cdots < a_n)$ **производящая функция** A является многочленом:

$$A(x) = \sum_{i=1}^{n} x^{a_i}.$$

Например, если $A = (0, 2, 4, 7, 10)$, то

$$A(x) = x^0 + x^2 + x^4 + x^7,$$

$$\Delta A(x) = x^{-7} + x^{-5} + x^{-4} + x^{-3} + 2x^{-2} + 4x^0 + 2x^2 + x^3 + x^4 + x^5 + x^7.$$

Вы можете проверить, что производящая функция для $\Delta A(x)$ равна $A(x) \cdot A(x^{-1})$. Таким образом, задача магистрали сводится к задаче полиномиальной факторизации. Подобно тому, как целое число можно разложить на простые множители, многочлен с целыми коэффициентами можно разложить на «простые» многочлены с целыми коэффициентами. Если мы сможем факторизовать $\Delta A(x)$ и определить, какие простые множители вносят вклад в $A(x)$ и какие простые множители вносят вклад в $A(x^{-1})$, то мы будем знать $A(x)$ и, следовательно, A. В 1982 году Розенблатт и Сеймур описали такой метод представления $\Delta A(x)$ как $A(x) \cdot A(x^{-1})$. Поскольку многочлен можно разложить на множители за время, полиномиальное по его максимальному показателю, $\Delta A(x)$ можно разложить за время, полиномиальное по отношению к общей длине отрезка, что дает желаемый псевдополиномиальный алгоритм для задачи магистрали.

ОСТАНОВИТЕСЬ и задумайтесь. Можно ли изменить описанный выше метод с производящей функцией для случая, когда в попарных разностях есть ошибки?

Расщепленные гены

В 1977 году Филлип Шарп и Ричард Робертс независимо друг от друга открыли **расщепленные гены**, т. е. гены, образованные несмежными участками ДНК.

Шарп гибридизировал РНК, кодирующую белок аденовируса, называемый **гексоном**, с одноцепочечной ДНК аденовируса. Если бы ген гексона был не-

прерывным, то он ожидал увидеть гибридизацию один-в-один оснований РНК с основаниями ДНК.

Тем не менее, к удивлению Шарпа, когда он наблюдал гибридизацию РНК-ДНК под электронным микроскопом, он увидел три петлевых структуры, а не непрерывный комплементарный сегмент, предполагаемый моделью непрерывного гена (рисунок ниже). Это наблюдение предполагает, что мРНК гексона должна быть построена из четырех несмежных фрагментов генома аденовируса. Эти четыре сегмента, называемые **экзонами**, разделены тремя фрагментами (петлями на рисунке ниже), называемыми **интронами**, формируя, таким образом, расщепленный ген. Расщепленные гены аналогичны журнальной статье, напечатанной на стр. 12, 17, 40 и 95, между которыми помещено много страниц рекламы.

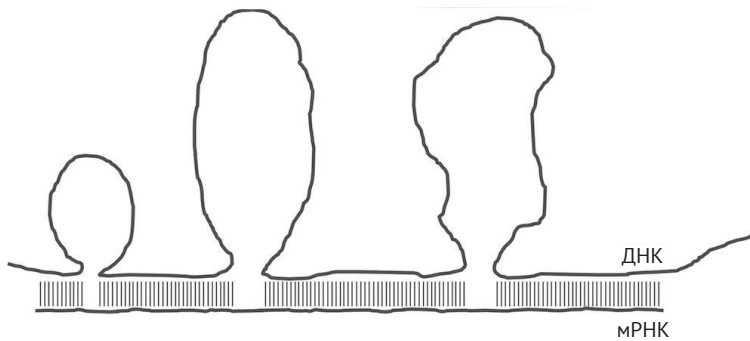


Рис. 4.25 Представление эксперимента Шарпа с электронной микроскопией, который привел к открытию расщепленных генов. Когда РНК гексона гибридизуется с породившей ее ДНК, образуются три отдельные петли. Поскольку петли присутствуют в ДНК и отсутствуют в РНК, эти петли (называемые интронами) должны быть удалены в процессе образования РНК

Открытие расщепленных генов поставило интересную задачу: *что происходит с интронами*? Другими словами, РНК, транскрибируемая из расщепленного гена (называемая **предшественником мРНК**, или **пре-мРНК**), должна быть длиннее, чем РНК, используемая в качестве матрицы для синтеза белка (называемая **матричной РНК**, или **мРНК**). («Информационная РНК», или «иРНК», – это синонимы мРНК. – *Прим. ред.*) Какой-то биологический процесс должен удалить интроны из пре-мРНК и соединить экзоны в единую цепь мРНК. Этот процесс преобразования пре-мРНК в мРНК известен как **сплайсинг** и осуществляется молекулярной машиной, называемой **сплайсосомой**.

Открытие расщепленных генов привело ко многим новым направлениям исследований. Биологи до сих пор спорят, какой цели служат интроны; некоторые интроны рассматриваются как «мусорная ДНК», тогда как другие содержат важные регуляторные элементы. Кроме того, разделение гена на экзоны часто варьируется от вида к виду. Например, ген в геноме курицы может иметь другое число экзонов, чем родственный ген в геноме человека.

Библиографические примечания

Псевдополиномиальный алгоритм для Теоремы магистрали был предложен Rosenblatt and Seymour, 1982[1]. Первый алгоритм секвенирования циклопептидов был предложен Ng et al., 2009[2]. Tang et al., 1999[3], открыли тета-дефенсин. Venkataraman et al., 2009[4], показали, что человеческие клетки можно обманом заставить производить тета-дефензин.

[1] https://epubs.siam.org/doi/10.1137/0603035?mobileUi=0.
[2] https://www.nature.com/articles/nmeth.1350.
[3] https://www.ncbi.nlm.nih.gov/pubmed/10521339.
[4] https://www.ncbi.nlm.nih.gov/pubmed/19402752.

Глава 5

Как мы сравниваем участки ДНК?

Динамическое программирование

Взлом нерибосомного кода

Клуб галстуков РНК

После открытия Уотсоном и Криком структуры двойной спирали ДНК в 1953 году физик Джордж Гамов основал «Клуб галстуков РНК» для известных ученых. Каждый член клуба должен был носить галстук, вышитый двойной спиралью, в клуб входили также четыре почетных члена (по одному на каждый нуклеотид). Гамов хотел, чтобы «Клуб галстуков РНК» выполнял больше, чем социальную функцию; собрав ведущие научные умы, он надеялся расшифровать послание, скрытое в ДНК, путем определения того, как РНК преобразуется в аминокислоты. Действительно, Сидней Бреннер и Фрэнсис Крик год спустя первыми открыли, что аминокислоты транслируются с кодонов (т. е. триплетов нуклеотидов).

«Клуб галстуков РНК» в конечном итоге мог похвастаться восемью нобелевскими лауреатами, но вся эта интеллектуальная огневая мощь не помогла его членам сделать оставшиеся шаги к расшифровке генетического кода. В 1961 году Маршалл Ниренберг синтезировал цепи РНК, состоящие только из урацила (…UUUUUUUUUUU…), добавил рибосомы и аминокислоты и получил пептид, состоящий только из фенилаланина (…PhePhePhePhe…). Таким образом, Ниренберг пришел к выводу, что кодон РНК UUU кодирует аминокислоту фенилаланин. После успеха Ниренберга Хар Гобинд Корана синтезировал цепь РНК …**UCUCUCUCUCUC**… и продемонстрировал, что она транслируется в … **SerLeuSerLeu**… После этих открытий остальная часть рибосомного генетического кода была быстро определена.

Почти четыре десятилетия спустя Мохамед Марахиэл решил заняться гораздо более сложной задачей расшифровки **нерибосомного кода**. Из нашей главы о секвенировании антибиотиков вы помните, что бактерии и грибы производят антибиотики и другие **нерибосомные пептиды (NRP)** без какой-либо зависимости от рибосомы и генетического кода. Вместо этого эти организмы производят NRP, используя гигантский белок, называемый NRP-синтетазой:

ДНК → РНК → NRP-синтетаза → NRP.

Синтетаза NRP, которая кодирует антибиотик тироцидин B1 длиной в 10 аминокислот, включает 10 сегментов, называемых **доменами аденилирования** (**А-домены**); каждый А-домен имеет длину около 500 аминокислот и отвечает за добавление одной аминокислоты к тироцидину B1.

Поколением ранее члены «Клуба галстуков РНК» задались вопросом: «Как РНК кодирует аминокислоту?» Теперь Марахиел решил ответить на гораздо более сложный вопрос: «Как каждый А-домен кодирует аминокислоту?»

От сравнения белков к нерибосомному коду

К счастью, Марахиелу уже были известны аминокислотные последовательности некоторых А-доменов, а также аминокислоты, которые они добавляют

к растущему пептиду. Ниже приведены три из этих А-доменов (взятых из трех разных бактерий), которые кодируют аспарагиновую кислоту (Asp), орцепин (Orn) и валин (Val) соответственно. В интересах экономии места мы покажем вам только короткие фрагменты, взятые из трех А-доменов:

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA
AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSAPTMISSLEILFAAGDRLSSQDAILARRAVGSGVYNAYGPTENTVLS
```

Рис. 5.1 Фрагменты трех А-доменов, кодирующие аспарагиновую кислоту (Asp), орцепин (Orn) и валин (Val)

ОСТАНОВИТЕСЬ и задумайтесь. Теперь у вас есть часть данных, которые были у Марахиела в 1999 году, когда он открыл нерибосомный код. Что бы вы сделали, чтобы его выдать в результате?

Марахиел предположил, что, поскольку А-домены имеют одинаковую функцию (т. е. добавление аминокислоты к растущему пептиду), разные А-домены должны иметь сходные части. А-домены также должны иметь разные части для включения разных аминокислот. Однако только три консервативных столбца (показаны красным ниже) являются общими для трех последовательностей и, вероятно, возникли случайно:

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA
AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSAPTMISSLEILFAAGDRLSSQDAILARRAVGSGVYNAYGPTENTVLS
```

Рис. 5.2 Общие столбцы доменов

ОСТАНОВИТЕСЬ и задумайтесь. Чем еще похожи эти три последовательности?

Если мы сдвинем вторую последовательность только на одну аминокислоту вправо, добавив **пробел** («–») в начало последовательности, то мы обнаружим 11 консервативных столбцов!

Представлено на рис. 5.3 (вверху). Добавление еще нескольких пробелов показывает 14 консервативных столбцов – рис. 5.3 (в середине). Дальнейший сдвиг показывает уже 19 консервативных столбцов – рис. 5.3 (внизу).

Оказывается, красные столбцы представляют собой **консервативное ядро**, общее для многих А-доменов. Теперь, когда Марахиел знал, как правильно *выровнять* А-домены, он предположил, что некоторые из оставшихся столбцов переменных должны кодировать Asp, Orn и Val. Он обнаружил, что нерибосомный код определяется нерибосомными сигнатурами длиной 8 аминокислот, которые показаны ниже фиолетовыми столбцами.

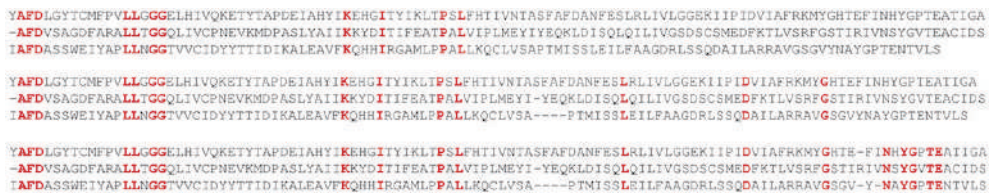


Рис. 5.3 Выявление консервативных столцов путем сдвига последовательности

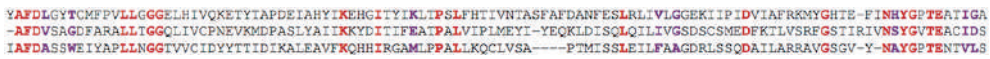


Рис. 5.4 Нерибосомные сигнатуры

Фиолетовые столбцы определяют обозначения **LTKVGHIG**, **VGEIGSID** и **AWMFAAVL**, кодирующие для Asp, Orn и Val соответственно:

LTKVGHIG → Asp
VGEIGSID → Orn
AWMFAAVL → Val

Важно отметить, что без предварительного создания консервативного ядра Марахиел не смог бы вывести нерибосомный код, поскольку 24 аминокислоты в сигнатурах выше не совпадают в исходном выравнивании:

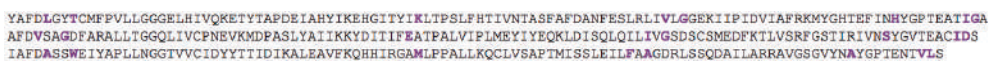


Рис. 5.5 Исходное выравнивание

Даже после идентификации сохранившегося ядра вам может быть интересно, был ли у Марахиела хрустальный шар; почему он выбрал именно эти 8 фиолетовых столбцов? Почему в сигнатурах должно быть 8 аминокислот, а не 5, а лучше 3? Достаточно сказать, что через 15 лет после первоначального открытия Марахиела нерибосомный код до сих пор полностью не изучен. Смотрите **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Светлячки и нерибосомный код**, чтобы лучше осознать сложности, лежащие в основе задачи, решенной Марахиелом.

Что общего между онкогенами и факторами роста?

Взлом нерибосомного кода Марахиела – лишь одна из многих биологических проблем, для решения которых полезно сравнение последовательностей. Еще один поразительный пример силы этого метода был получен в 1983 году, когда Рассел Дулиттл сравнил недавно секвенированный ген **фактора роста тромбоцитов** (**PDGF**) со всеми другими генами, известными в то время. Дулиттл ошеломил биологов-онкологов, когда показал, что PDGF очень похож на последовательность гена, известного как **v-sis**. Сходство двух генов вызывало недоумение, поскольку их функции сильно различались; ген PDGF кодирует

белок, стимулирующий рост клеток, тогда как v-sis является онкогеном или геном вирусов, вызывающим ракоподобную трансформацию инфицированных клеток человека. После открытия Дулиттла ученые выдвинули гипотезу о том, что некоторые формы рака могут быть вызваны тем, что хороший ген делает правильные вещи в неподходящее время. Связь между PDGF и v-sis установила новую парадигму; поиск всех новых последовательностей в базах данных последовательностей теперь является первостепенной задачей в геномике.

Однако остается вопрос: как лучше алгоритмически сравнивать последовательности? Возвращаясь к примеру с A-доменом (рис. 5.4), вставка пробелов для раскрытия сохраненного ядра, вероятно, показалась вам фокусом. Совершенно неясно, какой алгоритм мы использовали, чтобы решить, куда вставить символы пробела или как мы должны количественно определить «наилучшее» выравнивание трех последовательностей.

Введение в выравнивание последовательностей

Выравнивание[1] *последовательности как игра*

Для упрощения мы будем сравнивать только две последовательности за раз, вернувшись к сравнению множественных последовательностей в конце главы. Расстояние Хэмминга, которое подсчитывает несовпадения в двух строках, строго предполагает, что мы выравниваем *i*-й символ одной последовательности с *i*-м символом другой. Однако, поскольку биологические последовательности подвержены вставкам и делециям, часто бывает так, что *i*-й символ одной последовательности соответствует символу в совершенно другом месте другой последовательности. Таким образом, цель состоит в том, чтобы найти наиболее подходящее соответствие символов.

Например, ATGCATGC и TGCATGCA не имеют совпадающих позиций, поэтому их расстояние Хэмминга равно 8:

```
ATGCATGC
TGCATGCA
```

Тем не менее эти строки имеют семь совпадающих позиций, если мы выровняем их по-разному:

```
ATGCATGC—
—TGCATGCA
```

Строки ATGCTTA и TGCATTAA имеют более тонкое сходство:

```
ATGC—TTA—
—TGCATTAA
```

[1] Наряду с термином «выравнивание» используются термины «сравнение» и «сопоставление». – *Прим. ред.*

Эти примеры приводят нас к постулированию понятия хорошего выравнивания как соответствия как можно большему числу символов.

Вы можете думать о максимизации количества совпадающих символов в двух строках как об игре для одного человека. На каждом ходу у вас есть два варианта. Вы можете удалить первый символ из каждой последовательности, и в этом случае вы заработаете очко, если символы совпадут; в качестве альтернативы можете удалить первый символ из любой из двух последовательностей, и в этом случае вы не заработаете очков, но можете настроить себя на получение большего количества очков в последующих ходах. Ваша цель состоит в том, чтобы максимизировать количество очков.

На рис. 5.6 мы показываем один из способов этой игры для последовательностей ATGTTATA и ATCGTCC, набирающий 4 очка. Каждый раз, удаляя символ слева, мы добавляем его к растущему выравниванию ATGTTATA и ATCGTCC справа. Когда за ход удаляется только один символ, мы выравниваем его по пробелу.

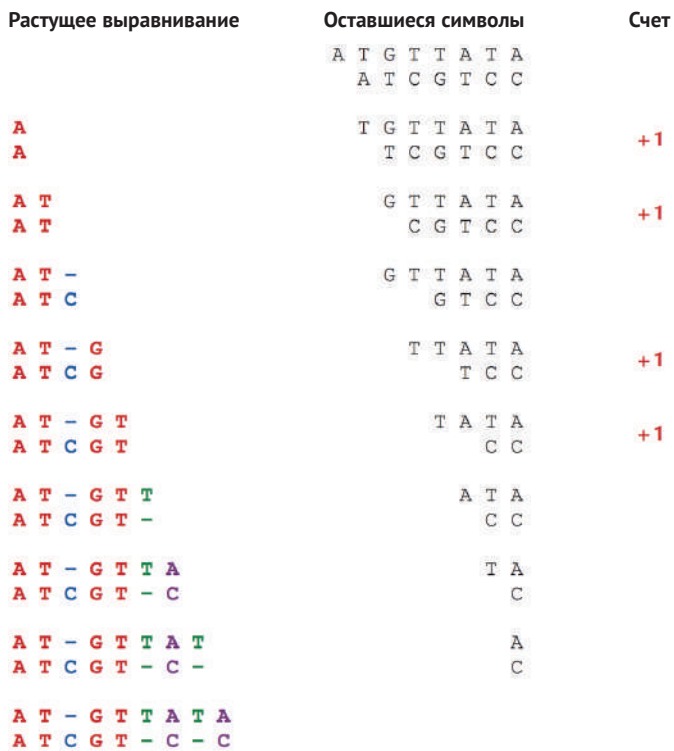


Рис. 5.6 Один из способов игры в выравнивание для строк ATGTTATA и ATCGTCC со счетом 4. На каждом шаге мы выбираем удаление одного или обоих символов слева из двух последовательностей в столбце «оставшиеся символы». Если мы удалим оба символа, то выровняем их по «растущему выравниванию». Если удаляем только один символ, то мы выравниваем этот символ с символом пробела в растущем выравнивании. Совпадающие символы отображаются красным цветом (и получают 1 балл). Несовпадающие символы отображаются фиолетовым цветом; символы, выровненные по пробелам, отображаются синим или зеленым цветом в зависимости от того, из какой последовательности они были удалены

Рисунок, представленный выше, показывает только один из многих возможных способов игры в выравнивание для строк ATGCATGC и TGCATGCA. Можете ли вы найти еще лучший способ играть в эту игру для строк?

Выравнивание последовательностей и самая длинная общая подпоследовательность

Теперь мы определим выравнивание последовательностей v и ω как двухстрочную матрицу, так что первая строка содержит символы v (по порядку), вторая строка – символы ω (по порядку) и символы пробела (называемые промежуточными символами и показанные ниже в виде тире). Они могут быть вставлены в обе строки, если два символа пробела не выровнены друг относительно друга. Вот выравнивание **ATGTTATA** и **ATCGTCC** из рисунка выше.

```
AT–GTTATA
ATCGT–C–C
```

Выравнивание представляет собой один из возможных сценариев, по которому последовательность v могла бы превратиться в ω. Столбцы, содержащие одну и ту же букву в обеих строках, называются **совпадениями** и представляют собой консервативные нуклеотиды, тогда как столбцы, содержащие разные буквы, называются несовпадениями и представляют собой однонуклеотидные замены. Столбцы, содержащие символ пробела, называются **индел**, столбец, содержащий символ пробела в верхней строке выравнивания, – **вставкой**, так как подразумевает вставку символа при преобразовании v в ω; столбец, содержащий символ пробела в нижней строке выравнивания, называется **удалением** (делецией), поскольку он указывает на удаление символа при преобразовании v в ω. Приведенное выше выравнивание имеет четыре совпадения, два несовпадения, одну вставку и два удаления.

Совпадения в выравнивании двух строк определяют **общую подпоследовательность** двух строк или последовательность символов, появляющихся в одном и том же порядке (хотя и не обязательно последовательно) в обеих строках. Например, приведенное выше выравнивание указывает на то, что **ATGT** является общей последовательностью **ATGTTATA** и **ATCGTCC**. Таким образом, выравнивание двух строк, максимизирующее количество совпадений, соответствует **самой длинной общей подпоследовательности (LCS)** этих строк. Обратите внимание, что две строки могут иметь более одной самой длинной общей подпоследовательности.

Задача на определение самой длинной общей подпоследовательности: *найдите самую длинную общую подпоследовательность двух строк.*

Input: две строки.

Output: самая длинная общая подпоследовательность этих строк.

Упражнение. Найдите все самые длинные общие подпоследовательности строк ACTGCA и CATCGC. Сколько таких подпоследовательностей вы нашли?

Если мы ограничимся двумя A-доменами, кодирующими Asp и Orn из введения, то в дополнение к уже найденным 19 совпадениям мы сможем найти еще 10 совпадений (показаны синим цветом ниже), что даст общую подпоследовательность длиной 29.

```
YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTASFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
-AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI-YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
```

Рис. 5.7 Общая подпоследовательность Asp и Orn

ОСТАНОВИТЕСЬ и задумайтесь. Какова самая длинная общая подпоследовательность этих строк?

Ни один из алгоритмических методов, которые мы изучали до сих пор, не поможет нам решить задачу самой длинной общей подпоследовательности, поэтому, прежде чем попросить вас решить эту задачу, мы изменим курс и опишем другую задачу, которая может показаться совершенно не связанной с выравниванием последовательностей.

Туристическая задача Манхэттена

Какова наилучшая стратегия осмотра достопримечательностей?

Представьте, что вы турист в Мидтауне Манхэттена и хотите увидеть как можно больше достопримечательностей по пути от угла 59-й улицы и 8-й авеню до угла 42-й улицы и 3-й авеню (рисунок ниже). Однако у вас мало времени, и на каждом перекрестке вы можете двигаться только на юг (↓) или на восток (→). Вы можете выбирать из множества различных путей по карте, но ни один путь не пройдет через все достопримечательности. Задача найти маршрут через город, удовлетворяющий данным правилам, который пройдет через наибольшее количество достопримечательностей, называется **туристической задачей Манхэттена**.

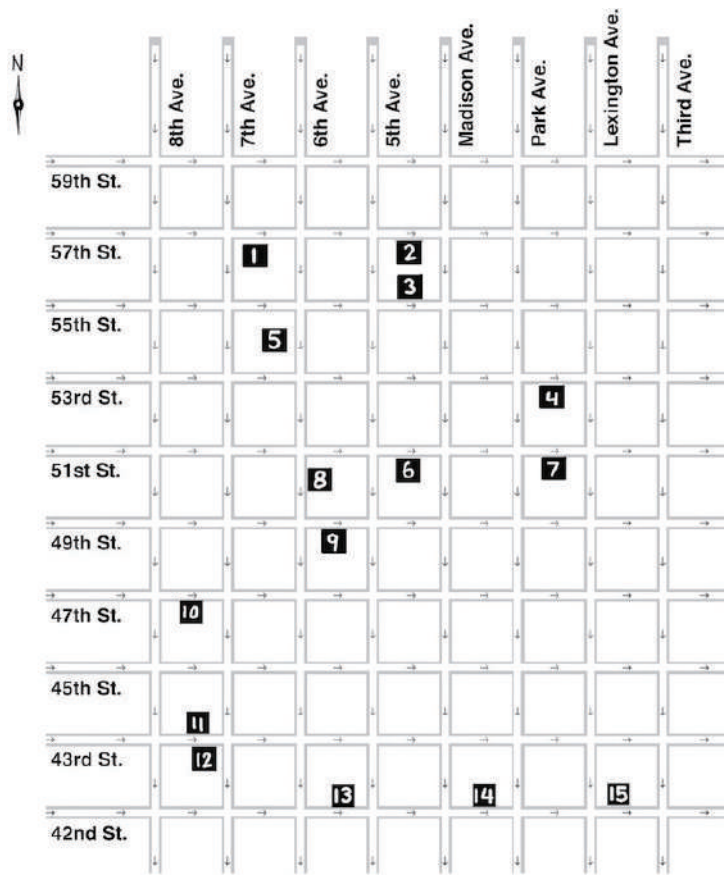


Рис. 5.8 Упрощенный Мидтаун Манхэттена. Вы начинаете на пересечении 59-й улицы и 8-й авеню в северо-западном углу и заканчиваете на пересечении 42-й улицы и 3-й авеню в юго-восточном углу, двигаясь только на юг или восток между перекрестками. Показанные достопримечательности: Карнеги-холл (1), Тиффани & Co. (2), Сони-билдинг (3), Музей современного искусства (4), отель «Четыре сезона» (5), собор Святого Патрика (6), Дженерал-Электрик-билдинг (7), Раидо-Сити-мюзик-холл (8), Рокфеллер-центр (9), Парамаунт-билдинг (10), Нью-Йорк-Таймс-билдинг (11), Таймс-сквер (12), Генеральное общество механиков и торговцев (13), Центральный вокзал (14) и Крайслер-билдинг (15)

Мы представим карту Манхэттена в виде ориентированного графа *ManhattanGraph*, в котором мы моделируем каждый перекресток как узел и представляем каждый городской квартал между двумя перекрестками в виде направленного ребра, указывающего допустимое направление движения (↓ или →), как показано на рис. 5.9. Затем мы присваиваем каждому направленному ребру **вес**, равный количеству достопримечательностей вдоль соответствующего квартала. Начальный (синий) узел называется **узлом-источником**, а конечный (красный) узел называется **узлом-стоком**. Добавление весов вдоль пути

от источника к стоку дает количество достопримечательностей на этом пути; поэтому, чтобы решить эту задачу туриста на Манхэттене, нам нужно найти в *ManhattanGraph* **путь максимального веса**, соединяющий источник со стоком (также называемый **самым длинным путем**).

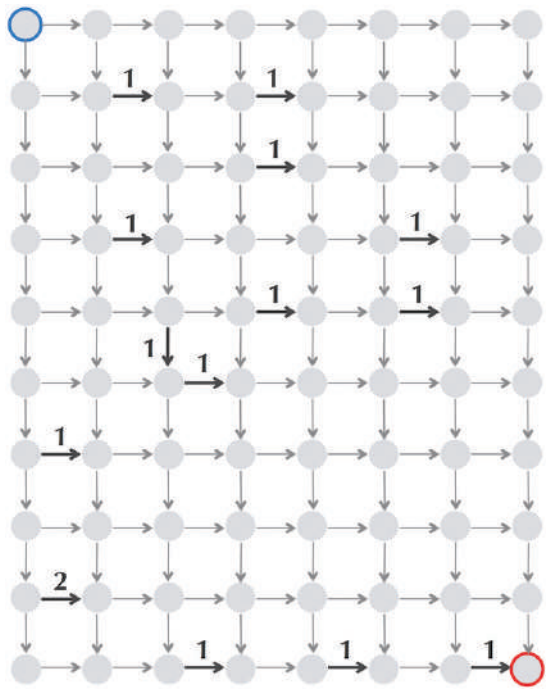


Рис. 5.9 Ориентированный граф *ManhattanGraph*, в котором каждое ребро взвешено по количеству достопримечательностей в этом городском квартале (веса ребер, равные 0, не показаны)

Мы можем смоделировать любую прямоугольную сетку улиц, используя аналогичный ориентированный граф; на рисунке ниже показан граф для гипотетического города с еще большим количеством достопримечательностей. В отличие от декартовой плоскости мы ориентируем оси этой сетки вниз и вправо. Поэтому синему узлу-источнику присваиваются координаты (0, 0), а красному узлу-стоку – координаты (n, m).

Из этого обсуждения вытекает следующее обобщение нашей исходной задачи.

Туристическая задача Манхэттена: *найти самый длинный путь в городе прямоугольной структуры.*

Input: взвешенная прямоугольная сетка размера $n \times m$ с $n + 1$ строками и $m + 1$ столбцами.

Output: самый длинный путь от источника (0,0) до стока (n, m) в сетке.

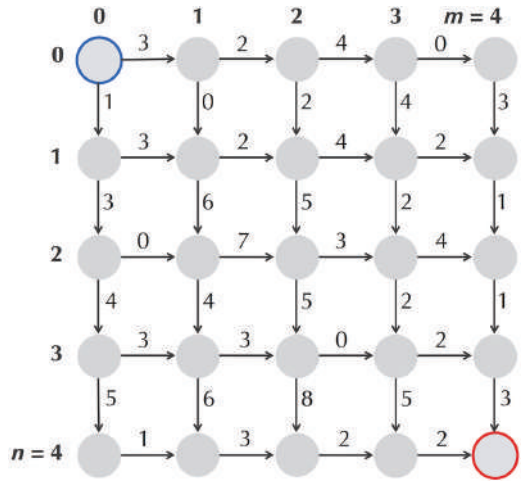


Рис. 5.10 Городская сетка размера $n \times m$, представленная в виде графа со взвешенными ребрами для $n = m = 4$. Нижний левый узел имеет индекс (4, 0), а верхний правый узел – индекс (0, 4)

Упражнение. Сколько различных путей существует от истока до стока в прямоугольной сетке 16×2?

Примечание. Имейте в виду, что, согласно нашим обозначениям, сетка 16×12 содержит $17 \cdot 13 = 221$ узел.

Применение алгоритма грубой силы к туристической задаче Манхэттена нецелесообразно, поскольку количество путей огромно. Разумный жадный алгоритм будет выбирать между двумя возможными направлениями в каждом узле (→ или ↓) в зависимости от того, сколько достопримечательностей вы увидите, если переместитесь только на один квартал на юг, а не на один квартал на восток. Например, на рисунке ниже мы начинаем с движения на восток от (0, 0), а не на юг, потому что горизонтальный край имеет три достопримечательности, а вертикальный край – только одну. К сожалению, этот жадный алгоритм может упустить самый длинный путь в долгосрочной перспективе.

ОСТАНОВИТЕСЬ и задумайтесь. Найдите более длинный путь на рисунке ниже.

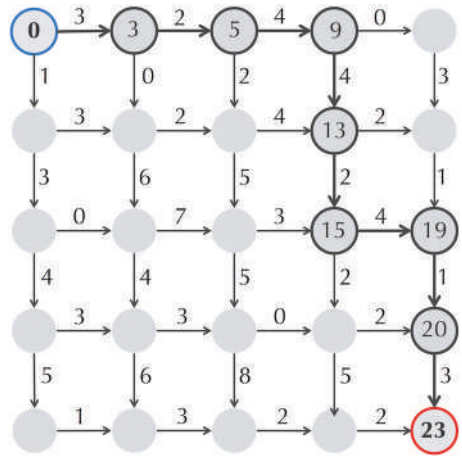


Рис. 5.11 Выделенный путь через этот граф, найденный жадным алгоритмом, не является самым длинным путем

Достопримечательности в произвольном ориентированном графе

В действительности улицы манхэттенского Мидтауна не образуют идеальную прямоугольную сетку, потому что Бродвей-авеню пересекает сетку по диагонали, но сеть улиц все же можно представить в виде ориентированного графа. На самом деле туристическая задача Манхэттена – это всего лишь частный случай более общей задачи поиска длиннейшего пути в произвольном ориентированном графе, например показанном ниже.

Задача поиска самого длинного пути в ориентированном графе: *найти самый длинный путь между двумя узлами в ориентированном графе, взвешенном по ребрам.*

Input: взвешенный по ребрам ориентированный граф с узлами-источниками и стоками.

Output: самый длинный путь от источника к стоку в ориентированном графе.

ОСТАНОВИТЕСЬ и задумайтесь. Какова длина самого длинного пути между источником и стоком в ориентированном графе, показанном на рисунке ниже справа?

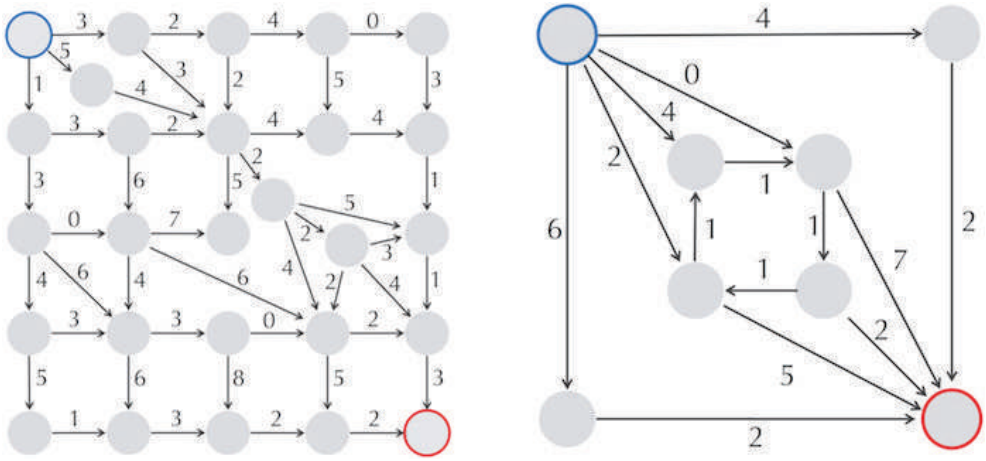


Рис. 5.12 Ориентированные графы, соответствующие гипотетической нерегулярной сетке городов

Если бы ориентированный граф содержал ориентированный цикл (например, четыре центральных ребра весом 1 на приведенном ниже графе), то турист мог бы пересекать этот цикл бесконечно, снова и снова посещая одни и те же достопримечательности и создавая путь огромной длины. По этой причине графы, которые мы будем рассматривать в этой главе, не содержат ориентированных циклов; такие графы **называются ориентированными ациклическими графами** (**DAG**).

Задача самого длинного пути в DAG: *найти самый длинный путь между двумя узлами в DAG со взвешенными ребрами.*

Input: взвешенный граф DAG с узлами источника и стока.

Output: самый длинный путь от источника к стоку в DAG.

ОСТАНОВИТЕСЬ и задумайтесь. Видите ли вы какое-либо сходство между самым длинным путем в задаче DAG и самой длинной общей задачей последовательности?

Выравнивание последовательности – это замаскированная туристическая задача Манхэттена

Ниже мы добавляем два массива целых числа к выравниванию ATGTTATA и ATCGTCC. Массив [0 1 2 2 3 4 5 6 7 8] содержит количество символов ATGTTATA, использованных до данного столбца в выравнивании. Точно так же массив

[0 1 2 3 4 5 5 6 6 7] содержит количество символов ATCGTCC, использованных до данного столбца (рис. 5.13 (вверху)).

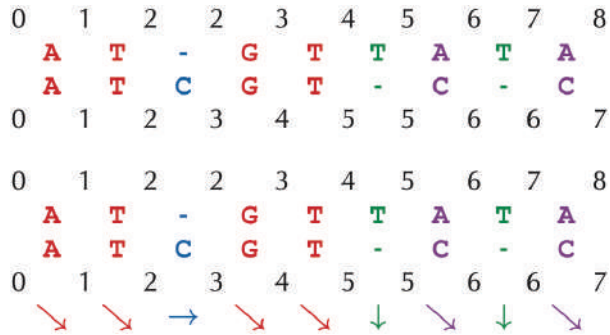


Рис. 5.13 Добавление массивов к выравниванию

Мы добавляем третий массив [↘ ↘ → ↘ ↘ ↓ ↘ ↓ ↘], записывая, представляет ли каждый столбец **совпадение**/**несовпадение** (↘/↘), **вставку** (→) или **удаление** (↓) (рис. 5.13 (внизу)).

Этот третий массив соответствует пути от источника к стоку в прямоугольной сетке 8×7. *i*-й узел этого пути состоит из *i*-го элемента [0 1 2 2 3 4 5 6 7 8] и *i*-го элемента [0 1 2 3 4 5 5 6 6 7]:

$$(0, 0) \searrow (1, 1) \searrow (2, 2) \rightarrow (2, 3) \searrow (3, 4) \searrow (4, 5) \downarrow (5, 5) \searrow (6, 6) \downarrow (7 , 6) \searrow (8, 7).$$

Этот путь показан на рис. 5.14. Обратите внимание, что, помимо горизонтальных и вертикальных ребер, мы добавили диагональные ребра, соединяющие (i, j) с $(i + 1, j + 1)$.

Мы называем этот DAG (воспроизведенный ниже) **графом выравнивания** строк v и ω, обозначаемым *AlignmentGraph*(v, ω), и называем путь от источника к стоку в этом DAG **путем выравнивания**. Каждое выравнивание v и ω можно рассматривать как набор инструкций для построения уникального пути выравнивания в *AlignmentGraph*(v, ω), где каждое **совпадение**/**несовпадение**, **вставка** или **удаление** соответствует ребру ↘/↘, → и ↓ соответственно. Таким образом, каждое выравнивание строк v и ω соответствует пути в *AlignmentGraph*(v, ω), и наоборот.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы использовать граф выравнивания, чтобы найти самую длинную общую подпоследовательность двух строк?

Упражнение. Постройте выравнивание ATGTTATA и ATCGTCC в соответствии с путем выравнивания, показанным ниже.

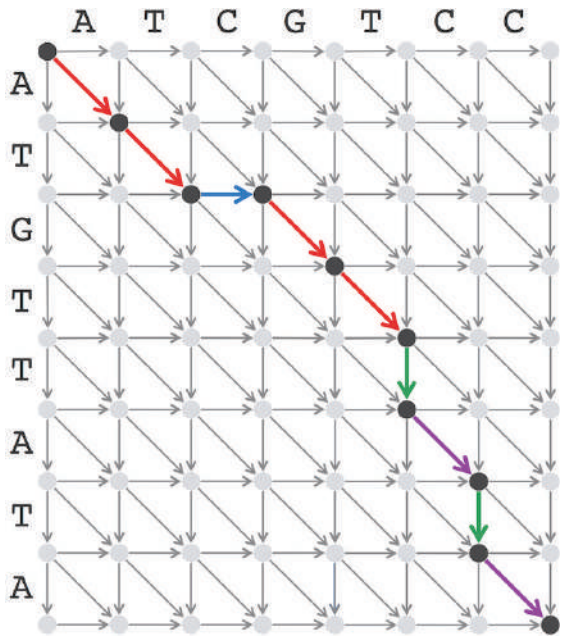


Рис. 5.14 Путь от источника к стоку в прямоугольной сетке 8×7

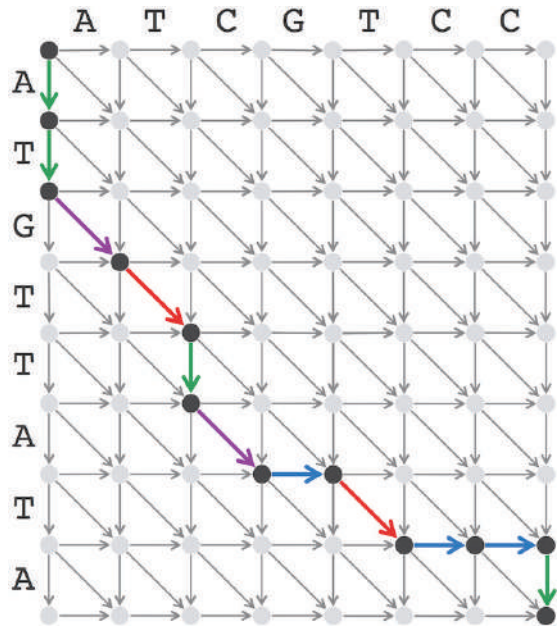


Рис. 5.15 Путь выравнивания

Напомним, что поиск самой длинной общей подпоследовательности двух строк эквивалентен поиску выравнивания этих строк, максимизирующего количество совпадений. На рисунке ниже мы выделяем все диагональные ребра AlignmentGraph(ATGTTATA, ATCGTCC), соответствующие совпадениям. Если мы присвоим вес 1 всем этим ребрам и 0 всем остальным ребрам, то задача о самой длинной общей подпоследовательности эквивалентна поиску самого длинного пути в этом взвешенном графе DAG! Таким образом, нам нужно разработать алгоритм для наибольшего пути в задаче DAG, но для этого нужно больше узнать о динамическом программировании, мощной алгоритмической парадигме, которая используется для решения тысяч задач из различных научных областей.

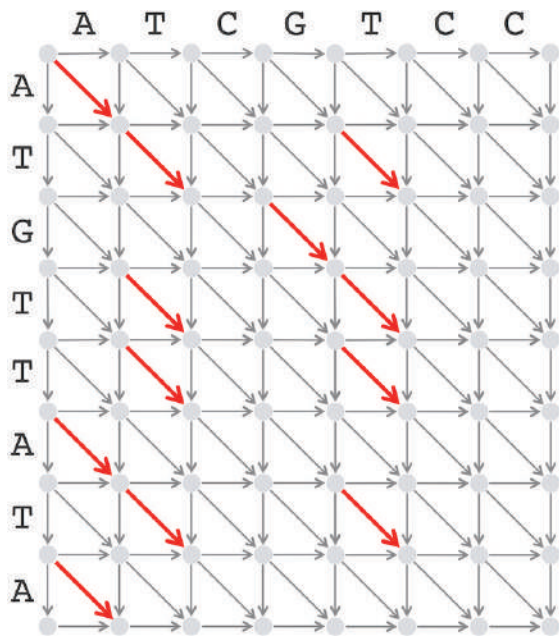


Рис. 5.16 *AlignmentGraph*(ATGTTATA, ATCGTCC) со всеми ребрами веса 1, окрашенными в красный цвет (все остальные ребра имеют вес 0). Эти ребра соответствуют потенциально совпадающим символам в выравнивании двух строк

Введение в динамическое программирование: задача размена монет

Жадный обмен денег

Представьте, что вы купили этот учебник в книжном магазине за 69,24 долл., за который заплатили 70 долл. наличными. Вам причитается сдача в размере 76 центов, и теперь кассир должен решить, дать ли вам горсть 76 одноцентовых монет или только четыре монеты (25 + 25 + 25 + 1 = 76). Дать сдачу в этом примере несложно, но он проливает свет на более общую задачу – как кассир может дать сдачу, используя наименьшее количество монет?

Разные валюты имеют разную стоимость монет, или **номиналы**. В США номиналы монет: (100, 50, 25, 10, 5, 1); в Римской республике они были (120, 40, 30, 24, 20, 10, 5, 4, 1). Эвристика, используемая кассирами во всем мире для сдачи, которую мы называем **GreedyChange**, итеративно выбирает монеты самого большого возможного номинала.

```
GreedyChange(money)
    change ← пустой набор монет
    while money > 0
        coin ← самый большой номинал, который меньше или равен money
        добавьте монету номиналом coin к коллекции монет change
        money ← money – coin
    return change
```

ОСТАНОВИТЕСЬ и задумайтесь. Всегда ли **GreedyChange** выдает минимально возможное количество монет?

Допустим, мы хотим обменять 48 денежных единиц (называемых динариями) в Древнем Риме. **GreedyChange** выдает пять монет (48 = 40 + 5 + 1 + 1 + 1), но мы можем дать сдачу, используя только две монеты (48 = 24 + 24). Таким образом, **GreedyChange** не является оптимальным для некоторых номиналов!

ОСТАНОВИТЕСЬ и задумайтесь. Во времена правления Августа номиналы римских монет были изменены на (1600, 800, 400, 200, 100, 50, 25, 2, 1). Почему такие номиналы облегчали жизнь римским кассирам? В более общем смысле что, по вашему мнению, может повлиять на то, когда **GreedyChange** будет производить сдачу наименьшим количеством монет?

Поскольку **GreedyChange** далек от идеала, нам нужно придумать другой метод. Мы можем представить монеты в количестве *d* произвольных номиналов массивом целых чисел:

$$Coins = (coin_1, \ldots, coin_d),$$

где значения $coin_i$ даны в порядке убывания. Мы говорим, что массив из *d* положительных целых чисел ($change_1, \ldots, change_d$) с количеством монет $change_1 + \cdots + change_d$ является **сдачей** целого числа *money* (для номиналов *Coins*), если

$$coin_1 \cdot change_1 + \cdots + coin_d \cdot change_d = money.$$

Например, для римских номиналов *Coins* = (120, 40, 30, 24, 20, 10, 5, 4, 1), оба (0, 1, 0, 0, 0, 0, 0, 1, 0, 3) и (0, 0, 0, 2, 0, 0, 0, 0, 0), сдача *money* = 48.

Мы рассмотрим задачу нахождения минимального количества монет, необходимых для сдачи, вместо фактического производства этих монет. Пусть *MinNumCoins*(*money*) обозначает минимальное количество монет, необходимое для обмена денег для данного набора номиналов (например, для римских номиналов *MinNumCoins*(48) = 2).

Задача сдачи монет: *найти минимальное количество монет, необходимое для сдачи.*

Input: целое число money и массив монет из *d* положительных целых чисел.

Output: минимальное количество монет номиналом *Coins*, составляющее сдачу *money*.

Рекурсивный размен денег

Поскольку жадное решение, которое использовали римские кассиры для решения проблемы сдачи, неверно, мы рассмотрим другой метод. Предположим, вам нужно разменять 76 динариев, а у вас есть монеты только трех самых мелких номиналов: *Coins* = (5, 4, 1). Минимальный набор монет на общую сумму 76 динариев должна быть одной из следующих:

- минимальный набор монет общим номиналом 75 динариев плюс одна монета номиналом 1 динарий;
- минимальный набор монет общим номиналом 72 динария плюс одна монета номиналом 4 динария;
- минимальный набор монет общим номиналом 71 динарий плюс одна монета номиналом 5 динариев.

Для общих номиналов $Coins = (coin_1, \ldots, coin_d)$, *MinNumCoins*(*money*) равно минимуму из *d* чисел:

$$MinNumCoins(money) = \min\begin{cases} MinNumCoins(money - coin_1) + 1 \\ \vdots \\ MinNumCoins(money - coin_d) + 1 \end{cases}.$$

Мы только что создали **рекуррентное соотношение** или уравнение для *MinNumCoins*(*money*) через *MinNumCoins*(*m*) для меньших значений *m*.

Рекуррентное соотношение, воспроизведенное выше, мотивирует следующий рекурсивный алгоритм (т. е. алгоритм, который может вызывать сам себя), решающий задачу изменения, вычисляя *MinNumCoins*(*m*) для все меньших и меньших значений *m*. В этом алгоритме |*Coins*| относится к количеству номиналов в монетах. Смотрите **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Ханойские башни**, если вы еще не сталкивались с рекурсивными алгоритмами.

```
RecursiveChange(money, Coins)
  if money = 0
    return 0
  MinNumCoins ← ∞
  for i ← 0 до |Coins| − 1
    if money ⩾ coin_i
      NumCoins ← RecursiveChange(money − coin_i, Coins)
      if NumCoins + 1 < MinNumCoins
        MinNumCoins ← NumCoins + 1
  return MinNumCoins
```

ОСТАНОВИТЕСЬ и задумайтесь. Разработайте **RecursiveChange** и запустите его на *money* = 76 с *Coins* = (5, 4, 1). Что получается в результате?

RecursiveChange может показаться эффективным, но он совершенно непрактичен, потому что он снова и снова пересчитывает оптимальную комбинацию монет для заданной величины *money*. Например, когда *money* = 76 и *Coins* = (5, 4, 1), *MinNumCoins*(70) вычисляется шесть раз, пять из которых показаны на рис. 5.17. Может показаться, что это не проблема, но *MinNumCoins*(30) будет вычисляться миллиарды раз!

Разменяваем деньги с помощью динамического программирования

Чтобы избежать множества рекурсивных вызовов, необходимых для вычисления *MinNumCoins*(*money*), мы будем использовать стратегию динамического программирования. Было бы неплохо знать все значения *MinNumCoins*(*money* – $coin_i$) к моменту вычисления *MinNumCoins*(*money*)? Вместо трудоемких вы-

зовов **RecursiveChange**(*money* – $coin_i$, *Coins*) мы могли бы просто найти значения *MinNumCoins*(*money* – $coin_i$) в массиве и таким образом вычислить *MinNumCoins*(*money*), используя только сравнения|*Coins*|.

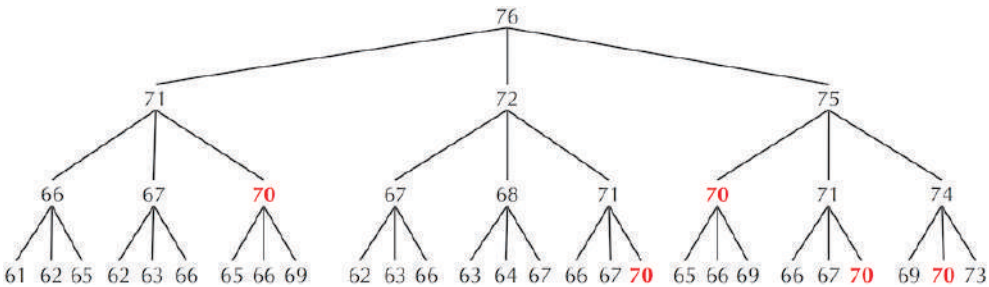


Рис. 5.17 Дерево, иллюстрирующее вычисление *MinNumCoins*(76) для номиналов *money* = (5, 4, 1). Ребра этого дерева представляют собой рекурсивные вызовы **RecursiveChange** для различных входных значений, при этом пять из шести вычислений *MinNumCoins*(70) выделены красным цветом. Когда *MinNumCoins*(70) вычисляется в шестой раз (соответствует пути 76 → 75 → 74 → 73 → 72 → 71 → 70), **RecursiveChange** уже вызывался сотни раз

Ключ к динамическому программированию – сделать шаг, который может показаться нелогичным. Вместо того чтобы вычислять *MinNumCoins*(*m*) для каждого значения *m* от 76 вниз до *m* = 0 с помощью рекурсивных вызовов, мы обратим наше мышление и вычислим *MinNumCoins*(*m*) от *m* = 0 и далее вверх до 76, сохранив все эти значения в массиве, чтобы вычислить *MinNumCoins*(*m*) только один раз для каждого значения *m*. *MinNumCoins*(*m*) по-прежнему вычисляется по тому же рекуррентному соотношению:

$$MinNumCoins(m) = \min \begin{cases} MinNumCoins(m-5)+1 \\ MinNumCoins(m-4)+1. \\ MinNumCoins(m-1)+1 \end{cases}$$

Например, если предположить, что мы уже вычислили *MinNumCoins*(*m*) для *m* < 7, *MinNumCoins*(6) больше на единицу, чем минимум *MinNumCoins*(6 – 5) = 1, MinNumCoins(6 – 4) = 2 и *MinNumCoins*(6 – 1) = 1. Таким образом, *MinNumCoins*(6) равно 1 + 1 = 2. В свою очередь, *MinNumCoins*(7) больше на единицу, чем минимум *MinNumCoins*(7 – 5) = 2, *MinNumCoins*(7 – 4) = 3, а *MinNumCoins*(7 – 1) = 2. В результате *MinNumCoins*(7) равно 2 + 1 = 3 и т. д., в результате чего получается следующая таблица.

m	0	1	2	3	4	5	6	7	8	9	10	11	12
MINNUMCOINS(***m***)	0	1	2	3	1	1	2	3	2	2	2	3	3

Рис. 5.18 *MinNumCoins*(*m*) для значений *m* от 0 до 12

Упражнение. Используйте динамическое программирование, чтобы заполнить десять значений *MinNumCoins*(*m*) из таблицы выше (т. е. для 13 ≤ m ≤ 22). Поместите все значения в одну строку.

Обратите внимание, что *MinNumCoins*(2) используется при вычислении как *MinNumCoins*(6), так и *MinNumCoins*(7), но вместо истощения вычислительных ресурсов из-за необходимости оба раза вычислять это значение с нуля мы просто сверяемся с предварительно вычисленным значением в массиве. Следующий алгоритм динамического программирования вычисляет *MinNumCoins*(*money*) с временем выполнения $O(money \cdot |Coins|)$.

```
DPChange(money, Coins)
  MinNumCoins(0) ← 0
  for m ← 1 до money
    MinNumCoins(m) ← ∞
      for i ← 0 до |Coins| − 1
        if m ⩾ coin_i
          if MinNumCoins(m − coin_i) + 1 < MinNumCoins(m)
            MinNumCoins(m) ← MinNumCoins(m − coin_i) + 1
  output MinNumCoins(money)
```

ОСТАНОВИТЕСЬ и задумайтесь Рассмотрите следующие модификации **DPChange**.

1. Если *money* = 10^9, **DPChange** требует огромного массива размером 10^9. Измените алгоритм **DPChange** так, чтобы требуемый размер массива не превышал значение наибольшего номинала монеты.
2. Напомним, что нашей первоначальной целью было набрать сдачу, а не просто вычислить *MinNumCoins*(*money*). **Измените DPChange**, чтобы он не только вычислял минимальное количество монет, но и выдавал номиналы этих монет.

Новый взгляд на туристическую задачу Манхэттена

Теперь вы должны быть готовы реализовать алгоритм решения проблемы туриста на Манхэттене. Следующий псевдокод вычисляет длину самого длинного пути к узлу (*i, j*) в прямоугольной сетке и основан на наблюдении, что единственный способ достичь узла (*i, j*) в туристической задаче Манхэттена – либо двигаться на юг (↓) от (*i* − 1, *j*), либо к востоку (→) от (*i*, *j* − 1).

```
SouthOrEast(i, j)
  if i = 0 и j = 0
    return 0
  x ← −∞, y ← −∞
  if i > 0
    x ← SouthOrEast(i − 1, j) + вес вертикального ребра в (i, j)
  if j > 0
    y ← SouthOrEast(i, j − 1) + вес горизонтального ребра в (i, j)
  return max{x, y}
```

ОСТАНОВИТЕСЬ и задумайтесь. Сколько раз **SouthOrEast**(3, 2) вызывается при вычислении **SouthOrEast**(9, 7)?

Аналогично **RecursiveChange**, **SouthOrEast** страдает от огромного количества рекурсивных вызовов, и нам необходимо переделать этот алгоритм с помощью динамического программирования. Помните, как **DPChange** работал от небольших обращений *вверх*? Чтобы найти длину самого длинного пути от источника (0, 0) до стока (*n*, *m*), мы сначала найдем длины самых длинных путей от источника ко всем узлам (*i*, *j*) в сетке, медленно расширяющейся наружу из источника. На первый взгляд может показаться, что мы придумали себе дополнительную работу, решив $n \cdot m$ разных задач вместо одной. Тем не менее **SouthOrEast** также решает все эти более мелкие проблемы, так же как **RecursiveChange** и **DPChange** вычисляют *MinNumCoins*(*m*) для всех значений $m < money$. Хитрость динамического программирования заключается в том, чтобы решать небольшие задачи один раз, а не миллиарды раз.

В дальнейшем мы будем обозначать длину наибольшего пути из (0, 0) в (*i*, *j*) как $s_{i,j}$. Вычислить $s_{0,j}$ (для $0 \le j \le m$) несложно, так как мы можем достичь только (0, *j*), двигаясь вправо (→) и не имея никакой гибкости в выборе пути. Таким образом, $s_{0,j}$ есть сумма весов первых *j* горизонтальных ребер, выходящих из источника. Точно так же $s_{i,0}$ представляет собой сумму весов первых *i* вертикальных ребер от источника (рис. 5.19).

Для $i > 0$ и $j > 0$ единственный способ достичь узла (*i*, *j*) – это двигаться вниз от узла (*i* – 1, *j*) или двигаться вправо от узла (*i*, *j* – *1*). Таким образом, $s_{i,j}$ можно вычислить как максимальное из двух значений:

- $s_{i-1,j}$ + вес вертикального ребра от (*i* – 1, *j*) до (*i*, *j*);
- $s_{i,j-1}$ + вес горизонтального ребра от (*i*, *j* – 1) до (*i*, *j*).

Теперь, когда мы вычислили $s_{0,1}$ и $s_{1,0}$, можем вычислить $s_{1,1}$. Вы можете прийти к (1, 1), двигаясь вниз от (0, 1) или вправо от (1, 0). Следовательно, $s_{1,1}$ является максимальным из двух значений:

- $s_{0,1}$ + вес вертикального ребра от (0, 1) до (1, 1) = 3 + 0 = 3;
- $s_{1,0}$ + вес горизонтального ребра от (1, 0) до (1, 1) = 1 + 3 = 4.

Поскольку наша цель – найти самый длинный путь из (0, 0) в (1, 1), мы делаем вывод, что $s_{1,1} = 4$. Поскольку мы выбрали горизонтальное ребро из (1, 0) в (1, 1), самый длинный путь через (1, 1) должен использовать это ребро, которое мы выделили на рис. 5.20.

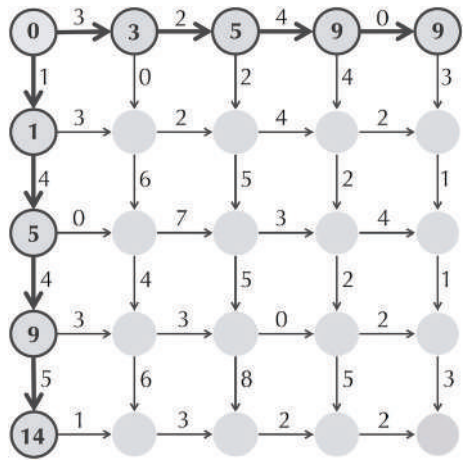


Рис. 5.19 Вычислить $s_{i,0}$ и $s_{0,j}$ нетрудно, потому что существует только один путь от источника к $(i, 0)$ и только один путь от источника к $(0, j)$

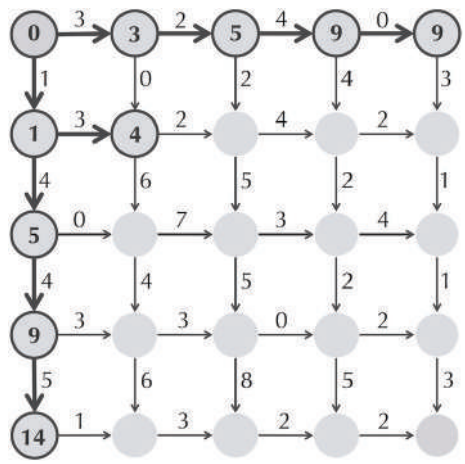


Рис. 5.20 При вычислении $s_{1,1}$ используется горизонтальное ребро из (1, 0), которое выделено

Аналогичная логика позволяет нам вычислить остальные значения в столбце 1; для каждого $s_{i,1}$ мы выделяем выбранное нами ребро, ведущее в $(i, 1)$, как показано на рисунке ниже.

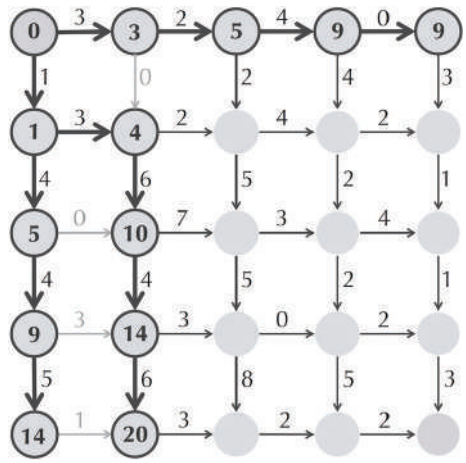


Рис. 5.21 Вычисление всех значений $s_{i,1}$ в столбце 1

Упражнение. Вычислите все пять значений $s_{i,2}$ в столбце 2. Выдайте ответ в виде списка целых чисел в одной строке.

Продолжая столбец за столбцом, как показано на рисунке внизу, мы можем вычислить значения $s_{i,j}$ за один проход графа, в конечном итоге вычислив $s_{4,4} = 34$.

Для каждого узла (i, j) мы выделим ребро, ведущее в (i, j), которое мы использовали для вычисления $s_{i,j}$. Однако обратите внимание, что у нас ничья, когда мы вычисляем $s_{3,3}$.

$$s_{3,3} = \begin{cases} s_{2,3} + \text{weight of vertical edge from } (2,3) \text{ to } (3,3) & = 20 + 2 = 22 \\ s_{3,2} + \text{weight of horizontal edge from } (3,2) \text{ to } (3,3) & = 22 + 0 = 22 \end{cases}.$$

Чтобы достичь (3, 3), мы могли бы использовать *либо* горизонтальное, либо вертикальное входящее ребро, поэтому мы выделим оба этих ребра на завершенном графе на рис. 5.22.

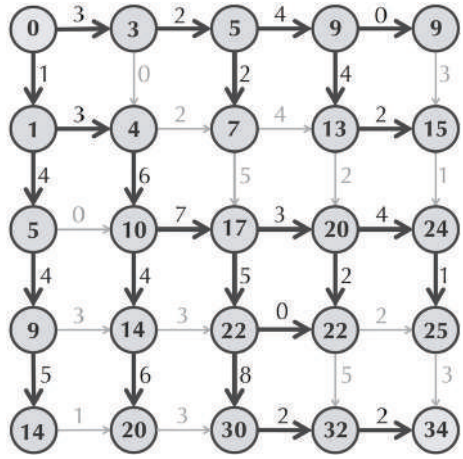


Рис. 5.22 Граф, отображающий все значения $s_{i,j}$

ОСТАНОВИТЕСЬ и задумайтесь. До сих пор мы только обсуждали, как найти длину самого длинного пути. Как можно использовать выделенные ребра для реконструкции самого длинного пути?

Теперь у нас есть набросок алгоритма динамического программирования для нахождения длины самого длинного пути в туристической задаче Манхэттена, который называется **ManhattanTourist**. В следующем псевдокоде $down_{i,j}$ и $right_{i,j}$ являются соответствующими весами вертикального и горизонтального ребер, входящих в узел (i, j). Обозначим матрицы, содержащие $(down_{i,j})$ и $(right_{i,j})$, как *Down* и *Right* соответственно.

```
ManhattanTourist(n, m, Down, Right)
  s_{0,0} ← 0
  for i ← 1 до n
    s_{i,0} ← s_{i-1,0} + down_{i-1,0}
  for j ← 1 до m
    s_{0,j} ← s_{0,j-1} + right_{0,j-1}
  for i ← 1 до n
    for j ← 1 до m
      s_{i,j} ← max{s_{i-1,j} + down_{i-1,j}, s_{i,j-1} + right_{i,j-1}}
  return s_{n,m}
```

Упражнение. Модифицируйте **ManhattanTourist**, чтобы найти длину самого длинного пути от истока до стока на графе, показанном ниже.

Подсказка: вы можете попробовать выполнить это упражнение вручную.

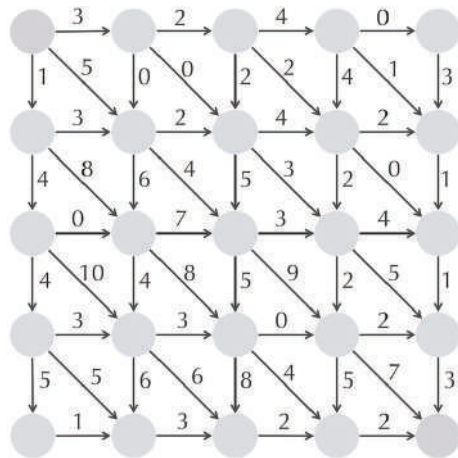


Рис. 5.23 Граф с диагональными ребрами, построенный для воображаемого города

От Манхэттена к произвольному DAG[1]

Выравнивание последовательности как построение графа в стиле Манхэттена

Увидев, как динамическое программирование решило туристическую задачу Манхеттена, вы должны быть готовы адаптировать **ManhattanTourist** для выравнивания графов с диагональными ребрами. Вспомните рисунок, воспроизведенный ниже, на котором мы смоделировали задачу о самой длинной общей подпоследовательности как поиск самого длинного пути в графе выравнивания «город», все «достопримечательности» (совпадения) которого лежат на диагональных ребрах с весом 1.

[1] DAG – это ориентированный ациклический граф. – *Прим. ред.*

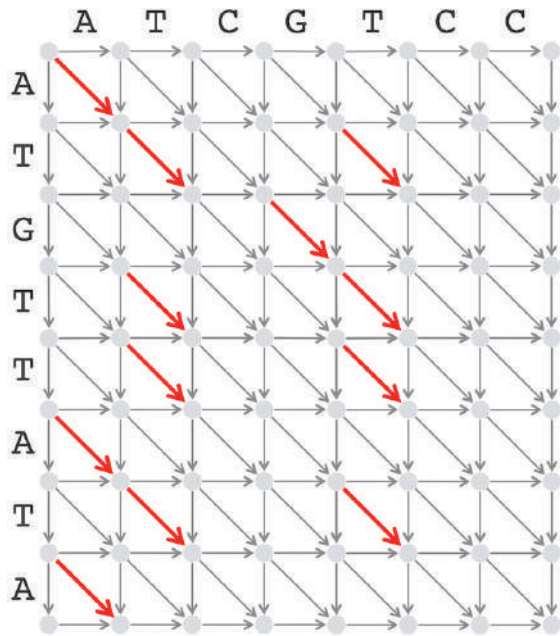


Рис. 5.24 Граф с диагональными ребрами

Вероятно, вы можете самостоятельно вычислить рекуррентное соотношение для графа выравнивания, но представьте на секунду, что вы еще не знаете, что LCS (Самая длинная общей подпоследовательность. – *Прим. ред.*) может быть представлен длиннейшим путем в графе выравнивания. Как поясняется в статье **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Поиск самой длинной общей подпоследовательности без строительства города**, нам не нужно строить город, подобный Манхэттену, чтобы вычислить длину LCS. Однако аргументы, необходимые для этого, утомительны.

Что еще более важно, различные приложения выравнивания намного сложнее, чем задача самой длинной общей последовательности, и требуют создания графа DAG с правильно выбранными весами ребер для моделирования специфики биологической проблемы. Вместо того чтобы рассматривать каждое последующее приложение выравнивания как новую пугающую задачу, мы хотели бы вооружить вас общим алгоритмом динамического программирования, который найдет самый длинный путь в любом графе DAG. Более того, многие задачи биоинформатики не имеют ничего общего с выравниванием, но их также можно решить как приложения задачи о самом длинном пути в задаче DAG.

Динамическое программирование в произвольном графе DAG

Для узла b в DAG определим s_b как длину самого длинного пути от источника к b. Мы называем узел a **предшественником** узла b, если в DAG есть ребро, со-

единяющее a с b; степень вхождения узла равна количеству его предшественников. **Счет** s_b узла b со степенью вхождения k вычисляется как максимум k слагаемых:

$$s_b = \max_{\text{all predecessors } a \text{ of node } b} \{s_a + \text{weight of edge from } a \text{ to } b\}.$$

Например, на приведенном ниже графе узел (1, 1) имеет трех предшественников. Вы можете прийти к (1, 1), двигаясь вправо от (1, 0), вниз от (0, 1) или по диагонали от (0, 0). Предполагая, что мы уже вычислили $s_{0,0}$, $s_{0,1}$ и $s_{1,0}$, мы можем вычислить $s_{1,1}$ как максимальное из трех значений.

$$s_b = \begin{cases} s_{0,1} + \text{weight of edge} \downarrow \text{connecting } (0,1) \text{ to } (1,1) = 3+0 = 3 \\ s_{1,0} + \text{weight of edge} \rightarrow \text{connecting } (1,0) \text{ to } (1,1) = 1+3 = 4 \\ s_{0,0} + \text{weight of edge} \searrow \text{connecting } (0,0) \text{ to } (1,1) = 0+5 = 5 \end{cases}.$$

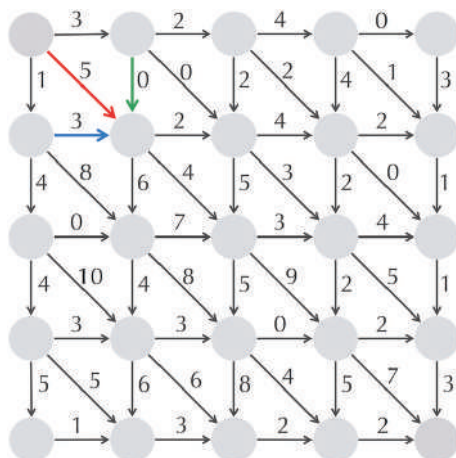


Рис. 5.25 У приведенного выше графа DAG (1, 1) есть три предшественника ((0, 0), (0, 1) и (1, 0)), которые используются при вычислении $s_{1,1}$

Аналогичный аргумент можно применить к графу выравнивания (ниже), чтобы вычислить длину LCS между последовательностями v и ω. Поскольку в этом случае все ребра имеют вес 0, кроме диагональных ребер веса 1, представляющих совпадения ($v_i = \omega_j$), мы получаем следующую рекуррентность для вычисления длины LCS.

$$s_{i,j} = \begin{cases} s_{i-1,j} + \text{weight of edge} \downarrow \text{between } (i-1, j) \text{ and } (i, j) \\ s_{i,j-1} + \text{weight of edge} \rightarrow \text{between } (i, j-1) \text{ and } (i, j) \\ s_{i-1,j-1} + \text{weight of edge} \searrow \text{between } (i-1, j-1) \text{ and } (i, j) \end{cases}.$$

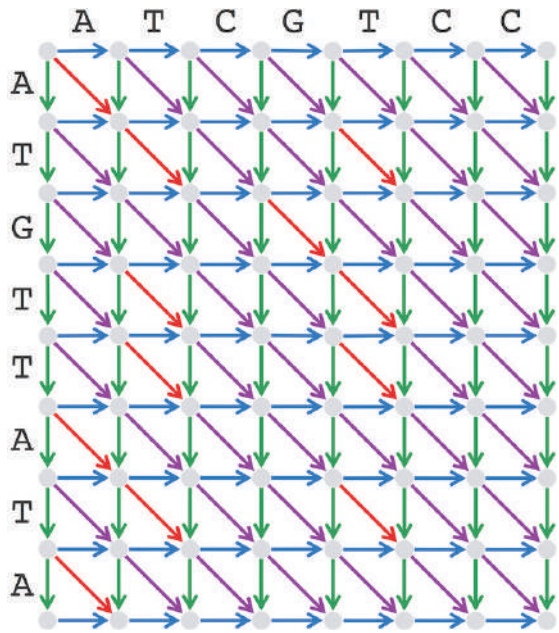


Рис. 5.26 Граф выравнивания

ОСТАНОВИТЕСЬ и задумайтесь. Приведенное выше повторение не включает ребра несовпадения. Почему это не является проблемой?

Аналогичный метод можно разработать для поиска самого длинного пути в любом графе DAG, как это предлагается в следующем упражнении.

Упражнение. Какова длина самого длинного пути между синим и красным узлами в DAG, показанном на рис. 5.27?

Топологические порядки

Не волнуйтесь, если вам трудно выполнить последнее упражнение. Загвоздка в использовании динамического программирования для нахождения длины самого длинного пути в DAG заключается в том, что мы должны решить, в каком порядке посещать узлы при вычислении значений s_b в соответствии с рекуррентностью.

$$s_b = \max_{\text{all predecessors } a \text{ of node } b} \{s_a + \text{weight of edge from } a \text{ to } b\}.$$

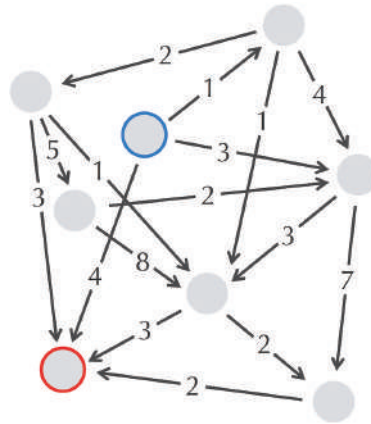


Рис. 5.27 Ориентированный ациклический граф DAG

Такой порядок узлов важен, поскольку к тому времени, когда мы достигнем узла b, значения s_a для всех его предшественников уже должны быть вычислены. Нам удалось скрыть эту задачу для прямоугольных сеток, потому что порядок, в котором мы вычисляли $s_{i,j}$, гарантировал, что мы никогда не рассмотрим узел, пока не посетим всех его предшественников.

Чтобы проиллюстрировать важность посещения узлов в правильном порядке, рассмотрим DAG, показанный ниже, который соответствует «задаче с одеванием». Как бы вы расположили узлы этого графа, чтобы не надеть сапоги раньше колготок?

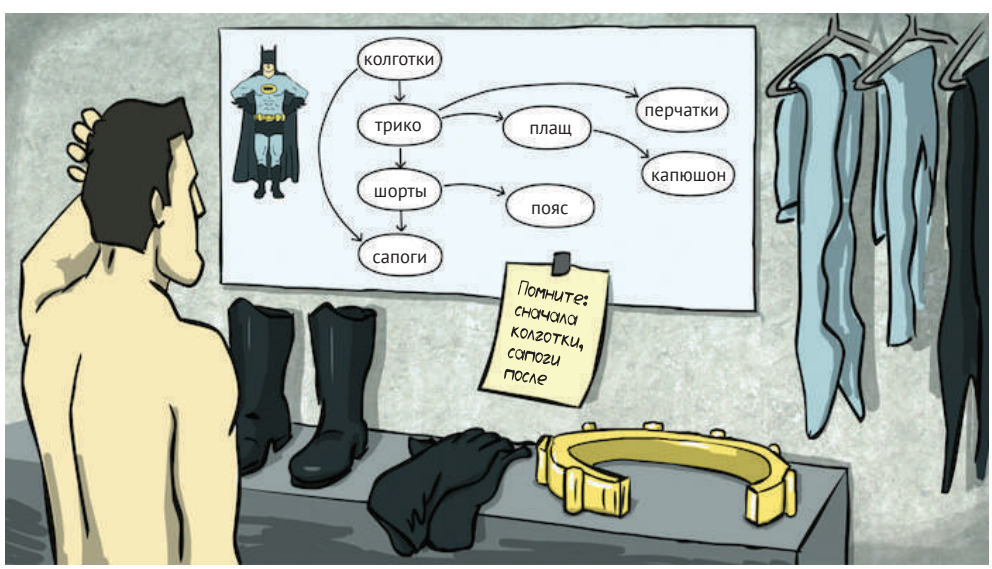


Рис. 5.28 Задача порядка одевания

Чтобы решить задачу с одеванием, нам нужно расположить узлы в графе DAG вдоль линии так, чтобы каждое направленное ребро соединяло узел с узлом справа от него (рисунок ниже). Чтобы одеться без ошибок, вы можете просто посещать узлы слева направо. Чтобы найти самый длинный путь в произвольной DAG, нам сначала нужно упорядочить узлы DAG так, чтобы каждый узел попадал после всех своих предшественников. Формально порядок узлов $(a_1, \ldots, a_k)$ в DAG называется **топологическим порядком**, если каждое ребро (a_i, a_j) DAG соединяет узел с меньшим номером с узлом с большим номером, т. е. $i < j$.

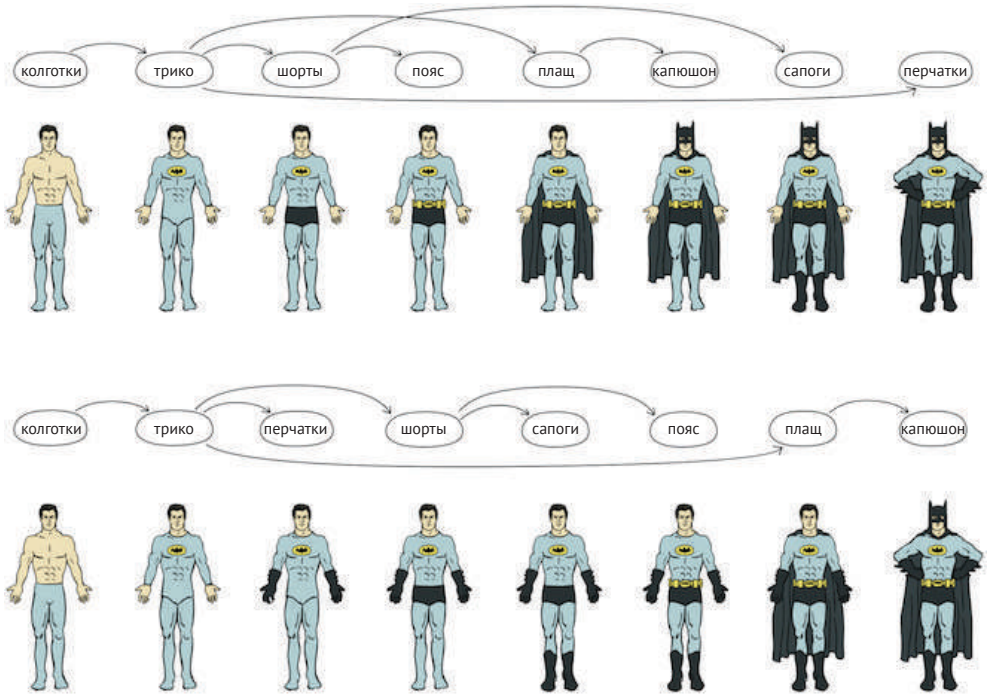


Рис. 5.29 Два разных топологических порядка задачи одевания

Упражнение. Сколько топологических порядков имеет описанная выше задача одевания?

ManhattanTourist может найти самый длинный путь в прямоугольной сетке, потому что его псевдокод (воспроизведенный внизу) неявно упорядочивает узлы в соответствии с топологическим порядком «строка за строкой», показанным на рисунке ниже.

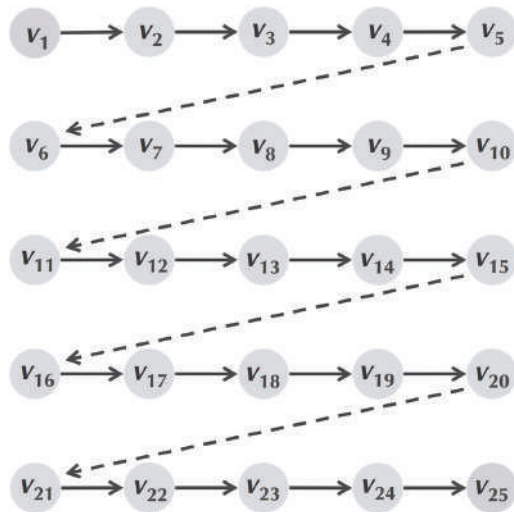


Рис. 5.30 Упорядочение узлов графа по строкам

```
ManhattanTourist(n, m, Down, Right)
  s_{0,0} ← 0
  for i ← 1 до n
    s_{i,0} ← s_{i-1,0} + down_{i,0}
  for j ← 1 до m
    s_{0,j} ← s_{0,j-1} + right_{0,j}
  for i ← 1 до n
    for j ← 1 до m
      s_{i,j} ← max{s_{i-1,j} + down_{i,j}, s_{i,j-1} + right_{i,j}}
  return s_{n,m}
```

Топологическое упорядочение «столбец за столбцом», показанное ниже, дает еще одно топологическое упорядочение прямоугольной сетки.

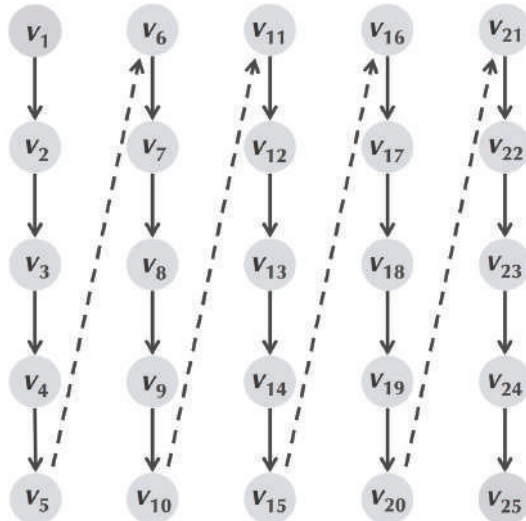


Рис. 5.31 Упорядочение узлов графа по столбцам

ОСТАНОВИТЕСЬ и задумайтесь. Перепишите псевдокод **ManhattanTourist** на основе дополнительного топологического порядка, показанного ниже.

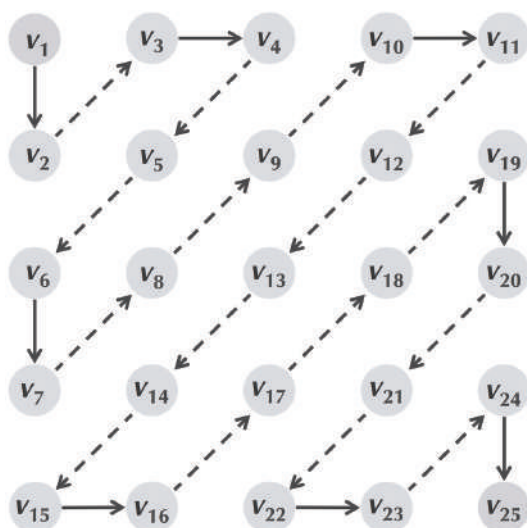


Рис.5.32 **Еще** один топологический порядок

Можно доказать, что любой DAG имеет топологический порядок и что этот топологический порядок может быть построен за время, пропорциональное количеству ребер в графе (подробнее см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Построение топологического порядка**). Получив топологический порядок, мы можем вычислить длину самого длинного пути от источника к стоку, посетив узлы DAG в порядке, определяемом топологическим порядком, который достигается с помощью следующего алгоритма. Для простоты мы предполагаем, что исходный узел является единственным узлом со степенью вхождения 0 в *Graph*.

```
LongestPath(Graph, source, sink)
    for каждого узла b в Graph
        s_b ← −∞
    s_source ← 0
    топологически упорядочить Graph
    for каждого узла b в Graph (следуя топологическому порядку)
        s_b ← max_all предшественники a узла b {s_a + вес ребра от a до b}
    return s_sink
```

Поскольку каждое ребро участвует только в одном рекуррентном возврате, время выполнения **LongestPath** пропорционально количеству ребер в графе *Graph*.

Теперь мы можем эффективно вычислить длину самого длинного пути в произвольном DAG, но пока не знаем, как преобразовать **LongestPath** в алгоритм, который будет *строить* этот самый длинный путь. В следующем разделе мы воспользуемся задачей самой длинной общей подпоследовательности, чтобы объяснить, как построить самый длинный путь в графе DAG.

Возвращаясь к графу выравнивания

Напомним, что ранее мы выделяли каждое ребро, выбранное **ManhattanTourist**, как показано на рис. 5.33.

Чтобы сформировать самый длинный путь, нам просто нужно найти путь от истока к стоку, образованный выделенными ребрами (таких путей может существовать более одного). Однако, если бы мы шли от истока к стоку вдоль выделенных ребер, мы могли бы зайти в тупик, например в узел (1, 2). Напротив, каждый путь от стока вернет нас к источнику, если мы вернемся в направлении, противоположном каждому выделенному ребру, как показано на рис. 5.34.

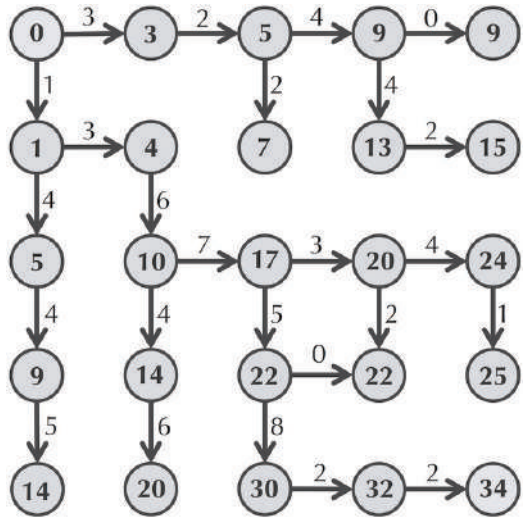


Рис. 5.33 Выделенные ребра графа

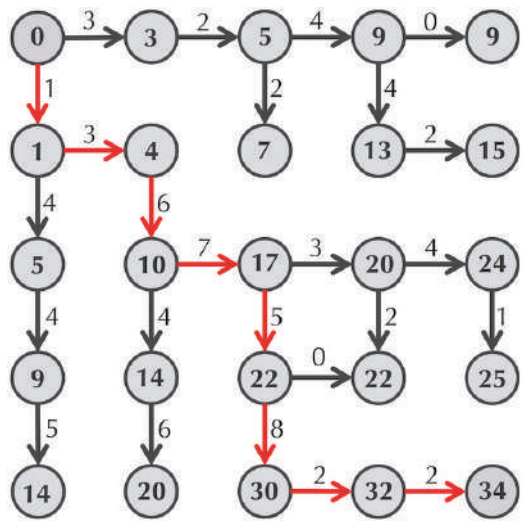


Рис. 5.34 Следование ребрам от стока обратно к истоку дает красный путь в DAG выше, который выделяет самый длинный путь от истока к стоку

Мы можем использовать эту идею поиска с возвратом для построения LCS строк v и w. Мы знаем, что если мы присвоим вес 1 ребрам в *AlignmentGraph*(v, ω), соответствующим совпадениям, и присвоим вес 0 всем остальным ребрам, то $s_{|v|, |\omega|}$ дает длину LCS. Следующий алгоритм сохраняет запись о том, какое ребро использовалось для вычисления каждого значения $s_{i, j}$, используя **указа-**

тели возврата, которые принимают одно из трех значений ↓, → или ↘. Указатели возврата хранятся в матрице *Backtrack*.

```
LCSBackTrack(v, ω)
   for i ← 0 до |v|
      s_i, 0 ← 0
   for j ← 0 до|ω|
      s_0,j ← 0
   for i ← 1 до |v|
      for j ← 1 to |ω|
         match ← 0
         if v_i - 1 = ω_j - 1
            match ← 1
         s_i,j ← max{s_i - 1, j , s_i, j - 1 , s_i - 1, j  - 1 + match }
         if s_i,j = s_i - 1, j
            Backtrack_i, j ← «↓»
         else if s_i, j = s_i, j - 1
            Backtrack_i, j ← «→»
         else if s_i, j = s_i - 1, j - 1 + match
            Backtracki_, j ← «↘»
   return Backtrack
```

ОСТАНОВИТЕСЬ и задумайтесь. Как изменение порядка этих трех операторов «**if**» повлияет на вычисление *Backtrack*?

Теперь нам нужно найти путь от источника к стоку, образованный выделенными ребрами. Следующий алгоритм решает задачу самой длинной общей подпоследовательности, используя информацию в *Backtrack*. **OutputLCS**(*Backtrack, v, i, j*) выводит LCS между *i*-префиксом *v* и *j*-префиксом ω. Начальным вызовом, который выводит LCS *v* и ω, является **OutputLCS**(*Backtrack, v,* |*v*|, |ω|).

```
OutputLCS(backtrack, v, i, j)
   if i = 0 или j = 0
      return «»
   if backtrack_i, j = «↓»
      return OutputLCS(backtrack, v, i - 1, j)
   else if backtrack_i, j = «→»
      return OutputLCS(backtrack, v, i, j - 1)
   else
      return OutputLCS(backtrack, v, i - 1, j - 1) + v_i
```

ОСТАНОВИТЕСЬ и задумайтесь. **OutputLCS** – это рекурсивный алгоритм, но он эффективен. Чем он отличается от неэффективных рекурсивных алгоритмов внесения изменений и поиска самого длинного пути в DAG?

Метод поиска с возвратом можно обобщить для построения самого длинного пути в любом DAG. Всякий раз, когда мы вычисляем s_b как

$$s_b = \max_{\text{all predecessors } a \text{ of node } b} \{s_a + \text{weight of edge from } a \text{ to } b\},$$

нам просто нужно сохранить предшественника b, который использовался при вычислении s_b, чтобы мы могли вернуться позже. Теперь вы готовы использовать поиск с возвратом, чтобы найти самый длинный путь в произвольной DAG.

Упражнение. В настоящее время **OutputLCS** находит одну LCS из двух строк. Измените **OutputLCS** (и **LCSBacktrack**), чтобы найти каждую LCS из двух строк.

Считаем выравнивания

Что не так с моделью LCS?

Вспомните выравнивание Марахиела кодирования A-доменов для Asp и Orn, которое имело **19** + **10** совпадений:

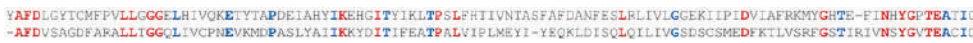


Рис. 5.35 Выравнивание Марахиела кодирования A-доменов для Asp и Orn

Нетрудно построить выравнивание, имеющее еще больше совпадений за счет введения большего количества инделов. Однако чем больше инделов мы добавляем, тем менее биологически релевантным становится выравнивание, поскольку оно все больше и больше расходится с биологически правильным выравниванием, обнаруженным Марахиелом. Ниже приведено выравнивание с максимальным количеством совпадений, представляющее LCS длиной **19** + **8** + **19** = 46 (зеленые символы представляют новые совпадения). Это выравнивание настолько длинное, что мы не можем уместить его в одной строке.

```
YAFDL--G-YTCMFP--VLL-GGGELHIV---Q-K-E--T-YTAPDEIAHYIK--EHGITYI---KLTPSL-FHT
-AFDVSAGD----FARA-LLTGG-QL-IVCPNEVKMDPASLY-A---I---IKKYD--IT-IFEA--TPALV---

IVNTASFAFDANFE-----S-LR-LIVLGG-----EKIIPIDVIAFRK-M---YGHTEFI---NHYGPTEATIGA
IPLMEYIY-----EQKLDISQLQILIV-GSDSCSME-----D---F-KTLVSRFGST--IRIVNSYGVTEACIDS
```

Рис. 5.36 Выравнивание с максимальным количеством совпадений

ОСТАНОВИТЕСЬ и задумайтесь. Если бы Марахиел построил приведенное выше выравнивание, смог бы он сделать вывод о восьми аминокислотных сигнатурах нерибосомного кода?

Ниже мы выделяем фиолетовым аминокислоты, представляющие нерибосомные сигнатуры. Хотя эти сигнатуры сгруппированы в восемь консервативных столбцов в выравнивании Марахиела с начала главы, только пять из этих столбцов «выжили» в выравнивании LCS, что делает невозможным вывод о нерибосомных сигнатурах:

```
YAFDL--G-YTCMFP--VLL-GGGELHIV---Q-K-E--T-YTAPDEIAHYIK--EHGITYI---KLTPSL-FHT
-AFDVSAGD----FARA-LLTGG-QL-IVCPNEVKMDPASLY-A---I---IKKYD--IT-IFEA--TPALV---

IVNTASFAFDANFE-----S-LR-LIVLGG-----EKIIPIDVIAFRK-M---YGHTEFI---NHYGPTEATIGA
IPLMEYIY-----EQKLDISQLQILIV-GSDSCSME-----D---F-KTLVSRFGST--IRIVNSYGVTEACIDS
```

Рис. 5.37 Выделены аминокислоты, представляющие нерибосомные сигнатуры

Незначительные совпадения, скрывающие реальный эволюционный сценарий, появились потому, что ничто не мешало нам вводить чрезмерное количество инделов при построении LCS. Вспоминая нашу первоначальную игру выравнивания, в которой мы вознаграждали за совпадающие символы, нам нужен какой-то способ *штрафовать* за инделы и несовпадения. Во-первых, давайте обработаем вставки. Предположим, что в дополнение к присвоению совпадениям премии в размере +1 мы решили оценить каждую вставку штрафом в –4. Выравнивание A-доменов с наивысшим баллом приближается к биологически правильному выравниванию, при этом шесть столбцов соответствуют правильно выровненным сигнатурам.

```
YAFDLGYTCMFP-VLL-GGGELHIV-QKETYTAPDEI-AHYIKEHGITYI-KLTPSLFHTIVNTASFAFDANFE
-AFDVS-AGDFARALLTGG-QL-IVCPNEVKMDPASLYA-IIKKYDIT-IFEATPAL--VIPLME-YIYEQKLD

-S-LR-LIVLGGEKIIPIDVIAFRKM---YGHTE-FINHYGPTEATIGA
ISQLQILIV-GSDSC-SME--DFKTLVSRFGSTIRIVNSYGVTEACIDS
```

Рис. 5.38 Шесть столбцов соответствуют правильно выровненным сигнатурам

Матрицы счета

Чтобы обобщить модель счета выравнивания, мы по-прежнему присуждаем +1 за **совпадения**, но также штрафуем **несовпадения** некоторой положительной константой μ (**штраф за несовпадение**) и **инделы** некоторой положительной константой σ (**штраф за индел**). В результате счет выравнивания равен следующему выражению:

$$\#\text{ совпадения} - \mu \cdot \#\text{ несовпадения} - \sigma \cdot \#\text{ вставки}.$$

Например, при параметрах $\mu = 1$ и $\sigma = 2$ следующему выравниванию будет присвоено значение −4.

A	T	–	G	T	T	A	T	A
A	T	C	G	T	–	C	–	C
+1	+1	-2	+1	+1	-2	-1	-2	-1

Рис. 5.39 При параметрах $\mu = 1$ и $\sigma = 2$ выравниванию присвоено значение −4

Биологи дополнительно уточнили эту функцию оценки, чтобы учесть тот факт, что некоторые мутации могут быть более вероятными, чем другие, что требует несовпадений и штрафных очков, которые различаются в зависимости от конкретных задействованных символов. Мы расширим k-буквенный алфавит, включив в него символ пробела, а затем построим **матрицу счета** размерностью $(k+1)\times(k+1)$, содержащую счет выравнивания каждой пары символов. Матрица счета для сравнения последовательностей ДНК ($k = 4$), когда все несовпадения штрафуются μ, а все вставки штрафуются σ, показана ниже.

	A	C	G	T	–
A	+1	$-\mu$	$-\mu$	$-\mu$	$-\sigma$
C	$-\mu$	+1	$-\mu$	$-\mu$	$-\sigma$
G	$-\mu$	$-\mu$	+1	$-\mu$	$-\sigma$
T	$-\mu$	$-\mu$	$-\mu$	+1	$-\sigma$
–	$-\sigma$	$-\sigma$	$-\sigma$	$-\sigma$	

Рис. 5.40 Матрица счета для сравнения последовательностей ДНК

Хотя матрицы счета для сравнения последовательностей ДНК обычно определяются только параметрами μ и σ, матрицы счета для сравнения последовательностей белков по-разному взвешивают разные мутации и становятся весьма сложными. Один из примеров, матрица счета PAM_{250}, показан ниже (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Матрицы счета PAM**).

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y	–
A	2	-2	0	0	-3	1	-1	-1	-1	-2	-1	0	1	0	-2	1	1	0	-6	-3	-8
C	-2	12	-5	-5	-4	-3	-3	-2	-5	-6	-5	-4	-3	-5	-4	0	-2	-2	-8	0	-8
D	0	-5	4	3	-6	1	1	-2	0	-4	-3	2	-1	2	-1	0	0	-2	-7	-4	-8
E	0	-5	3	4	-5	0	1	-2	0	-3	-2	1	-1	2	-1	0	0	-2	-7	-4	-8
F	-3	-4	-6	-5	9	-5	-2	1	-5	2	0	-3	-5	-5	-4	-3	-3	-1	0	7	-8
G	1	-3	1	0	-5	5	-2	-3	-2	-4	-3	0	0	-1	-3	1	0	-1	-7	-5	-8
H	-1	-3	1	1	-2	-2	6	-2	0	-2	-2	2	0	3	2	-1	-1	-2	-3	0	-8
I	-1	-2	-2	-2	1	-3	-2	5	-2	2	2	-2	-2	-2	-2	-1	0	4	-5	-1	-8
K	-1	-5	0	0	-5	-2	0	-2	5	-3	0	1	-1	1	3	0	0	-2	-3	-4	-8
L	-2	-6	-4	-3	2	-4	-2	2	-3	6	4	-3	-3	-2	-3	-3	-2	2	-2	-1	-8
M	-1	-5	-3	-2	0	-3	-2	2	0	4	6	-2	-2	-1	0	-2	-1	2	-4	-2	-8
N	0	-4	2	1	-3	0	2	-2	1	-3	-2	2	0	1	0	1	0	-2	-4	-2	-8
P	1	-3	-1	-1	-5	0	0	-2	-1	-3	-2	0	6	0	0	1	0	-1	-6	-5	-8
Q	0	-5	2	2	-5	-1	3	-2	1	-2	-1	1	0	4	1	-1	-1	-2	-5	-4	-8
R	-2	-4	-1	-1	-4	-3	2	-2	3	-3	0	0	0	1	6	0	-1	-2	2	-4	-8
S	1	0	0	0	-3	1	-1	-1	0	-3	-2	1	1	-1	0	2	1	-1	-2	-3	-8
T	1	-2	0	0	-3	0	-1	0	0	-2	-1	0	0	-1	-1	1	3	0	-5	-3	-8
V	0	-2	-2	-2	-1	-1	-2	4	-2	2	2	-2	-1	-2	-2	-1	0	4	-6	-2	-8
W	-6	-8	-7	-7	0	-7	-3	-5	-3	-2	-4	-4	-6	-5	2	-2	-5	-6	17	0	-8
Y	-3	0	-4	-4	7	-5	0	-1	-4	-1	-2	-2	-5	-4	-4	-3	-3	-2	0	10	-8
–	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	

Рис. 5.41 Матрица счета PAM_{250} для выравнивания белков. Здесь штраф за индел был установлен на 8

От глобального к локальному выравниванию

Глобальное выравнивание

Теперь вы должны быть готовы изменить граф выравнивания, чтобы решить обобщенную форму задачи выравнивания, которая использует матрицу счета в качестве входных данных.

Глобальная задача выравнивания: *найдите выравнивание двух строк с наивысшим счетом, как определено матрицей счета.*

Input: две строки и матрица счета *Score*.

Output: выравнивание строк, счет выравнивания которых (как определено *Score*) максимален среди всех выравниваний строк.

Чтобы решить глобальную задачу выравнивания, мы по-прежнему должны найти самый длинный путь в графе выравнивания после обновления весов ребер, чтобы они отражали значения в матрице счета. Вспоминая, что **делеции** соответствуют вертикальным ребрам (↓), вставки – горизонтальным ребрам (→), а **совпадения**/**несовпадения** – диагональным ребрам (↘/↘), мы получаем следующую рекуррентность для $s_{i,j}$, длины самого длинного пути от (0, 0) до (*i*, *j*):

$$s_{i,j} = \max \begin{cases} s_{i-1,j} & + Score(v_i, -) \\ s_{i,j-1} & + Score(-, w_j). \\ s_{i-1,j-1} & + Score(v_i, w_j) \end{cases}$$

Когда награда за совпадение равна +1, штраф за несовпадение равен μ, а штраф за вставку равен σ, рекурренцию выравнивания можно записать следующим образом:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} & -\sigma \\ s_{i,j-1} & -\sigma \\ s_{i-1,j-1} & +1 \quad \text{если } v_i = w_j \\ s_{i-1,j-1} & -\mu \quad \text{если } v_i \neq w_j \end{cases}$$

На рис. 5.42 показан пример графа выравнивания.

Ограничения глобального выравнивания

Анализ **генов гомеобокса** предлагает пример проблемы, для которой глобальное выравнивание может не выявить биологически значимых сходств. Эти гены регулируют эмбриональное развитие и присутствуют у большого количества видов, от мух до человека. Гены гомеобокса длинные, и они сильно различаются между видами, но область длиной примерно 60 аминокислот в каждом гене, называемая **гомеодоменом**, высоко консервативна. Например, рассмотрим гомеодомены мыши и человека, представленные на рис. 5.43.

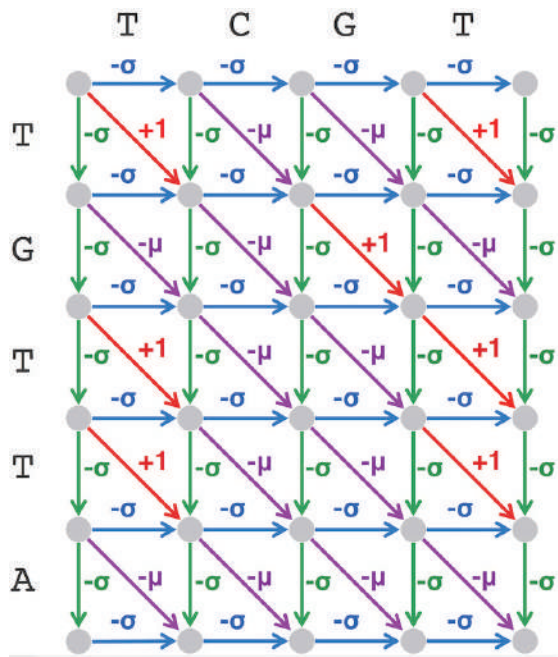


Рис. 5.42 *AlignmentGraph*(**TGTTA**, **TCGT**), где каждое ребро окрашено в зависимости от того, представляет ли оно совпадение, несовпадение, вставку или удаление

Мышь
...ARRSRTHFTKFQTDILIEAFEKNRFPGIVTREKLAQQTGIPESRIHIWFQNRRARHPDPG...
...ARQKQTFITWTQKNRLVQAFERNPFPDTATRKKLAEQTGLQESRIQMWFQKQRSLYLKKS...
Человек

Рис. 5.43 Гомеодомены мыши и человека

Непосредственный вопрос состоит в том, как найти этот консервативный сегмент в гораздо более длинных генах и игнорировать соседние области, которые обнаруживают мало сходства. Глобальное выравнивание ищет сходство между двумя строками по всей их длине; однако при поиске гомеодоменов мы ищем меньшие локальные области сходства, и нам не нужно выравнивать все строки. Например, глобальное выравнивание между двумя приведенными ниже последовательностями имеет 22 совпадения, 18 **вставок** и 2 несовпадения, что дает результат 22 – 18 – 2 = 2 (если σ = μ = 1).

GCC-C-AGTC-TATGT-CAGGGGGCACG--A-GCATGCACA-
GCCGCC-GTCGT-T-TTCAG----CA-GTTATGT-T-CAGAT

---G----C-----C--CAGTCTATG-TCAGGGGGCACGAGCATGCACA
GCCGCCGTCGTTTTCAGCAGT-TATGTTCAG-----A------T-----

Рис. 5.44 Глобальное выравнивание последовательностей

Однако эти последовательности можно выровнять по-разному (с 17 совпадениями и 32 вставками) на основе высококонсервативного интервала, представленного подстроками CAGTCTATGTCAG и CAGTTATGTTCAG.

Это выравнивание имеет меньше совпадений и более низкий счет 17 – 32 = –15, даже несмотря на то, что консервативная область выравнивания дает оценку 12 – 2 = 10, что вряд ли является случайностью.

На рис. 5.45 показаны два пути выравнивания, соответствующие этим двум различным выравниваниям. Верхний путь, соответствующий второму выров-

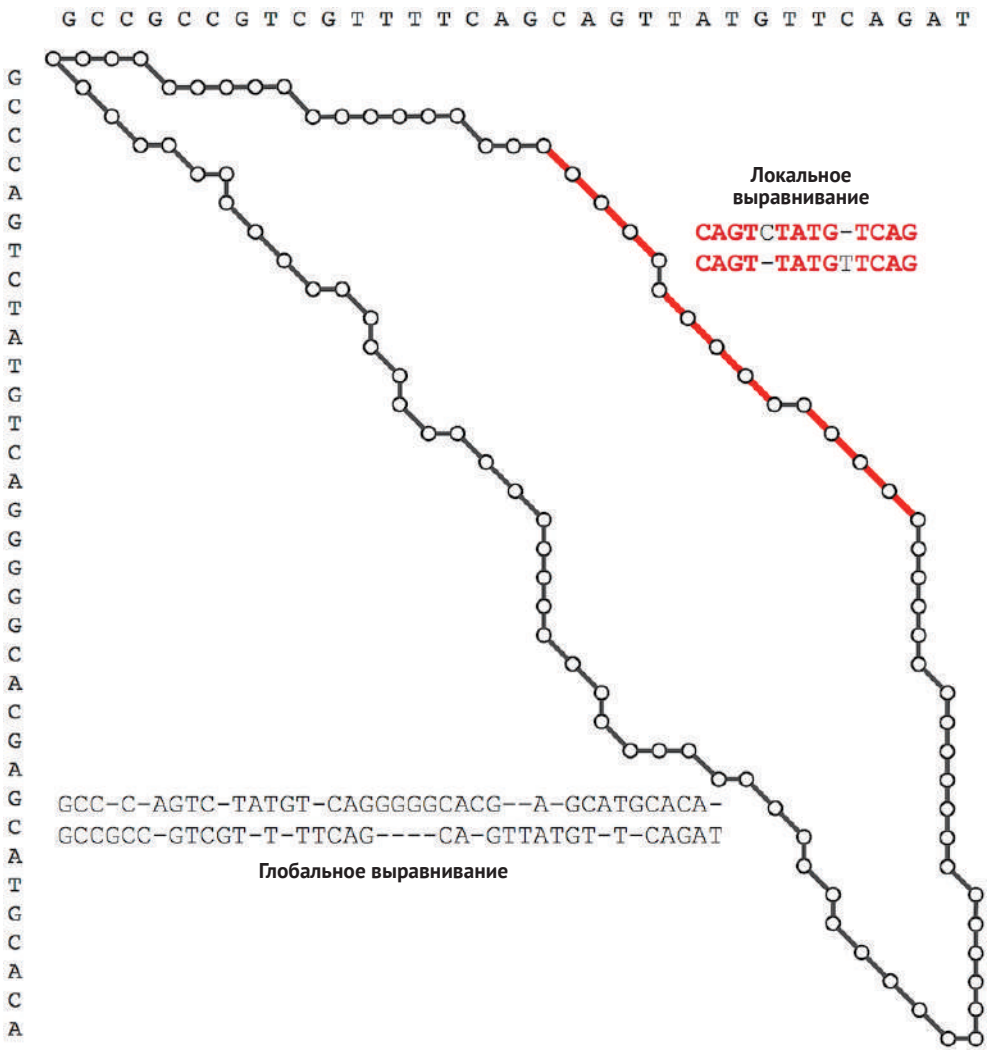


Рис. 5.45 Глобальное и локальное выравнивание двух цепей ДНК, имеющих общий высококонсервативный интервал. Релевантное выравнивание, которое захватывает этот интервал (верхний путь), проигрывает нерелевантному выравниванию (нижний путь), поскольку первое влечет за собой большие штрафы за вставки

ниванию выше, проигрывает, потому что он содержит много сильно оштрафованных отступов по обе стороны от диагонали, соответствующей сохраняемому интервалу. В результате глобальное выравнивание выводит биологически нерелевантный нижний путь. Как мы можем исключить биологически нерелевантные выравнивания, когда присутствуют локальные сходства?

Когда биологически значимое сходство присутствует в некоторых частях последовательностей v и ω и отсутствует в других, биологи пытаются игнорировать глобальное выравнивание и вместо этого выравнивают *подстроки* v и ω, что дает **локальное выравнивание** двух строк. Задача поиска подстрок, которые максимизируют глобальный счет выравнивания по всем подстрокам v и ω, называется **задачей локального выравнивания**.

Задача локального выравнивания: *найдите локальное выравнивание с наивысшим счетом между двумя строками.*

Input: строки v и ω, а также матрица счета *Score*.

Output: подстроки v и ω, глобальный счет выравнивания которых (определенный параметром *Score*) максимален среди всех подстрок v и ω.

Простой, но неэффективный способ решения задачи локального выравнивания состоит в том, чтобы найти самый длинный путь, соединяющий каждую пару узлов в графе выравнивания (а не только узлы, соединяющие источник и сток, как в глобальной задаче выравнивания), а затем выбрать путь, имеющий максимальный вес среди всех этих путей максимальной длины.

ОСТАНОВИТЕСЬ и задумайтесь. Каково время выполнения этого алгоритма?

Бесплатные поездки на такси в графе выравнивания

Для создания более быстрого алгоритма представьте себе «бесплатную поездку на такси» от источника (0, 0) до узла, представляющего начальный узел фиксированного (красного) интервала на рисунке ниже. Представьте также бесплатную поездку на такси от конечного узла фиксированного интервала до стока. Если бы такие поездки были доступны, то вы могли бы добраться до начального узла фиксированного интервала бесплатно, вместо того чтобы нести большие штрафы, как при глобальном выравнивании. Затем вы можете перемещаться по фиксированному интервалу к его конечному узлу, накапливая много положительных результатов совпадения, сталкиваясь при этом с небольшим количеством штрафов за несовпадение и удаление. Наконец, вы можете совершить еще одну бесплатную поездку от конечного узла фиксиро-

ванного интервала до стока. Результирующий счет этой поездки равен счету выравнивания только фиксированных интервалов, как и требовалось.

Единственная проблема с идеей «бесплатных поездок» заключается в том, что мы не знаем заранее, где начинается и заканчивается локальное выравнивание (т. е. фиксированный красный интервал на рисунке ниже)! Тем не менее мы можем обеспечить бесплатную поездку от источника ко *всем* узлам и от *всех* узлов к стоку.

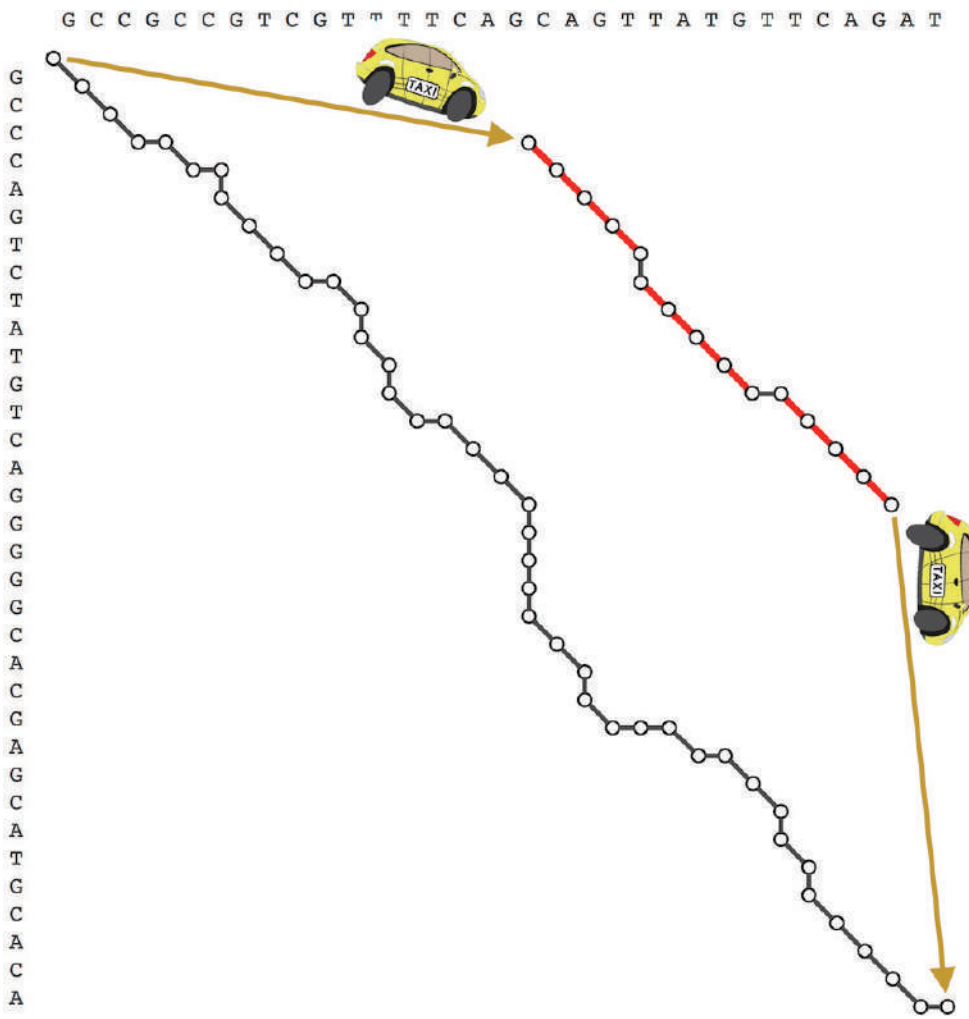


Рис. 5.46 Изменение предыдущего рисунка путем добавления ребер «бесплатная поездка на такси» (имеющих вес 0), соединяющих источник с начальным узлом фиксированного интервала и соединяющих конечный узел фиксированного интервала со стоком. Эти новые ребра позволяют нам оценивать только локальное выравнивание, содержащее фиксированный интервал

Соединение источника (0, 0) с каждым другим узлом путем добавления ребра с нулевым весом и соединение каждого узла со стоком (n, m) ребром с нулевым весом приведет к созданию DAG, идеально подходящего для решения задачи локального выравнивания, как показано ниже. Из-за бесплатных поездок на такси нам больше не нужно строить самый длинный путь между каждой парой узлов в графе – самый длинный путь от источника к стоку дает оптимальное локальное выравнивание!

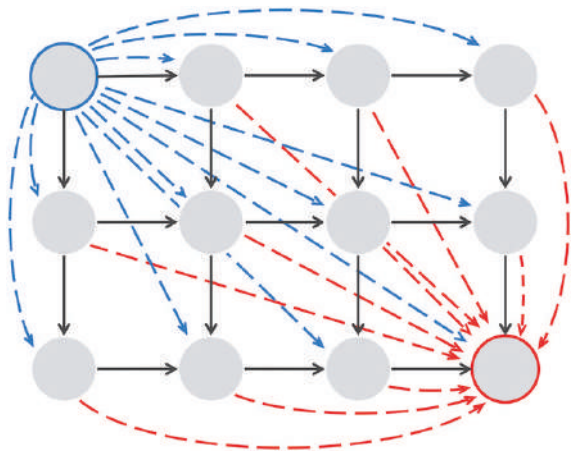


Рис. 5.47 Алгоритм локального выравнивания вводит ребра нулевого веса (показаны синими пунктирными линиями), соединяющие источник (0, 0) с каждым другим узлом в графе выравнивания, а также ребра нулевого веса (показаны красными пунктирными линиями), присоединяющие каждый узел к узлу стока

Общее количество ребер в приведенном выше графе равно $O(|v| \cdot |\omega|)$, что все еще мало. Поскольку время выполнения поиска самого длинного пути в DAG определяется количеством ребер в графе, результирующий алгоритм локального выравнивания будет быстрым. Что касается вычисления значений $s_{i,j}$, добавление ребер нулевого веса из (0, 0) к каждому узлу сделало исходный узел (0, 0) предшественником каждого узла (i, j). Следовательно, теперь в (i, j) входят четыре ребра, что добавляет только один новый член к рекуррентному соотношению самого длинного пути:

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j} + Score(v_i, -) \\ s_{i,j-1} + Score(-, w_j) \\ s_{i-1,j-1} + Score(v_i, w_j) \end{cases}.$$

Рекуррентное соотношение выше включает бесплатные поездки от *source* = (0, 0), но не включает бесплатные поездки до *sink* = (*n*, *m*). Так как *sink* имеет любой другой узел в качестве предшественника, $s_{n,m}$ будет равно наибольшему значению $s_{i,j}$ по всему графу выравнивания,

$$s_{n,m} = \max_{0 \le i \le n, 0 \le j \le n} s_{i,j}.$$

ОСТАНОВИТЕСЬ и задумайтесь. После вычисления всех значений $s_{i,j}$ в графе выравнивания как вы можете найти, где начинается и заканчивается путь, соответствующий наилучшему локальному выравниванию?

Вам все еще может быть интересно, как мы можем использовать эти бесплатные поездки на такси по графу выравнивания. Дело в том, что вы отвечаете за разработку любого типа DAG, подобного Манхэттену, который вам нравится, если он адекватно моделирует конкретную задачу выравнивания. Трансформации, такие как бесплатные поездки на такси, станут общей темой этой главы. Различные проблемы выравнивания можно решить, создав соответствующую DAG с как можно меньшим количеством ребер (чтобы свести к минимуму время выполнения), назначив веса ребрам для моделирования требований проблемы, а затем найдя самый длинный путь в этой DAG.

Меняющиеся грани выравнивания последовательности

В этом разделе мы опишем три задачи сравнения последовательностей и позволим вам применить то, что вы уже узнали, для их решения. Подсказка: идея состоит в том, чтобы представить каждую задачу как пример самого длинного пути в задаче DAG.

Задача 1. Расстояние редактирования

В 1966 году Владимир Левенштейн ввел понятие **расстояния редактирования** между двумя строками как минимальное количество **операций редактирования**, необходимых для преобразования одной строки в другую. Здесь операция редактирования представляет собой вставку, удаление или замену одного символа. Например, TGCATACT можно преобразовать в ATCCGAT с помощью шести операций редактирования, что означает, что расстояние редактирования между этими строками не превышает 6.

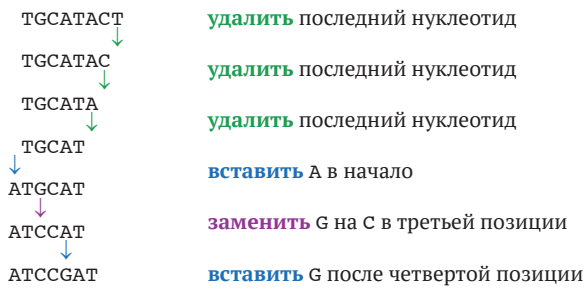


Рис. 5.48 Расстояние редактирования

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы преобразовать TGCATACT в ATCCGAT, используя меньшее количество операций?

На самом деле расстояние редактирования между TGCATACT и ATCCGAT равно 5:

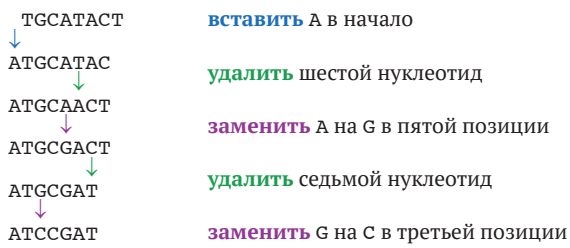


Рис. 5.49 Настоящее расстояние редактирования

Левенштейн ввел понятие расстояние редактирования, но не описал алгоритм его вычисления, разработку которого мы оставляем вам.

Задача определения расстояния редактирования: *найдите расстояние редактирования между двумя строками.*

Input: две строки.

Output: расстояние редактирования между этими строками.

Задача 2. Настройка выравнивания

Скажем, мы хотим сравнить NRP-синтетазу длиной примерно 20 000 аминокислот из *Bacillus brevis* с A-доменом длиной примерно 600 аминокислот из *Streptomyces roseosporus*, бактерии, которая производит мощный антибиотик даптомицин. Мы надеемся найти область в более длинной белковой последо-

вательности v, которая имеет большое сходство со всей более короткой последовательностью ω. Глобальное выравнивание не будет работать, потому что оно пытается выровнять все v по всем ω; локальное выравнивание не будет работать, потому что оно пытается выровнять подстроки как v, так и ω. Таким образом, у нас есть отдельное приложение для выравнивания, которое называется **задачей настройки выравнивания**.

«Подгонка» ω к v требует нахождения подстроки v' строки v, которая максимизирует счет глобального выравнивания между v' и ω среди всех подстрок v. Например, наилучшие глобальные, локальные и подходящие выравнивания v = CGTAGGCTTAAGGTTA и ω = ATAGATA показаны на рисунке ниже (со штрафами за несовпадение и вставку, равными 1).

Задача настройки выравнивания: *построить выравнивание с наивысшим счетом между двумя строками.*

Input: строки v и ω, а также матрица *Score*.

Output: выравнивание с наивысшим счетом между v и ω, как определено матрицей счета Score.

Обратите внимание на рисунок, который показывает, что оптимальное локальное выравнивание (со счетом 3) не является подходящим. С другой стороны, счет оптимального глобального выравнивания (6 – 9 – 1 = –4) меньше, чем у наиболее подходящего выравнивания (5 – 2 – 2 = +1).

Рис. 5.50 Глобальное, локальное и настроенное выравнивание. Черные символы не влияют на локальное и настроенное выравнивание и показаны только для иллюстрации соседних областей

Задача 3. Выравнивание с перекрытием

В главе о сборке генома мы обсуждали, как использовать перекрывающиеся риды для сборки, – задача, которая осложнялась ошибками в ридах. Выравнивание концов гипотетических ридов, показанных ниже, позволяет найти перекрытия между ридами с ошибками.

```
ATGCATGCCGG
     T—CC—GAAAC
```

Выравнивание с перекрытием строк $v = v_1 \ldots v_n$ и $\omega = \omega_1 \ldots \omega_m$ – это глобальное выравнивание суффикса v с префиксом ω. Оптимальное выравнивание

с перекрытием строк v и ω максимизирует глобальную оценку выравнивания между i-суффиксом v и j-префиксом ω (т. е. между $v_1 \ldots v_n$ и $\omega_1 \ldots \omega_j$) среди всех i и j.

Задача выравнивания с перекрытием: *построить выравнивание с перекрытием между двумя строками с наивысшим счетом.*

Input: две строки и матрица *score*.

Output: выравнивание с перекрытием между двумя строками с наивысшим счетом, как определено матрицей счета *score*.

Штрафы за вставки и удаления при выравнивании последовательности

Штрафы за аффинные пробелы

Мы видели, что введение штрафов за несовпадение и вставки может привести к более биологически адекватным глобальным выравниваниям. Однако даже с этой более надежной моделью счета выравнивание A-доменов, которое мы ранее построили (со штрафом за задержку $\sigma = 4$), по-прежнему показывает только 6 из 8 консервативных фиолетовых столбцов, соответствующих нерибосомным сигнатурам:

```
YAFDLGYTCMFP-VLL-GGGELHIV-QKETYTAPDEI-AHYIKEHGITYI-KLTPSLFHTIVNTASFAFDANFE
-AFDVS-AGDFARALLTGG-QL-IVCPNEVKMDPASLYA-IIKKYDIT-IFEATPAL--VIPLME-YIYEQKLD

-S-LR-LIVLGGEKIIPIDVIAFRKM---YGHTE-FINHYGPTEATIGA
ISQLQILIV-GSDSC-SME--DFKTLVSRFGSTIRIVNSYGVTEACIDS
```

Рис. 5.51 Выравнивание A-доменов показывает только 6 из 8 консервативных фиолетовых столбцов

ОСТАНОВИТЕСЬ и задумайтесь. Не выявит ли увеличение штрафа за вставку (индел) с $\sigma = 4$ до $\sigma = 10$ биологически правильное выравнивание?

В нашей ранее определенной **линейной модели подсчета** если σ является штрафом за вставку или удаление одного символа, то $\sigma \cdot k$ является штрафом за вставку или удаление интервала из k символов. Эта модель суммирования, к сожалению, приводит к неадекватному счету биологических последовательностей. Мутации часто вызываются ошибками в репликации ДНК, которые вставляют или удаляют весь интервал из k нуклеотидов одномоментно, а не

как k независимых вставок или делеций. Таким образом, штрафование такой вставки на $\sigma \cdot k$ представляет собой чрезмерное наказание. Например, выравнивание справа более адекватно, чем выравнивание слева, но в то же время время они получат одинаковый счет.

```
GATCCAG      GATCCAG
GA-C-AG      GA--CAG
```

Пробел – это непрерывная последовательность пропусков в строке выравнивания. Один из способов более подходящего счета пропусков состоит в том, чтобы определить **аффинный штраф** за пропуск длины k как $\sigma + \varepsilon \cdot (k - 1)$, где σ – штраф за открытие пробела, начисляемый на первый символ в пробеле, а ε – это штраф за расширение пробела, начисляемый на каждый дополнительный символ в пробеле. Мы выбираем ε меньше, чем σ, чтобы аффинный штраф за пробел длины k был меньше, чем штраф за k независимых однонуклеотидных вставок ($\sigma \cdot k$). Например, при штрафах за аффинные пробелы выравнивание слева вверху оштрафовано на 2σ, тогда как выравнивание справа оштрафовано только на $\sigma + \varepsilon$.

Задача выравнивания со штрафами за аффинные пробелы:
построить глобальное выравнивание с наивысшим счетом между двумя строками (со штрафами за аффинные пробелы).

Input: две строки, *score* и числа σ и ε.

Output: глобальное выравнивание с наивысшим счетом между этими строками, как определено матрицей *score* и штрафами за открытие пробела и расширение σ и ε.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы изменили граф выравнивания, чтобы решить эту задачу?

На рис. 5.52 показано, как можно смоделировать штрафы за аффинные пробелы в графе выравнивания, введя новое «длинное» ребро для каждого пробела.

Поскольку мы заранее не знаем, где должны располагаться пробелы, нам нужно добавить ребра с учетом всех возможных пробелов. Таким образом, штрафы за аффинные пробелы можно компенсировать, добавляя все возможные вертикальные и горизонтальные ребра в граф выравнивания для представления всех возможных пробелов. В частности, мы добавляем ребра, соединяющие (i, j) как с $(i + k, j)$, так и с $(i, j + k)$ с весами $\sigma + \varepsilon \cdot (k - 1)$ для всех возможных размеров пробела k, как показано на рис. 5.53. Для двух последовательностей длины n количество ребер в результирующем графе выравнивания, моделирующем штрафы за аффинные пробелы, увеличивается с $O(n^2)$ до $O(n^3)$.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы спроектировать DAG всего с $O(n^2)$ ребрами, чтобы решить задачу выравнивания со штрафами за аффинные пробелы?

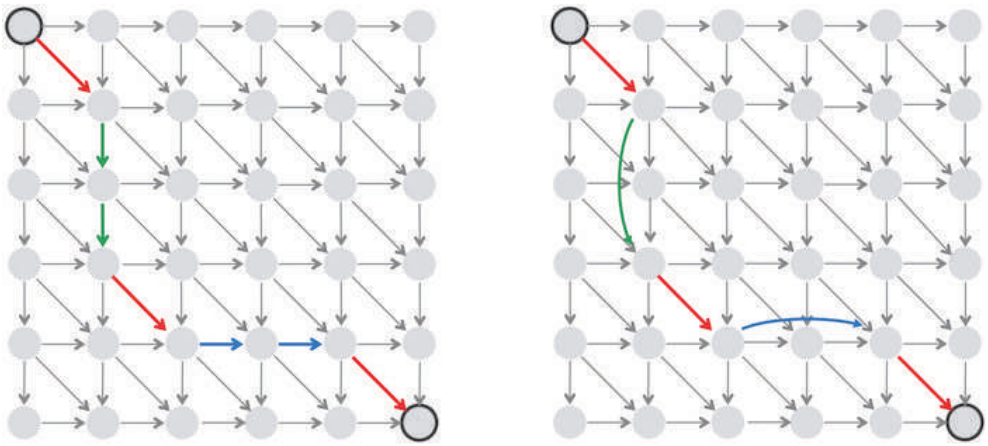


Рис. 5.52 Представление пробелов в графе выравнивания слева как «длинных» ребер вставки и удаления в графе выравнивания справа. Для промежутка длины k вес соответствующего длинного ребра равен $\sigma + \varepsilon \cdot (k - 1)$

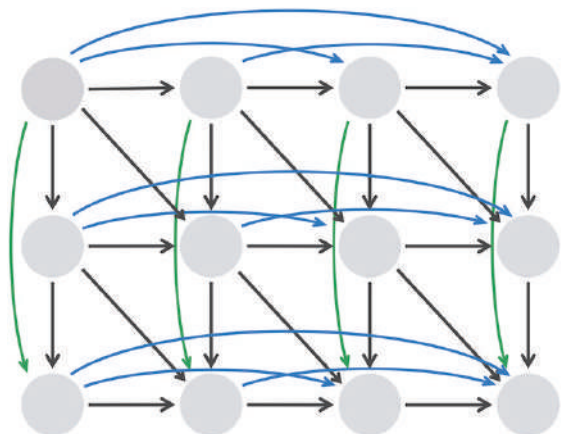


Рис. 5.53 Добавление ребер, соответствующих инделам для всех возможных размеров пробелов, добавляет большое количество ребер в граф выравнивания

Строительство графа Манхэттена на трех уровнях

Уловка для уменьшения количества ребер в DAG в задаче выравнивания с аффинными штрафами за пробел заключается в увеличении количества узлов. Для этого построим граф выравнивания на трех уровнях; для каждого узла (i, j) мы построим три разных узла: $(i, j)_{\text{lover}}$, $(i, j)_{\text{middle}}$ и $(i, j)_{\text{upper}}$. Средний уровень будет содержать диагональные ребра веса $score(v_i, \omega_j)$, представляющие совпадения и несовпадения. Нижний уровень будет иметь только вертикальные ребра с весом $-\varepsilon$ для представления расширений пробелов в v, а верхний уровень будет иметь только горизонтальные ребра с весами $-\varepsilon$ для представления расширений пробелов в ω (рисунок ниже).

Рис. 5.54 Построение трехуровневого графа для выравнивания со штрафами за аффинные пробелы. Нижний уровень соответствует расширениям пробелов в v, средний уровень соответствует совпадениям и несовпадениям, а верхний уровень соответствует расширениям пробелов в ω

Жизнь в таком трехуровневом городе была бы трудной, потому что в настоящее время нет возможности перемещаться между разными уровнями. Чтобы решить эту задачу, мы добавляем ребра, отвечающие за открытие и закрытие пробелов. Чтобы смоделировать открытие пробела, мы соединяем каждый узел $(i, j)_{\text{middle}}$ с узлами $(i + 1, j)_{\text{lover}}$ и $(i, j + 1)_{\text{upper}}$; затем взвешиваем эти ребра с по-

мощью −σ. Закрытие пробела не влечет за собой штрафа, поэтому мы вводим ребра нулевого веса, соединяющие узлы : $(i, j)_{\text{lover}}$ и $(i, j)_{\text{upper}}$ с соответствующим узлом $(i, j)_{\text{middle}}$ на среднем уровне. В результате пробел длины k начинается и заканчивается на среднем уровне и начисляется −σ за первый символ, −ε за каждый последующий символ и 0 за закрытие пробела, что приводит к общему штрафу в размере $\sigma + \varepsilon \cdot (k - 1)$, как и требовалось. На рисунке ниже показано, как путь с рис. 5.37 пересекает трехуровневый граф.

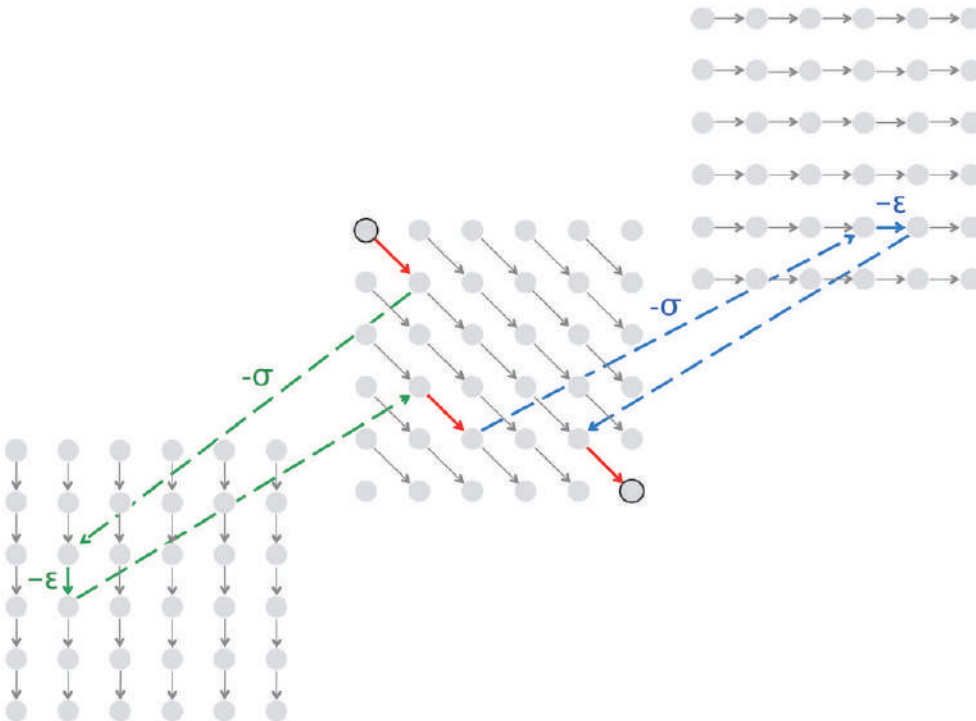


Рис. 5.55 Трехуровневый граф

Упражнение. Докажите, что количество ребер в трехуровневом графе, изображенном на рисунке ниже, не превосходит $7 \cdot n \cdot m$ для последовательностей длины n и m.

Только что построенный DAG может быть сложным, но он использует только $O(n \cdot m)$ ребер для последовательностей длины n и m, и самый длинный путь в этом графе по-прежнему создает оптимальное выравнивание с штрафами за аффинные пробелы. Трехуровневый граф выравнивания преобразуется

в систему трех рекуррентных соотношений, показанную ниже. Здесь $lower_{i,j}$, $middle_{i,j}$ и $upper_{i,j}$ – длины самых длинных путей от исходного узла до $(i, j)_{\text{lover}}$, $(i, j)_{\text{middle}}$ и $(i, j)_{\text{upper}}$ соответственно.

$$lower_{i,j} = \max \begin{cases} lower_{i-1,j} & -\varepsilon \\ middle_{i-1,j} & -\sigma \end{cases};$$

$$upper_{i,j} = \max \begin{cases} upper_{i,j-1} & -\varepsilon \\ middle_{i,j-1} & -\sigma \end{cases};$$

$$middle_{i,j} = \max \begin{cases} lowwer_{i,j} \\ middle_{i-1,j-1} + score(v_i, w_j). \\ upper_{i,j} \end{cases}$$

Переменная $lower_{i,j}$ вычисляет счет оптимального выравнивания между i-префиксом υ и j-префиксом ω, заканчивающимся делецией (т. е. вертикальным краем), тогда как переменная $upper_{i,j}$ вычисляет счет оптимального выравнивания этих префиксов, заканчивающихся вставкой (т. е. горизонтальным краем), и переменная $middle_{i,j}$ вычисляет счет оптимального выравнивания, заканчивающегося совпадением или несовпадением. Первый рекуррентный член для $lower_{i,j}$ и $upper_{i,j}$ соответствует расширению пробела, тогда как второй член соответствует его возникновению.

ОСТАНОВИТЕСЬ и задумайтесь. Вычислите оптимальное выравнивание со штрафами за аффинные пробелы для А-доменов, рассмотренных в начале этого раздела и воспроизведенных ниже. Как различные штрафы за открытие пробела и его расширение влияют на качество выравнивания?

Строка 1:

YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTAS
FAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTEFINHYGPTEATIGA

Строка 2:

AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYIYE
QKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS

Упражнение. Разработайте алгоритм для вычисления оптимального локального (а не глобального) выравнивания с штрафами за аффинные пробелы.

Компактное выравнивание последовательности

Вычисление счета выравнивания с использованием линейной памяти

Чтобы представить настройку выравнивания, мы использовали пример выравнивания NRP-синтетазы длиной 20 000 аминокислот из *Bacillus brevis* с А-доменом длиной 600 аминокислот из *Streptomyces roseosporus*. Однако вы, возможно, не сможете построить это выравнивание на своем компьютере, потому что память, необходимая для хранения матрицы динамического программирования, значительна.

Время выполнения алгоритма динамического программирования для выравнивания двух последовательностей длин n и m пропорционально количеству ребер в их графе выравнивания, которое равно $O(n \cdot m)$. Память, необходимая для этого алгоритма, также равна $O(n \cdot m)$, так как нам нужно хранить ссылки с возвратом. Теперь мы продемонстрируем, как построить выравнивание всего за $O(n)$ памяти за счет удвоения времени выполнения (это означает, что время выполнения по-прежнему составляет $O(n \cdot m)$). В этом разделе для простоты мы рассмотрим случай глобального выравнивания со штрафами за неаффинные пропуски, т. е. когда каждый символ в пропуске дает одинаковый счет.

Алгоритм «разделяй и властвуй» часто работает, когда решение большой проблемы может быть построено с помощью решений более мелких задач. Такая стратегия проходит в два этапа. Фаза **разделения** разбивает задачу на более задачи и решает их; фаза **объединения** сшивает меньшие решения в решение исходной задачи. Чтобы узнать больше, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Алгоритмы «разделяй и властвуй»**.

Прежде чем мы приступим к алгоритму «разделяй и властвуй» для выравнивания в линейном пространстве, обратите внимание, что если мы хотим вычислить только счет выравнивания, а не получить само выравнивание, то требуемое пространство можно легко сократить до удвоенного количества узлов в одном столбце графа выравнивания, или $O(n)$. Это уменьшение вытекает из наблюдения, что единственными значениями, необходимыми для вычисления оценок выравнивания в столбце j, являются счета выравнивания в столбце $j - 1$. Следовательно, счета выравнивания в столбцах перед столбцом $j - 1$ могут быть отброшены при вычислении оценок выравнивания для столбца j, как показано на рисунке ниже. К сожалению, для нахождения самого длинного пути требуется хранить всю матрицу указателей возврата, что требует квадратичного пространства. Идея метода сокращения пространства заключается в том, что нам не нужно хранить какие-либо указатели возврата, если мы готовы потратить немного больше времени. На самом деле мы покажем, как построить выравнивание без сохранения указателей возврата.

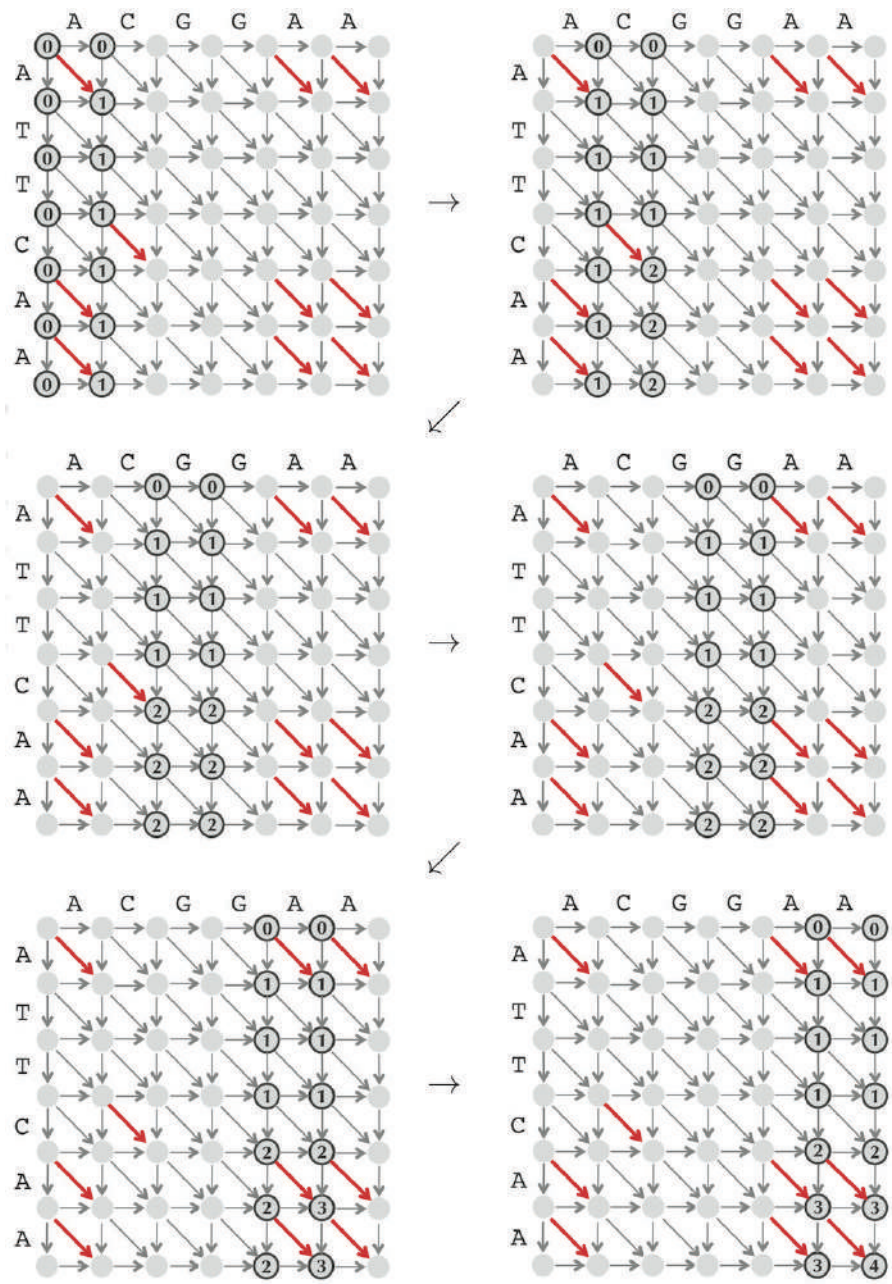


Рис. 5.56 Вычисление счета выравнивания LCS путем сохранения оценок всего в двух столбцах графика выравнивания. Как и прежде, все красные ребра имеют вес 1, а все остальные ребра имеют вес 0

Задача среднего узла

Для заданных строк $v = v_1 \ldots v_n$ и $\omega = \omega_1 \ldots \omega_m$ определите $middle = \lfloor m/2 \rfloor$. Средний столбец $AlignmentGraph(v, \omega)$ – это столбец, содержащий все узлы $(i, middle)$ для $0 \le i \le n$. Самый длинный путь от источника к стоку на графе выравнивания должен где-то пересекать средний столбец, и наша первая задача – выяснить, где используется только $O(n)$ памяти. Мы называем узел, в котором самый длинный путь пересекает средний столбец, **средним узлом** (обратите внимание, что разные самые длинные пути могут иметь разные средние узлы, а данный самый длинный путь может иметь более одного среднего узла). На рисунке ниже $middle = 3$, и оптимальный путь выравнивания пересекает средний столбец в (уникальном) среднем узле (4, 3).

Ключевым наблюдением является то, что мы можем найти средний узел самого длинного пути без необходимости строить этот путь в графе выравнивания. Мы будем классифицировать путь от источника к стоку как ***i*-путь**, если он проходит через средний столбец в строке i. Например, выделенный ниже путь является 4-путем, поскольку он проходит через средний столбец в строке 4. Для каждого i от 0 до n мы хотели бы найти длину самого длинного i-го пути (обозначаемого как $Length(i)$), потому что наибольшее значение $Length(i)$ среди всех i покажет средний узел.

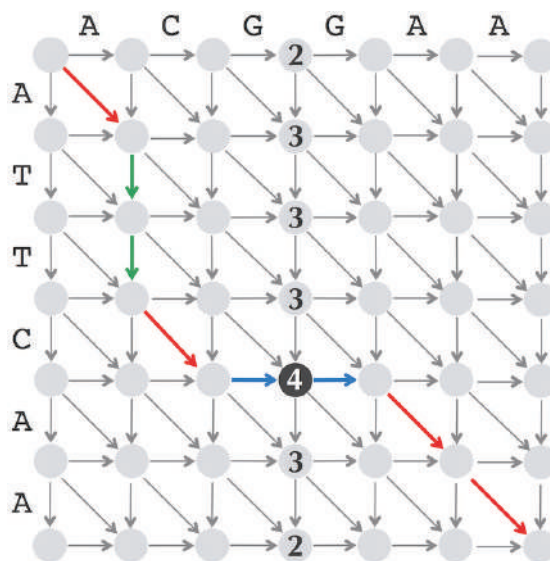


Рис. 5.57 Граф выравнивания ATTCAA и ACGGAA с выделенным путем выравнивания LCS. Число внутри каждого узла $(i, middle)$ в среднем столбце равно длине (i). Узел maximizing$Length(i)$ является средним узлом (может существовать несколько средних узлов); средний узел на этом графике окрашен в черный цвет

Упражнение. Найдите все средние узлы для всех самых длинных путей при поиске самой длинной общей подпоследовательности ACCA и CAAC.

Пусть *FromSource*(*i*) обозначает длину самого длинного пути от источника, заканчивающегося в (*i*, *middle*), а *ToSink*(*i*) обозначает длину самого длинного пути от (*i*, *middle*) до стока. Безусловно,

$$Length(i) = FromSource(i) + ToSink(i),$$

поэтому нам нужно вычислить *FromSource*(*i*) и *ToSink*(*i*) для каждого *i*.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы вычислить *FromSource*(*i*) и *ToSink*(*i*) в линейном пространстве? Сколько времени это займет?

Значение *FromSource*(*i*) равно просто $s_{i,middle}$, которое, как мы уже знаем, может быть вычислено в линейном пространстве. Таким образом, значения *FromSource*(*i*) для всех *i* хранятся как $s_{i,middle}$ в среднем столбце графа выравнивания, и для их вычисления требуется пройти половину графа выравнивания от столбца 0 до среднего столбца. Поскольку для вычисления *FromSource*(*i*) нам нужно исследовать примерно половину ребер графа выравнивания, мы говорим, что время выполнения, необходимое для вычисления всех значений *FromSource*(*i*), пропорционально половине «площади» графа выравнивания, или $n \cdot m/2$ (рис. 5.58).

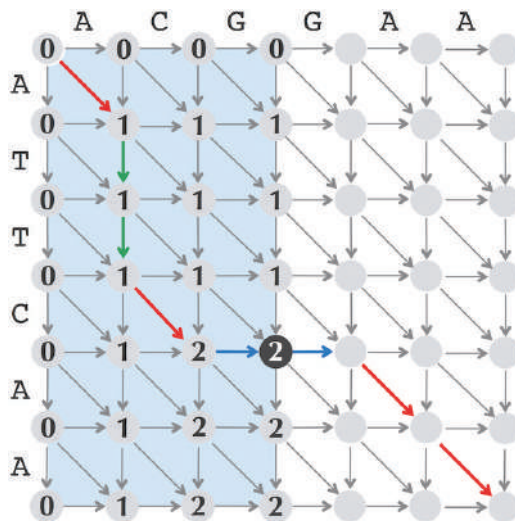


Рис. 5.58 Вычисление *FromSource*(*i*) для всех *i* может быть выполнено за *O*(*n*) пространство и время, пропорциональное $n \cdot m/2$

Вычисление *ToSink*(*i*) эквивалентно нахождению самого длинного пути от стока к (*i*, *middle*), если все направления ребер поменялись на обратные. Вместо того чтобы перенаправлять ребра, мы можем реверсировать строки $\upsilon = \upsilon_1 \ldots \upsilon_n$ и $\omega = \omega_1 \ldots \omega_m$ и найти $s_{n-i,\, m-middle}$ в графе выравнивания для $\upsilon_n \ldots \upsilon_1$ и $\omega_m \ldots \omega_1$. Таким образом, вычисление *ToSink*(*i*) похоже на вычисление *FromSource*(*i*) и также может быть выполнено в пространстве O(*n*) и времени выполнения, пропорциональном $n \cdot m/2$, или половине площади графа выравнивания (рисунок ниже). В сумме мы можем вычислить все значения *Length*(*i*) = *FromSource*(*i*) + *ToSink*(*i*) в линейном пространстве со временем выполнения, пропорциональным $n \cdot m/2 + n \cdot m/2 = n \cdot m$, что является общей площадью графа выравнивания.

Похоже, мы потратили много времени только на то, чтобы найти единственный узел на пути выравнивания! Вы можете подумать, что этот метод обречен на провал, потому что мы уже потратили $O(n \cdot m)$ времени (вся площадь графа выравнивания), чтобы получить очень мало информации.

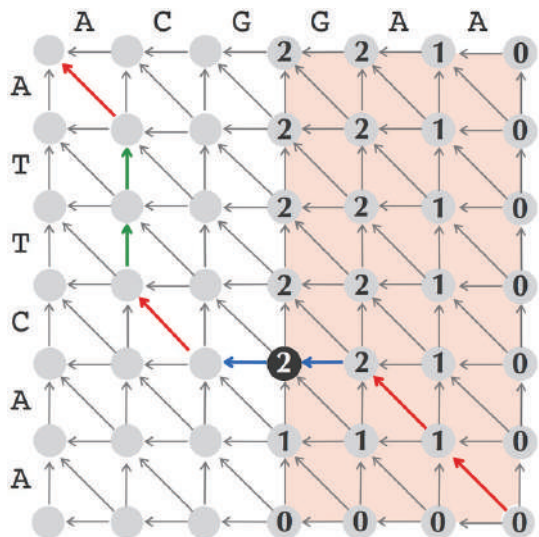


Рис. 5.59 Вычисление *ToSink*(*i*) для всех *i* также может быть выполнено за *O*(*n*) пространство и время, пропорциональное $n \cdot m/2$; для этого требуется изменить направление всех ребер на противоположное и рассматривать сток как источник

Удивительно быстрый и экономичный алгоритм выравнивания

Как только мы нашли средний узел, мы автоматически знаем два прямоугольника, через которые должен проходить самый длинный путь по обе стороны от среднего узла. Как показано на рисунке ниже, один из этих прямоугольников состоит из всех узлов выше и левее среднего узла, тогда как другой прямоугольник состоит из всех узлов ниже и правее среднего узла. Таким образом, площадь двух выделенных прямоугольников составляет половину общей площади графа выравнивания.

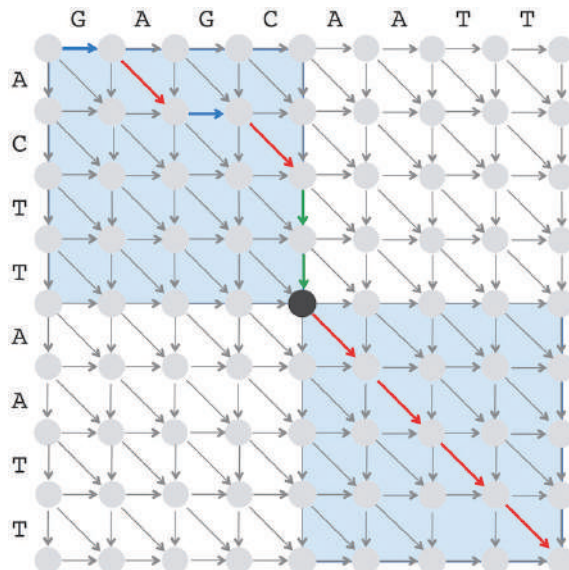


Рис. 5.60 Средний узел (показан черным) определяет два выделенных прямоугольника и показывает, что оптимальный путь должен проходить внутри этих прямоугольников. Поэтому мы можем исключить оставшиеся части графа выравнивания из рассмотрения для поиска оптимального пути выравнивания

Теперь мы можем *разделить* задачу нахождения длиннейшего пути из (0, 0) в (n, m) на две подзадачи: найти длиннейший путь из (0, 0) в средний узел; найти самый длинный путь от среднего узла до (n, m). Шаг объединения находит два средних узла внутри меньших прямоугольников, что может быть выполнено за время, пропорциональное сумме площадей этих прямоугольников, или $n \cdot m/2$ (рис. 5.61). Обратите внимание, что мы восстановили три узла оптимального пути.

На следующей итерации мы будем «разделять и властвовать», чтобы найти четыре средних узла за время, равное сумме площадей еще меньших синих прямоугольников, имеющих общую площадь $n \cdot m/2$ (рис. 5.62). Мы реконструировали почти все узлы оптимального пути выравнивания!

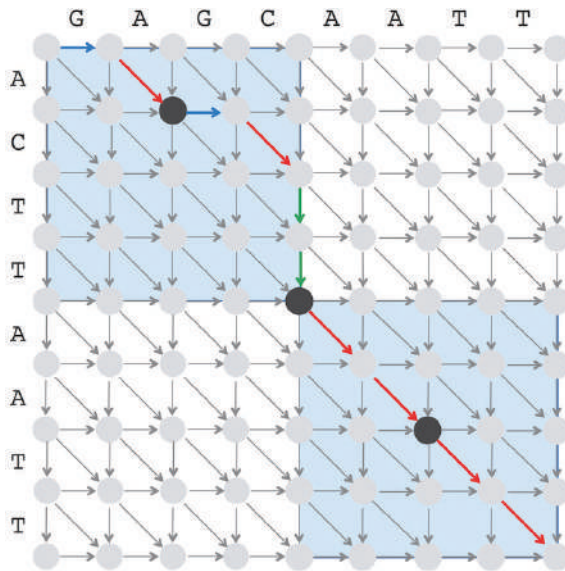


Рис. 5.61 Поиск средних узлов (показанных еще двумя черными кружками) внутри ранее определенных прямоугольников

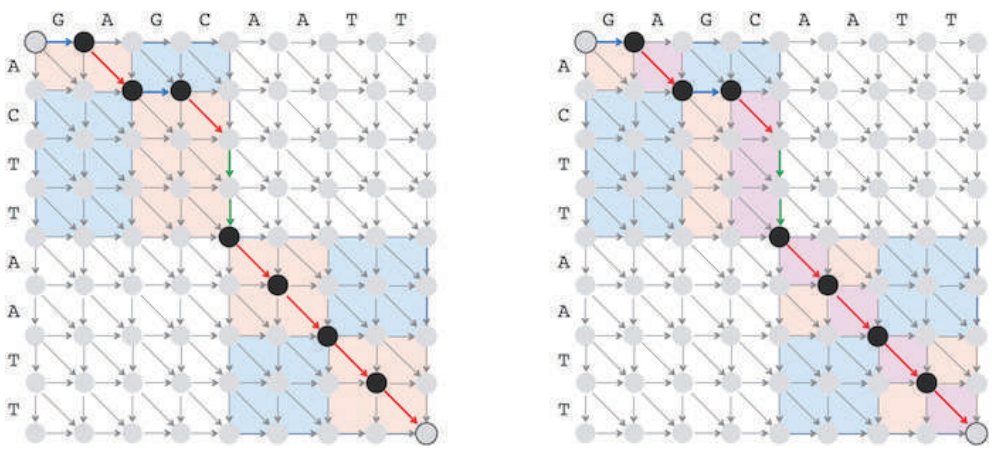


Рис. 5.62 Поиск средних узлов (выделены черными кружками) внутри ранее определенных синих прямоугольников

ОСТАНОВИТЕСЬ и задумайтесь. Сколько времени потребуется, чтобы найти все узлы на пути оптимального выравнивания?

Как правило, на каждом новом шаге, перед последним шагом, мы удваиваем количество найденных средних узлов и вдвое сокращаем время выполнения, необходимое для поиска средних узлов. Действуя таким образом, мы найдем средние узлы всех прямоугольников (и таким образом построим весь ряд!) за время, равное

$$n \cdot m + n \cdot m/2 + n \cdot m/4 + \cdots < 2 \cdot n \cdot m = O(n \cdot m).$$

Таким образом, мы пришли к алгоритму с квадратичным временем, который требует только линейного пространства.

Задача среднего ребра

Вместо того чтобы просить вас реализовать алгоритм «разделяй и властвуй», основанный на поиске среднего узла, мы воспользуемся более элегантным методом, основанным на поиске **среднего ребра** или ребра оптимального пути выравнивания, начинающегося в среднем узле (для данного среднего узла может существовать более одного среднего ребра). Как только мы находим среднее ребро, мы определили два прямоугольника, через которые должен пройти самый длинный путь по обе стороны от среднего ребра. Но теперь эти два прямоугольника занимают даже меньше половины площади графа выравнивания (рис. 5.63), что является преимуществом выбора среднего ребра вместо среднего узла.

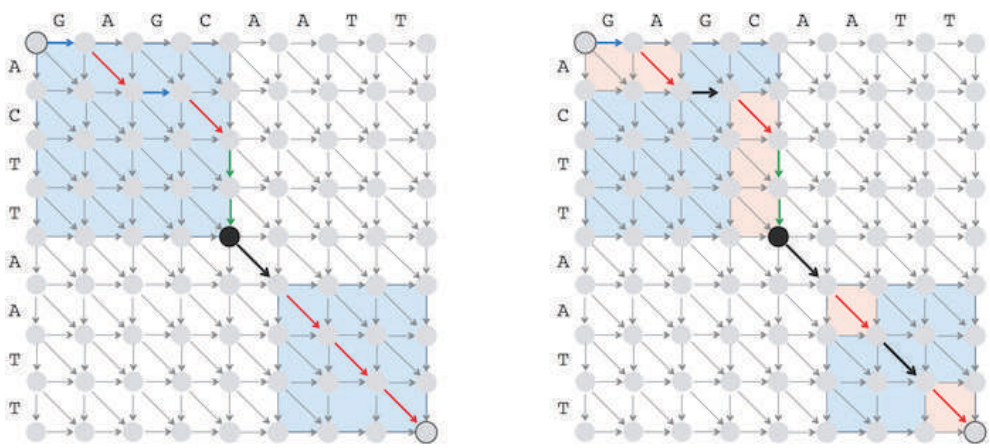


Рис. 5.63 (Слева) Среднее ребро (выделено жирным) начинается в среднем узле (показан черным кругом). Оптимальный путь проходит внутри первого выделенного прямоугольника, проходит через среднее ребро и затем проходит внутри второго выделенного прямоугольника. Остальные части графа выравнивания, занимающие более половины площади, образованной графом, из дальнейшего рассмотрения мы можем исключить. (Справа) Поиск средних ребер (выделены жирным) внутри ранее определенных прямоугольников

Задача поиска среднего ребра в линейном пространстве: *найти среднее ребро в графе выравнивания в линейном пространстве.*

Input: две строки и матрица *score*.

Output: среднее ребро в графе выравнивания этих строк (где длины ребер определяются матрицей *score*).

Приведенный ниже псевдокод для **LinearSpaceAlignment** описывает, как рекурсивно найти самый длинный путь в графе выравнивания, построенном для подстроки $v_{top+1} \ldots v_{bottom}$ строки v и подстроки $\omega_{left+1} \ldots \omega_{right}$ строки ω. **LinearSpaceAlignment** вызывает функцию *MiddleNode*(*top*, *bottom*, *left*, *right*), которая выдает координату *i* среднего узла (*i*, *j*), определяемую последовательностями $v_{top+1} \ldots v_{bottom}$ и $\omega_{left+1} \ldots \omega_{right}$. **LinearSpaceAlignment** также вызывает функцию *MiddleEdge*(*top*, *bottom*, *left*, *right*), которая выдает →, ↓ или ↘ в зависимости от того, является ли среднее ребро горизонтальным, вертикальным или диагональным. Выравнивание строк v и ω в линейном пространстве создается путем вызова **LinearSpaceAlignment**(0, *n*, 0, *m*). Случай *left* = *right* описывает выравнивание пустой строки по строке $v_{top+1} \ldots v_{bottom}$, что тривиально вычисляется как счет промежутка, образованного вертикальными ребрами *bottom* – *top*.

```
LinearSpaceAlignment(v, ω, top, bottom, left, right)
    if left = right
        output путь, сформированный вертикальными ребрами bottom − top
    if top = bottom
        output путь, сформированный горизонтальными ребрами right − left
    middle ← ⌊(left + right)/2⌋
    midEdge ← MiddleEdge(v, ω, top, bottom, left, right)
    midNode ← вертикальная координата начального узла midEdge
    LinearSpaceAlignment(v, ω, top, midNode, left, middle)
    output midEdge
    if midEdge = «→» или midEdge = «↘»
        middle ← middle + 1
    if midEdge = «↓» или midEdge = «↘»
        midNode ← midNode + 1
    LinearSpaceAlignment(v, ω, midNode, bottom, middle, right)
```

Один простой способ реализовать **LinearSpaceAlignment** – вывести каждое ребро как символ v (вертикаль), H (горизонталь) или D (диагональ). Результирующая строка, построенная из трехсимвольного алфавита {v, D, H}, определяет путь через граф выравнивания и позволяет нам реконструировать выравнивание.

Упражнение. Разработайте компактный алгоритм локального выравнивания последовательностей.

Эпилог. Множественное выравнивание последовательностей

Построение трехмерного Манхэттена

Аминокислотные последовательности белков, выполняющих одну и ту же функцию, скорее всего, имеют сходство, но это сходство может быть неуловимым в случае эволюционно отдаленных видов. Теперь у вас есть арсенал алгоритмов для выравнивания пары последовательностей, но, если сходство последовательностей слабое, парное выравнивание может не идентифицировать биологически родственные последовательности. Однако часто одновременное сравнение многих последовательностей позволяет найти сходство, которое не удается выявить при простом парном сравнении последовательностей. Биоинформатики иногда говорят, что парное выравнивание шепчет, а множественное выравнивание кричит.

Теперь мы готовы использовать парный анализ последовательностей, чтобы применить наше интуитивное предположение для сравнения нескольких последовательностей. В нашем трехстороннем выравнивании A-доменов из введения мы обнаружили 19 консервативных столбцов:

```
1: YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTA
2: -AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI
3: IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA

   1: SFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
   2: -YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
   3: ----PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
```

Однако сходство между A-доменами не ограничивается этими 19 колонками, так как мы можем найти 10 + 9 + 12 = 31 полуконсервативную колонку, каждая из которых имеет две совпадающие аминокислоты:

```
1: YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTA
2: -AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI
3: IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA

   1: SFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
   2: -YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
   3: ----PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
```

Множественное выравнивание строк $v^1, \ldots, v^t$, также называемое **выравниванием по *t***, задается матрицей, имеющей t строк, где i-я строка содержит символы v^i по порядку, перемежающиеся символами пробела. Мы также предполагаем, что ни один столбец в множественном выравнивании не содержит только символы пробела. В приведенном ниже трехстороннем выравнивании мы выделили самый популярный символ в каждом столбце, используя заглавные буквы:

```
A  T  –  G  T  T  a  T  A
A  g  C  G  a  T  C  –  A
A  T  C  G  T  –  C  T  c

0  1  2  2  3  4  5  6  7  8
0  1  2  3  4  5  6  7  7  8
0  1  2  3  4  5  5  6  7  8
```

Матрица множественного выравнивания представляет собой обобщение матрицы попарного выравнивания на более чем две последовательности. Три массива, показанные под этим выравниванием, записывают соответствующее количество символов в **ATGTTATA**, **AGCGATCA** и **ATCGTCTC**, встречающихся до заданной позиции. Вместе эти три массива соответствуют пути в трехмерной сетке:

$$(0, 0, 0) \rightarrow (1, 1, 1) \rightarrow (2, 2, 2) \rightarrow (2, 3, 3) \rightarrow (3, 4, 4) \rightarrow (4, 5, 5) \rightarrow (5, 6, 5) \rightarrow (6, 7, 6) \rightarrow (7, 7, 7) \rightarrow (8, 8, 8)$$

Поскольку граф выравнивания для двух последовательностей представляет собой сетку квадратов, граф выравнивания для трех последовательностей представляет собой сетку кубов. Каждый узел в трехстороннем графе выравнивания имеет до семи входящих ребер, как показано на рисунке ниже.

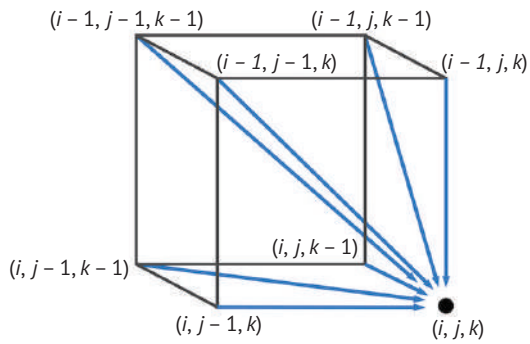


Рис. 5.64 Трехсторонний граф выравнивания

Счет множественного выравнивания определяется как сумма счетов столбцов выравнивания (или, что то же самое, весов ребер на пути выравнивания), при этом оптимальным является такое выравнивание, которое максимизирует этот счет. В случае аминокислотного алфавита мы можем использовать очень общий метод подсчета очков, который определяется t-мерной матрицей, содержащей 21^t элементов и описывающей счета всех возможных комбинаций t символов (представляющих 20 аминокислот и символ пробела). Интуитивно мы должны награждать более консервативные столбцы более высоким счетом. Дополнительные сведения см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Счет множественных выравниваний**.

Задача множественного выравнивания: *найдите выравнивание с наивысшей счетом между несколькими строками в заданной матрице счета.*

Input: набор t строк и t-мерная матрица *Score*.

Output: множественное выравнивание этих строк, счет которых (определенный матрицей *Score*) максимален среди всех возможных выравниваний этих строк.

Простой алгоритм динамического программирования, примененный к t-мерному графу выравнивания, решает задачу множественного выравнивания для t строк. Для трех последовательностей v, ω и u мы определяем $s_{i,j,k}$ как длину самого длинного пути от источника (0, 0, 0) до узла (i, j, k) в графе выравнивания. Повторение для $s_{i,j,k}$ в трехмерном случае аналогично повторению для попарного выравнивания:

$$s_{i,j,k} = \max \begin{cases} s_{i-1,j,k} & + score(v_i, -, -) \\ s_{i,j-1,k} & + score(-, w_j, -) \\ s_{i,j,k-1} & + score(-, -, u_k) \\ s_{i-1,j-1,k} & + score(v_i, w_j, -) \\ s_{i-1,j,k-1} & + score(v_i, -, u_k) \\ s_{i,j-1,k-1} & + score(-, w_j, u_k) \\ s_{i-1,j-1,k-1} & + score(v_i, w_j, u_k) \end{cases}.$$

В случае t последовательностей длины n граф выравнивания состоит примерно из n^t узлов, и каждый узел имеет до $2^t - 1$ входящих ребер, что дает общее время выполнения $O(n^t \cdot 2^t)$. С ростом t алгоритм динамического программирования становится нецелесообразным. Для устранения этого узкого места алгоритма неоптимального множественного выравнивания было предложено много вариантов эвристики.

В **задаче самой длинной общей подпоследовательности при множественном выравнивании** счет столбца матрицы выравнивания равен 1, если все символы столбца идентичны, и 0, если хотя бы один символ не совпадает.

Жадный алгоритм множественного выравнивания

Обратите внимание, что множественное выравнивание

```
AT-GTTaTA
AgCGaTC-A
ATCGT-CTc
```

индуцирует три попарных выравнивания:

```
AT-GTTaTA      AT-GTTaTA      AgCGaTC-A
AgCGaTC-A      ATCGT-CTc      ATCGT-CTc
```

Но можем ли мы работать в обратном направлении, объединяя оптимальные парные выравнивания в множественное выравнивание?

ОСТАНОВИТЕСЬ и задумайтесь.

1. Делает ли оптимальное множественное выравнивание оптимальное парное выравнивание?
2. Попробуйте объединить приведенные ниже попарные выравнивания в множественное выравнивание строк CCCCTTTT, TTTTGGGG и GGGGCCCC.

```
CCCCTTTT----      ----CCCCTTTT      TTTTGGGG----
----TTTTGGGG      GGGGCCCC----      ----GGGGCCCC
```

К сожалению, мы не всегда можем объединить оптимальное парное выравнивание в множественное, потому что некоторые парные выравнивания могут быть несовместимы (как показано тремя парными выравниваниями выше).

Первое парное выравнивание подразумевает, что CCCC предшествует TTTT в множественном выравнивании, построенном из этих трех парных выравниваний. Третье попарное выравнивание подразумевает, что TTTT предшествует GGGG в множественном выравнивании. Но второе попарное выравнивание подразумевает, что GGGG стоит перед CCCC в множественном выравнивании. Таким образом, CCCC должно предшествовать TTTT, которое должно предшествовать GGGG, которое должно предшествовать CCCC, что вступает в противоречие.

Чтобы избежать несовместимости, некоторые алгоритмы множественного выравнивания пытаются построить множественное выравнивание из попарных жадным способом, что не всегда оптимально. Жадная эвристика начинается с выбора двух строк, имеющих наивысший счет попарного выравнивания (среди всех возможных пар строк), а затем использует это попарное выравнивание в качестве строительного блока для итеративного добавления по одной строке за раз к растущему множественному выравниванию. Мы выравниваем две ближайшие строки на первом этапе, потому что они часто дают наилучшие шансы построить надежное множественное выравнивание. По той же причине затем мы выбираем строку с максимальным счетом по сравнению с текущим выравниванием на каждом этапе. Но что означает выравнивание строки относительно выравнивания других строк?

Выравнивание нуклеотидных последовательностей с k столбцами может быть представлено в виде матрицы профиля размерностью $4 \times k$, подобной показанной ниже, которая содержит частоты нуклеотидов из каждого столбца (выравнивания аминокислот представлены матрицами профиля $20 \times k$). Эври-

стика жадного множественного выравнивания добавляет строку к текущему выравниванию, создавая попарное выравнивание между строкой и профилем текущего выравнивания. В результате задача построения множественного выравнивания t последовательностей сводится к построению $t - 1$ попарного выравнивания.

ОСТАНОВИТЕСЬ и задумайтесь. Разработайте алгоритм для выравнивания строки по матрице профиля. Как бы вы сосчитали столбцы в таком выравнивании?

Выравнивание

T	C	G	G	G	–	g	T	T	T	t	t
c	C	–	–	t	G	A	c	T	T	a	C
a	C	G	–	G	G	A	T	T	T	t	C
T	t	G	G	G	–	A	c	T	T	t	t
a	–	–	–	G	–	–	–	T	–	C	–
T	t	G	G	G	G	A	c	T	T	C	C
T	C	G	–	–	G	A	T	T	c	a	t
–	–	–	G	G	G	A	T	T	c	C	–
T	a	G	G	G	G	A	a	c	–	–	C
T	C	G	G	G	t	A	T	a	a	C	C

Профиль

A:	.2	.1	0	0	0	0	.8	.1	.1	.1	.2	0
C:	.1	.5	0	0	0	0	0	.3	.1	.2	.4	.5
G:	0	0	.7	.6	.8	.6	.1	0	0	0	0	0
T:	.6	.2	0	0	.1	.1	0	.5	.8	.6	.2	.3

Рис. 5.65 Матрица профиля для множественного выравнивания 10 последовательностей. Сумма каждого столбца матрицы профиля равна 1 минус частота символа пробела. Наиболее часто встречающиеся нуклеотиды в каждом столбце показаны цветными заглавными буквами

Хотя жадные алгоритмы множественного выравнивания хорошо работают для похожих последовательностей, их производительность ухудшается для несходных последовательностей, поскольку жадные методы могут быть введены в заблуждение ложным попарным выравниванием. Если первые две последовательности, выбранные для построения множественного выравнивания, выровнены таким образом, что это несовместимо с оптимальным множественным выравниванием, то ошибка в этом начальном попарном выравнивании будет повторяться на всем пути до окончательного множественного выравнивания.

Узнав, как выравнивать несколько последовательностей, вы теперь готовы для решения сложной задачи, сравнимой с задачей, с которой Марахиел и его сотрудники столкнулись в 1999 году.

Заключительная задача. В 1999 году Марахиел получил нерибосомный код только для 14 из 20 протеиногенных аминокислот (Ala, Asn, Asp, Cys, Gln, Glu, Ile, Leu, Phe, Pro, Ser, Thr, Tyr, Val), потому что у него отсутствовали последовательности А-домена для оставшихся шести аминокислот. Теперь, когда доступно гораздо больше А-доменов, у вас есть возможность заполнить пробелы в оригинальной работе Марахиела. Постройте множественное выравнивание 397 А-доменов, выявите консервативные столбцы в этом выравнивании и сделайте наилучшее предположение о сигнатурах, кодирующих все 20 протеиногенных аминокислот, на основе выравнивания Марахиела.

☞ **Загрузить данные А-домена (формат .csv)**

Сопутствующие материалы

Светлячки и нерибосомный код

Когда Марахиел в конце 1990-х годов начал свою новаторскую работу по расшифровке нерибосомного кода, 160 А-доменов уже были секвенированы, а аминокислоты, которые они кодируют, были экспериментально идентифицированы. Однако до сих пор неясно, как именно каждый А-домен кодирует конкретную аминокислоту.

Мы показали выравнивание трех интервалов А-доменов в основном тексте главы, но Марахиел выровнял все 160 идентифицированных А-доменов, чтобы выявить консервативное ядро. Тем не менее все еще оставалось неясным, какие столбцы в этом выравнивании определяли нерибосомные сигнатуры.

Помощь пришла от необычного союзника – *Photinus pyralis*, самого распространенного вида светлячков (рис. 5.66). Светлячки вырабатывают фермент под названием **люцифераза**, который помогает им выдавать свет для привлечения партнеров ночью. Разные виды имеют разные модели физиологии свечения, и самки реагируют на самцов одного и того же вида, реагируя на цвет, продолжительность и интенсивность их свечения.

Рис. 5.66 Светлячок *Photinus pyralis*

Какое отношение светлячки имеют к нерибосомному коду? Люцифераза светлячка принадлежит к классу ферментов, называемых **аденилатобразу-**

ющими ферментами, которые имеют сходство с доменами аденилирования. Вот почему, когда в 1996 году Питер Брик опубликовал трехмерную структуру люциферазы светлячка, Марахиел сразу обратил на это внимание. В 1997 году он и Брик объединили усилия и использовали люциферазу светлячка в качестве каркаса для реконструкции первой трехмерной структуры А-домена, кодирующего фенилаланин (Phe). Стоит отметить, что этот А-домен принадлежал NRP-синтетазе, кодирующей *Gramicidin Soviet*, первый антибиотик массового производства.

Марахиел и Брик на самом деле построили трехмерную структуру более крупного комплекса, содержащего как А-домен, так и Phe. Эта трехмерная структура предоставила информацию об аминокислотных остатках в А-домене, расположенном рядом с Phe, гипотетически **активным карманом** А-домена. Кроме того, Марахиел продемонстрировал экспериментально и с помощью вычислений, что аминокислоты в этом активном кармане определяют нерибосомный код, что привело к появлению 8 фиолетовых столбцов в трехстороннем выравнивании, которое мы показали в начале главы.

На рис. 5.67 показан частичный нерибосомный код, полученный Марахиелом. Хотя ему удалось вывести некоторые сигнатуры, нерибосомный код очень **избыточен**, а это означает, что все несколько мутировавших вариантов сигнатуры кодируют одну и ту же аминокислоту. Для некоторых аминокислот эта избыточность ярко выражена; например, Марахиел идентифицировал три очень разные сигнатуры AWMFAAVL, AFWIGGTF и FESTAAVY, кодирующие Val.

Аминокислота	Сигнатура	Аминокислота	Сигнатура
Ala	LLFGIAVL	Leu	AFMLGMVF
Asn	LTKLGEVG	Orn	MENLGLIN
Asp	LTKVGHIG	Orn	VGEIGSID
Cys	HESDVGIT	Phe	AWTIAAVC
Cys	LYNLSLIW	Pro	VQLIAHVV
Gln	AQDLGVVD	Ser	VWHLSLID
Glu	AWHFGGVD	Thr	FWNIGMVH
Glu	AKDLGVVD	Tyr	GTITAEVA
Ile	GFFLGVVY	Tyr	ALVTGAVV
Ile	AFFYGITF	Tyr	ASTVAAVC
Leu	AWFLGNVV	Val	AFWIGGTF
Leu	AWLYGAVM	Val	FESTAAVY
Leu	GAYTGEVV	Val	AWMFAAVL

Рис. 5.67 Частичный нерибосомный код Марахиела. Некоторые протеиногенные аминокислоты отсутствуют в этой таблице, потому что их не было в наборе данных Марахиела

Поиск LCS[1] *без постройки города*

Определим ***i*-префикс** строки как подстроку, образованную ее первыми буквами *i*, и ***j*-суффикс** строки как подстроку, образованную ее последними бук-

[1] LCS: Самая длинная общая подпоследовательность. – *Прим. ред.*

вами j. Кроме того, для заданных строк v и ω пусть $LCS_{i,j}$ обозначает LCS между i-префиксом v и j-префиксом ω, и пусть $s_{i,j}$ будет длиной $LCS_{i,j}$. По определению $s_{i,0} = s_{0,j} = 0$ для всех значений i и j. Далее, $LCS_{i,j}$ может содержать как v_i, так и ω_j, и в этом случае эти символы совпадают, а $LCS_{i,j}$, j расширяет LCS более коротких префиксов $v_1 \ldots v_{i-1}$ и $\omega_1 \ldots \omega_{j-1}$. В противном случае либо v_i, либо ω_j отсутствует в $LCS_{i,j}$. Если v_i отсутствует в $LCS_{i,j}$, то это LCS, следовательно, также является LCS из $v_1 \ldots v_{i-1}$ и $\omega_1 \ldots \omega_j$. Аналогичный аргумент применим к случаю, когда ω_j отсутствует в $LCS_{i,j}$. Таким образом, $s_{i,j}$ удовлетворяет следующей рекуррентности, той же, которую мы получили в основном тексте, используя манхэттенскую сетку.

$$s_{i,j} = \max \begin{cases} s_{i-1,j} & +0 \\ s_{i,j-1} & +0 \\ s_{i-1,j-1} & +1, \text{ если } v_i = w_j \end{cases}.$$

Построение топологической сортировки

Первые приложения топологического упорядочения возникли в результате крупных управленческих проектов, в которых пытались запланировать последовательность задач на основе их зависимостей (например, «Задача порядка одевания»). В этих проектах задачи представлены узлами, а ребро соединяет узел a с узлом b, если задача a должна быть завершена до того, как можно будет запустить задачу b.

ОСТАНОВИТЕСЬ и задумайтесь. Докажите, что в каждом DAG есть узел без входящих ребер и узел без исходящих ребер.

Следующий алгоритм построения топологической сортировки основан на наблюдении, что каждый DAG имеет по меньшей мере один узел без входящих ребер. Мы пометим один из этих узлов как v_1, а затем удалим этот узел из графа вместе со всеми его исходящими ребрами. Результирующий граф также является DAG, который, в свою очередь, должен иметь узел без входящих ребер; мы помечаем этот узел v_2 и снова удаляем его из графа вместе с его исходящими ребрами. Результирующий алгоритм работает до тех пор, пока не будут удалены все узлы, создавая топологический порядок $\omega_1 \ldots \omega_n$. Время выполнения этого алгоритма пропорционально количеству ребер во входном DAG.

```
TopologicalOrdering(Graph)
  List ← пустой список
  Candidates ← набор всех узлов в Graph, не содержащих входных ребер
  while набор Candidates не пустой
    выберите произвольный узел a из Candidates
```

```
    добавьте a к концу списка List и удалите a из Candidates
    for каждого выходящего ребра из a до другого узла b
      удалить ребро (a, b) из Graph
      if b не имеет других входящих ребер
        добавьте b к Candidates
  if Graph имеет ребра, которые еще не были удалены
    return «входящий граф не является графом DAG»
  else
    return List
```

Матрица счета PAM[1]

Мутации нуклеотидной последовательности гена часто изменяют аминокислотную последовательность транслируемого белка. Некоторые из этих мутаций нарушают способность белка функционировать, что делает их редкими событиями в молекулярной эволюции. Asn, Asp, Glu и Ser являются наиболее «мутабельными» аминокислотами, а Cys и Trp – наименее мутабельными. Знание вероятности каждой возможной мутации позволяет биологам строить матрицы счета аминокислот для биологически обоснованных выравниваний последовательностей, в которых разные замены штрафуются по-разному. Запись матрицы счета аминокислот *score*(*i, j*) обычно отражает, как часто *i*-я аминокислота заменяет *j*-ю аминокислоту в выравниваниях последовательностей родственных белков. В результате оптимальные выравнивания аминокислотных последовательностей могут иметь очень мало совпадений, но по-прежнему представляют собой биологически адекватные выравнивания.

Как биологи узнают, какие мутации более вероятны, чем другие? Если нам известен большой набор попарных выравниваний *родственных последовательностей* (например, имеющих не менее 90 % общих аминокислот), то вычисление *score*(*i, j*) основано на подсчете того, сколько раз выравниваются соответствующие аминокислоты. Однако нам нужно заранее знать матрицу счета, чтобы построить этот набор начальных выравниваний, – уловка-22!

К счастью, правильное выравнивание очень похожих последовательностей настолько очевидно, что его можно построить даже с помощью примитивной схемы оценки, не учитывающей разную склонность к мутациям (например, +1 для совпадений и –1 для несовпадений и вставок), тем самым разрешая головоломку. После построения этих очевидных выравниваний мы можем использовать их для вычисления новой матрицы счета, которую мы можем использовать итеративно для формирования все менее и менее очевидных выравниваний.

Это упрощенное описание скрывает некоторые детали. Например, вероятность мутации Ser в Phe у видов, дивергировавшихся 1 млн лет назад, меньше,

[1] PAM, «Point Accepted Mutation», или «точечная принятая мутация», – это замена одной аминокислоты в первичной структуре белка другой аминокислотой, которая принимается процессами естественного отбора. – *Прим. ред.*

чем вероятность той же мутации у видов, дивергировавшихся 100 млн лет назад. Это наблюдение подразумевает, что матрицы счета для сравнения белков должны зависеть от сходства организмов и скорости эволюции интересующих белков. На практике белки, которые биологи используют для создания начального выравнивания, очень похожи, поскольку в них сохранено 99 % аминокислот (например, большинство белков, общих для человека и шимпанзе). Последовательности, сходные на 99%, считаются расходящимися на одну **единицу PAM** («PAM» – «точечная принятая мутация»). Вы можете думать о единице PAM как о количестве времени, в течение которого у «среднего» белка мутирует 1 % аминокислот.

Матрица счета PAM1 определяется следующим образом из множества парных выравниваний 99 % сходных белков. Для заданного набора попарных выравниваний пусть $M(i, j)$ будет числом раз, когда i-я и j-я аминокислоты появляются в одном и том же столбце, деленным на общее количество раз, когда i-я аминокислота появляется во всех последовательностях. Пусть $f(j)$ будет частотой j-й аминокислоты в последовательностях или количеством раз, когда она появляется во всех последовательностях, деленным на общую длину двух последовательностей. (i, j^*)-й элемент матрицы PAM1 определяется как:

$$\log[M(i, j) / f(j)].$$

Для большего количества единиц n матрица PAMn вычисляется на основе наблюдения, что матрица Mn (результат умножения M саме на себя n раз) содержит эмпирические вероятности того, что одна аминокислота мутирует в другую в течение n единиц PAM. Таким образом, (i, j)-й элемент матрицы PAMn задается выражением:

$$\log[M^n(i, j) / f(j)].$$

Это приближение предполагает, что частоты встречаемости аминокислот $f(j)$ остаются постоянными во времени, а мутационные процессы в интервале одной единицы PAM последовательно протекают в течение длительных периодов времени. Для больших n матрицы счета PAM часто позволяют нам находить родственные белки, даже если в выравнивании мало совпадений.

Алгоритмы «разделяй и властвуй»

Мы будем использовать задачу сортировки списка целых чисел в качестве примера алгоритма «разделяй и властвуй». Начнем с задачи объединения, в которой мы хотим объединить два отсортированных списка $List_1$ и $List_2$ в один. Приведенный ниже алгоритм слияния объединяет два отсортированных списка в один отсортированный список за время $O(|List_1| + |List_2|)$, итеративно выбирая наименьший оставшийся элемент в списках $List_1$ и $List_2$ и перемещая его в растущий отсортированный список.

```
Merge(List1, List2)
    SortedList ← пустой список
    while и List1 и List2 не пустые одновременно
        if пока самый маленький элемент List1 меньше самого маленького
        элемента в List2
            move самый маленький элемент из List1 в конец SortedList
        else
            переместить List2 в конец SortedList
    переместить любой оставшийся элемент в списках List1 и List2 в конец
    SortedList
    return SortedList
```

Рис. 5.68. Процесс объединения отсортированных списков (2, 5, 7, 8) и (3, 4, 6) в отсортированный список (2, 3, 4, 5, 6, 7, 8)

Объединение **Merge** было бы полезно для сортировки произвольного списка, если бы мы знали, как разделить произвольный (несортированный) список на два уже отсортированных списка половинного размера. Однако может показаться, что это вернуло нас к тому, с чего мы начали, за исключением того, что теперь нам нужно сортировать два меньших списка вместо одного большого. Тем не менее сортировка двух меньших списков является предпочтительной вычислительной задачей. Чтобы понять почему, давайте рассмотрим алгоритм **Mergesort**, который **делит** несортированный список на две части, а затем рекурсивно решает каждую меньшую задачу сортировки перед объединением отсортированных списков.

```
Mergesort(List)
    if список List состоит из единственного элемента
        return List
    FirstHalf ← первая половина списка List
    SecondHalf ← вторая половина списка List
    SortedFirstHalf ← Mergesort(FirstHalf)
    SortedSecondHalf ← Mergesort(SecondHalf)
    SortedList ← Merge(SortedFirstHalf, SortedSecondHalf)
    return SortedList
```

ОСТАНОВИТЕСЬ и задумайтесь. Каково время выполнения **Mergesort**?

На рисунке ниже показано **дерево рекурсии** сортировки **Mergesort**, состоящее из уровней $\log_2(n)$, где n – размер исходного неотсортированного списка. На нижнем уровне мы должны объединить два отсортированных списка примерно из $n/2$ элементов в каждом, что требует времени $O(n/2 + n/2) = O(n)$. На следующем высшем уровне мы должны объединить четыре списка из $n/4$ элементов, что потребует $O(n/4 + n/4 + n/4 + n/4) = O(n)$ времени. Этот паттерн можно обобщить: i-й уровень содержит 2^i списков, каждый из которых содержит приблизительно $n/2^i$ элементов и требует $O(n)$ времени для объединения. Поскольку в дереве рекурсии есть уровни $\log_2(n)$, сортировка слиянием требует времени выполнения $O(n \cdot \log_2(n))$ в целом, что обеспечивает ускорение по сравнению с наивными алгоритмами сортировки $O(n^2)$.

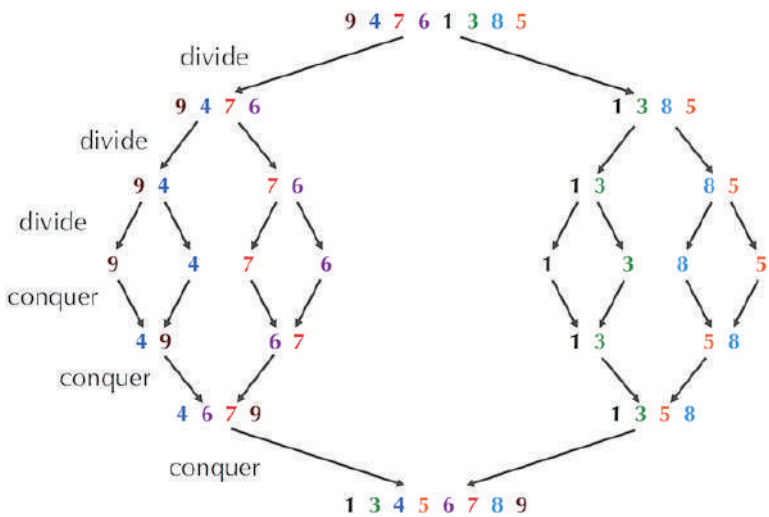


Рис. 5.69 Дерево рекурсии для сортировки списка из восьми элементов с помощью **Mergesort**. Шаги разделения *divide* (верхние) состоят из $\log_2(8) = 3$ уровней, где входной список разбивается на меньшие и меньшие подсписки. Шаги объединения *conquer* (нижние) состоят из того же количества уровней, поскольку отсортированные подсписки снова объединяются

Счет множественных выравниваний

Выбор оценочной функции может существенно повлиять на качество множественного выравнивания. В основном тексте главы мы описали способ счета выравнивания по t-путям с помощью t-мерной матрицы счета. Ниже мы опишем более практические методы оценки выравниваний.

Столбцы t-образного выравнивания описывают путь в t-мерном графе выравнивания, веса ребер которого определяются оценочной функцией. Используя статистически мотивированный показатель энтропии, счет множествен-

ного выравнивания определяется как сумма энтропий ее столбцов. Напомним из нашей главы о поиске регуляторных мотивов, что энтропия столбца равна

$$-\sum p_x \cdot \log_2 p_x,$$

где сумма берется по всем символам x, присутствующим в столбце, а p_x – это частота символа x в столбце.

Изучая мотивы в предыдущей главе, мы увидели, что более консервативные столбцы будут иметь более низкие показатели энтропии. Поскольку мы хотим максимизировать счет выравнивания, мы используем отрицательную энтропию, чтобы гарантировать, что более консервативные столбцы получат более высокий счет. Таким образом, поиск самого длинного пути в t-мерном графе выравнивания соответствует нахождению множественного выравнивания с минимальной энтропией.

Упражнение. Вычислите показатель энтропии трех-стороннего выравнивания Марахиела, воспроизведенного ниже.

```
1: YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTA
2: -AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI
3: IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA

    1: SFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
    2: -YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
    3: ----PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
```

Еще одним популярным методом оценки является счет суммы пар **Sum-of-Pairs score (SP-score)**. Множественное выравнивание *Alignment* для t последовательностей индуцирует попарное выравнивание между i-й и j-й последовательностями с показателем $s(Alignment, i, j)$. SP-счет для множественного выравнивания просто суммирует счета каждого индуцированного попарного выравнивания:

$$SP\text{-}Score(Alignment) = \sum_{1 \le i \le j \le t} s(Alignment, i, j).$$

Упражнение. Подсчитайте SP-счет трехстороннего выравнивания Марахиела, воспроизведенного ниже.

```
1: YAFDLGYTCMFPVLLGGGELHIVQKETYTAPDEIAHYIKEHGITYIKLTPSLFHTIVNTA
2: -AFDVSAGDFARALLTGGQLIVCPNEVKMDPASLYAIIKKYDITIFEATPALVIPLMEYI
3: IAFDASSWEIYAPLLNGGTVVCIDYYTTIDIKALEAVFKQHHIRGAMLPPALLKQCLVSA

    1: SFAFDANFESLRLIVLGGEKIIPIDVIAFRKMYGHTE-FINHYGPTEATIGA
    2: -YEQKLDISQLQILIVGSDSCSMEDFKTLVSRFGSTIRIVNSYGVTEACIDS
    3: ----PTMISSLEILFAAGDRLSSQDAILARRAVGSGV-Y-NAYGPTENTVLS
```

Библиографические примечания

Расстояние редактирования было введено Levenshtein, 1966[1]. Алгоритм локального выравнивания, описанный в тексте, был предложен Smith and Waterman, 1981[2]. Doolittle et al., 1983[3], выявили сходство между онкогенами и PDGF, они не знали об алгоритме Смита–Уотермана. Трехмерные структуры люциферазы светлячка и А-домена из *Bacilus brevis* были опубликованы Conti, Franks, and Brick, 1996[4], а также Conti et al., 1997[5]. Нерибосомный код был впервые представлен Stachelhaus, Mootz, and Marahiel, 1999[6].

[1] https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.
[2] https://www.ncbi.nlm.nih.gov/pubmed/7265238.
[3] https://www.ncbi.nlm.nih.gov/pubmed/6304883.
[4] https://www.ncbi.nlm.nih.gov/pubmed/8805533.
[5] https://www.ncbi.nlm.nih.gov/pubmed/9250661.
[6] https://www.ncbi.nlm.nih.gov/pubmed/10421756.

Глава 6

Есть ли в человеческом геноме «хрупкие» области?

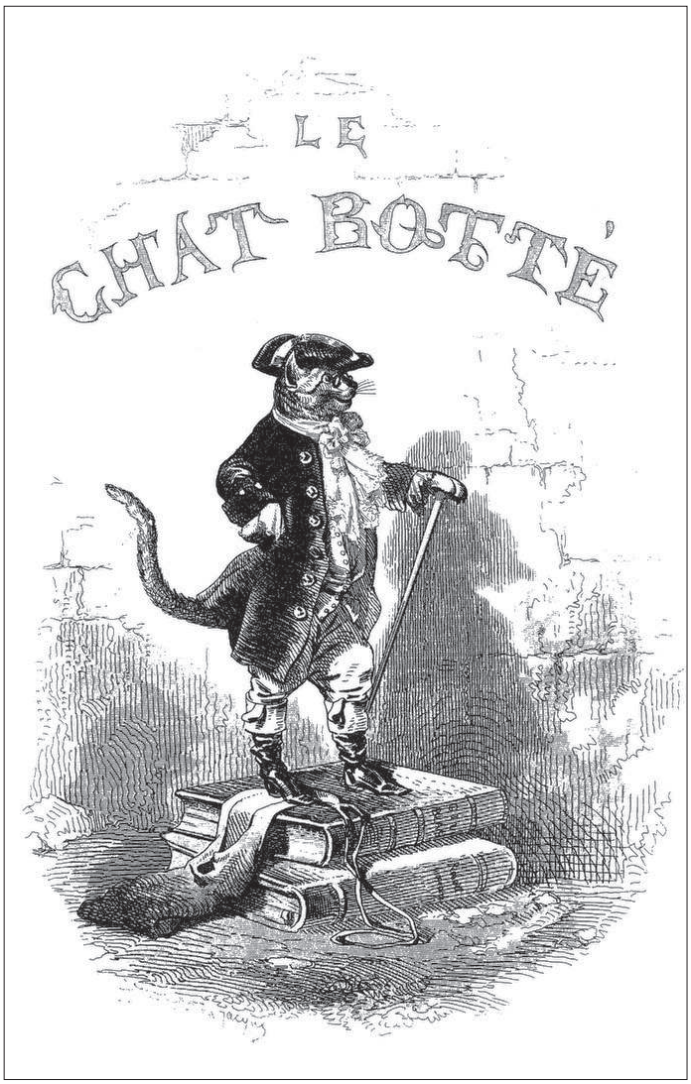
Комбинаторные алгоритмы

О мышах и людях

«Мне еще сказали, – сказал кот, – что вы также можете превратиться в самое маленькое животное, например в крысу или мышь. Но я с трудом могу в это поверить. Должен признаться вам, что я думаю, что это было бы совершенно невозможно.

«Невозможно?!» – воскликнул великан-людоед. – «Вот увидишь!»

Он тут же превратился в мышь и начал бегать по полу. Как только кот увидел это, он набросился на мышь и съел ее.



Насколько различаются геномы человека и мыши?

Когда Шарль Перро описал превращение людоеда в мышь в «Коте в сапогах», он вряд ли мог предположить, что спустя три столетия исследования покажут, что геномы человека и мыши удивительно похожи. Почти у каждого человеческого гена есть мышиный аналог, хотя мыши значительно превосходят нас, когда речь идет об обонятельных генах, отвечающих за обоняние. По сути, мы мыши без хвостов – у нас даже есть гены, необходимые для создания хвоста, но эти гены «замолчали» в ходе нашей эволюции.

Мы начали со сказочного вопроса: «Как великан-людоед может превратиться в мышь?» Поскольку у нас с мышами большая часть общих генов, мы теперь задаем вопрос об эволюции млекопитающих: «Какие эволюционные силы преобразовали геном предка человека и мыши в современные геномы человека и мыши?»

Если бы не по годам развитый ребенок вырос, читая сказки, и захотел узнать о том, чем отличаются геномы человека и мыши, то вот что мы бы ему сказали. Вы можете разрезать 23 хромосомы человека на 280 частей, перетасовать эти фрагменты ДНК, а затем склеить части вместе в новом порядке и получить 20 хромосом мыши. Правда, однако, в том, что эволюция не использует драматическоих операций вырезания и вставки; вместо этого она применяет более мелкие изменения, известные как **рекомбинации генома**, которыми мы и займемся в этой главе.

К сожалению, наша биоинформационная машина времени не перенесет нас более чем на несколько столетий в прошлое. Если бы это было так, мы могли бы отправиться на 75 млн лет назад во времени, наблюдая, как люди медленно превращаются в маленьких пушистых животных, живших по соседству с динозаврами. Затем мы могли бы вернуться в настоящее, наблюдая за тем, как это животное эволюционировало в мышь. В этой главе мы надеемся определить рекомбинации генома, которые разделили геномы человека и мыши, без необходимости переделывать нашу машину времени.

Синтенные блоки

Чтобы упростить сравнение геномов, сначала сосредоточимся на Х-хромосоме, которая является одной из двух определяющих пол хромосом у млекопитающих и сохранила почти все свои гены на протяжении всей эволюции млекопитающих. Таким образом, мы можем рассматривать Х-хромосому как «мини-геном» при сравнении мышей и людей, поскольку гены этой хромосомы не прыгали по разным хромосомам (и наоборот). Для получения дополнительной информации см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Почему содержание генов Х-хромосом млекопитающих настолько консервативно?**.

Оказывается, не только то, что большинство генов человека имеет мышиные аналоги, но и то, что сотни подобных генов часто выстраиваются один за другим в одном и том же порядке в геномах двух видов. Каждый из 11 цвет-

ных сегментов на рисунке ниже представляет такую процессию подобных генов; эти процессии называются **синтенными блоками**. Позже мы объясним, как строить синтенные блоки и что означают левое и правое **направления** блоков.

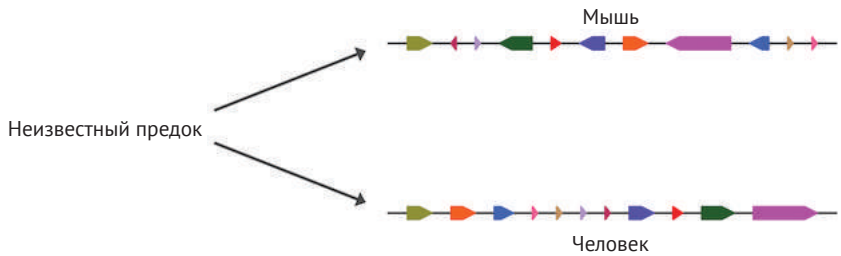


Рис. 6.1 Х-хромосомы мыши и человека, представленные в виде 11 цветных направленных сегментов (синтенных блоков)

Синтенные блоки упрощают сравнение Х-хромосом мыши и человека примерно со 150 млн пар оснований до 11 единиц. Это упрощение аналогично сравнению двух похожих фотографий. Если мы будем сравнивать изображения по одному пикселю за раз, мы можем быть ошеломлены масштабом проблемы; вместо этого нам нужно уменьшить масштаб, чтобы заметить паттерны более высокого уровня. Не случайно биологи используют термин «разрешение» для обсуждения уровня, на котором анализируются геномы.

Реверсии

Вы, наверное, задавались вопросом, как меняется геном, когда он подвергается перестройке. Геномные рекомбинации были обнаружены 90 лет назад, когда Альфред Стёртевант изучал мутантов плодовых мушек с алыми и персиковыми глазами, а также с дельтовидными крыльями неправильной формы. Стёртевант проанализировал гены, кодирующие эти признаки, названные **алый**, **персиковый** и **дельта**, и был поражен, обнаружив, что расположение этих генов у *Drosophila melanogaster* (алый, персиковый, дельта) отличается от их расположения у *Drosophila simulans*. Он немедленно предположил, что хромосомный сегмент, содержащий **персиковый** и **дельту**, должен был быть перевернут. Для получения дополнительной информации см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Открытие рекомбинаций генома**.

Стёртевант был свидетелем наиболее распространенной формы рекомбинации генома, называемой **реверсией**, которая переворачивает часть ДНК-последовательности хромосомы и инвертирует направление любых синтенных блоков в пределах этого участка.

На рисунке ниже показана серия из семи реверсий, превращающих Х-хромосому мыши в Х-хромосому человека.

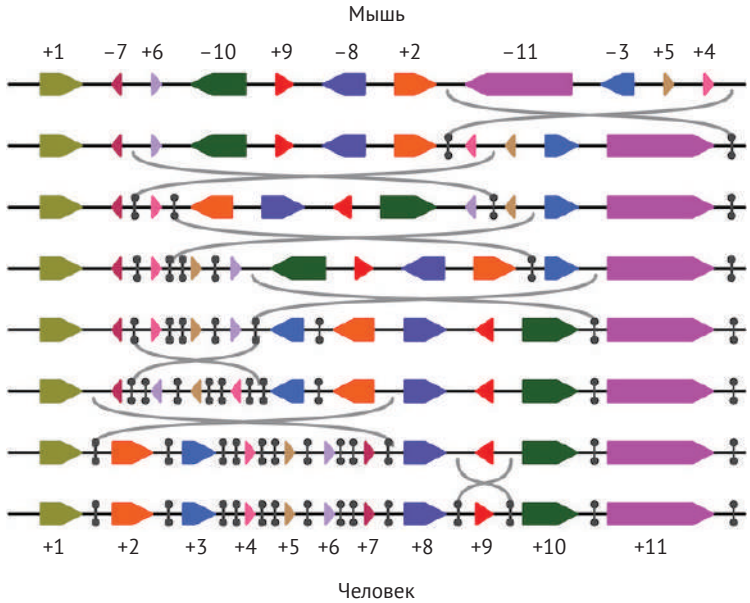


Рис. 6.2 Трансформация Х-хромосомы мыши в Х-хромосому человека в серии семи реверсий. Каждый синтенный блок имеет уникальный цвет и помечен целым числом от 1 до 11; положительный или отрицательный знак каждого целого числа указывает направление синтенного блока (указывая вправо или влево соответственно). Два коротких вертикальных сегмента отмечают крайние точки инверсированного интервала в каждой реверсии. Предположим, что этот эволюционный сценарий верен и что, скажем, пятая структура синтенного блока сверху представляет собой истинную наследственную структуру. Затем произошли первые четыре поворота на эволюционном пути от мышей к общему предку человека и мыши (путешествие назад во времени), а последние три поворота произошли на эволюционном пути от общего предка к человеку (путешествие вперед во времени). В этой главе нас не интересует реконструкция генома предков и, следовательно, нас не интересует, движется ли определенное обращение назад или вперед во времени

Если этот сценарий верен, то Х-хромосома общего предка человека и мыши должна быть представлена одним из порядков промежуточных синтенных блоков. К сожалению, эта серия из семи реверсий предлагает только один из 1070 различных семиэтапных сценариев трансформации Х-хромосомы мыши в Х-хромосому человека. Мы понятия не имеем, какой сценарий является правильным, и даже не уверены, что правильный сценарий – это именно семь реверсий.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы преобразовать Х-хромосому мыши в Х-хромосому человека, используя всего шесть реверсий?

Независимо от того, сколько реверсий разделяет Х-хромосомы человека и мыши, реверсии должны быть редкими геномными событиями. Действительно, рекомбинации генома обычно вызывают гибель или бесплодие мутировавшего организма, тем самым препятствуя передаче рекомбинации следующему поколению. Однако крошечная доля рекомбинаций генома может оказать положительное влияние на выживание и размножение вида в результате естественного отбора. Когда популяция оказывается изолированной от остальных популяций своего вида на достаточно долгое время, рекомбинация даже может создать новый вид.

Точки перестановки

Геология предлагает наводящую на размышления аналогию для размышлений об эволюции генома. Вам может понравиться думать о рекомбинациях генома как о «геномных землетрясениях», которые резко меняют хромосомную архитектуру организма. Геномные рекомбинации контрастируют с гораздо более частыми точечными мутациями, которые действуют медленно и аналогичны «геномной эрозии».

Вы можете визуализировать реверсию как разрыв генома с обеих сторон хромосомного интервала, переворачивание интервала и последующее склеивание полученных сегментов в новом порядке. Имея в виду, что землетрясения чаще происходят вдоль линий разломов, мы задаемся вопросом, действует ли аналогичный принцип для реверсий, – происходят ли они снова и снова в одних и тех же областях генома? Фундаментальный вопрос в исследованиях эволюции хромосом заключается в том, возникают ли **точки разрыва (разреза)** реверсий (т. е. концы инвертированных интервалов) вдоль «линий разлома», называемых **горячими точками рекомбинации**. Если такие горячие точки в геноме человека существуют, мы хотим найти их и определить, как они могут быть связаны с генетическими нарушениями, связанными с рекомбинациями.

Конечно, следует строго определить, что мы подразумеваем под «горячей точкой рекомбинации». Повторно исследуя сценарий с семью реверсиями, превращающий Х-хромосому мыши в Х-хромосому человека (рис. 6.2), мы обозначаем конечные точки каждой реверсии с помощью вертикальных сегментов. Области, затронутые множественными реверсиями, обозначены несколькими вертикальными сегментами в Х-хромосоме человека. Например, область, прилегающая к заостренной стороне блока 3, используется как конечная точка четвертой и пятой реверсий. Поэтому мы разместили две вертикальные линии между блоками 3 и 4 в Х-хромосоме человека. Однако тот факт, что мы указали две точки разрыва в этой области, не означает, что эта область является горячей точкой рекомбинации, поскольку приведенные в рис 6.2 реверсии представляют собой лишь один возможный эволюционный сценарий. Поскольку истинный сценарий рекомбинации неизвестен, не сразу ясно, как мы можем определить, существуют ли горячие точки рекомбинации вообще.

Модель эволюции хромосом со случайными разрывами

В 1973 году Сусуму Оно предложил модель эволюции хромосом со **случайными разрывами (разрезами)**. Эта гипотеза утверждает, что точки разрыва рекомбинаций возникают случайным образом, что означает, что горячих точек рекомбинации в геномах млекопитающих не существует. Тем не менее модели Оно, когда она была представлена, не хватало подтверждающих ее доказательств. В конце концов, как мы можем определить, существуют ли горячие точки рекомбинации, не зная точной последовательности рекомбинаций, разделяющих два вида?

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующие вопросы.

1. Предположим, что серия случайных реверсий приводит к одному огромному синтенному блоку, покрывающему 90 % генома, в дополнение к 99 крошечным синтенным блокам, покрывающим оставшиеся 10 % генома. Стоит ли нам удивляться?
2. Что, если в результате случайных реверсий получится 100 синтенных блоков примерно одинаковой длины? Стоит ли нам удивляться?

Идея, которую мы хотели бы донести до вас в этих вопросах, заключается в том, что мы можем протестировать модель случайных разрывов, анализируя распределение длин синтенных блоков. Например, длина синтенных блоков человека и мыши на Х-хромосоме сильно различается, при этом самый большой блок (блок 11, показанный на рис. 6.2) занимает почти 25 % всей длины Х-хромосомы. Соответствует ли это изменение длины синтенного блока модели случайных разрывов?

В 1984 году Джозеф Надо и Бенджамин Тейлор задались вопросом, какой должна быть ожидаемая длина синтенных блоков после N реверсий, происходящих в случайных местах генома. Если исключить маловероятное событие, когда две случайные реверсии разрезают хромосому точно в одном и том же месте, то N случайных реверсий разрезают хромосому в $2N$ местах и производят $2N + 1$ синтенный блок. На рисунке ниже показан результат вычислительного эксперимента, в котором 320 случайных реверсий применяются к симулированной хромосоме, состоящей из 25 000 генов, что дает $2 \cdot 320 + 1 = 641$ синтенный блок. Средний размер синтенного блока составляет $25\,000/641 \approx 34$ гена, но это не означает, что все синтенные блоки должны иметь примерно 34 гена. Если мы выберем случайные места для точек разрыва, то в некоторых блоках может быть всего несколько генов, тогда как в других блоках может быть более сотни.

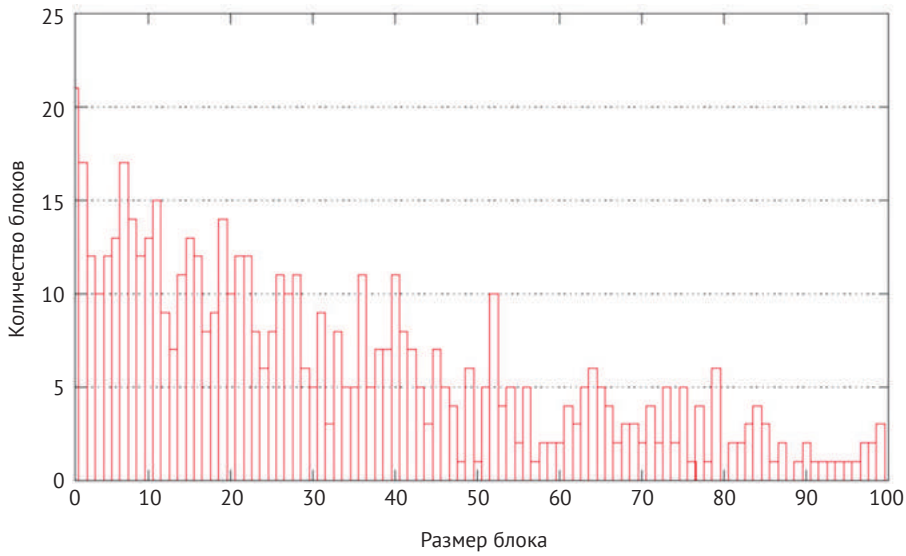


Рис. 6.3 Гистограмма, показывающая количество блоков каждого размера для генома с 25 000 генов (приблизительное количество генов в геноме млекопитающих) после 320 случайно выбранных реверсий. Блоки, содержащие более 100 генов, не показаны

ОСТАНОВИТЕСЬ и задумайтесь. Если N случайных реверсий генерируют $2N + 1$ синтенный блок, то почему семь реверсий, показанных на рис. 6.2, генерируют только 11 синтенных блоков?

Рисунок 6.4 усредняет результаты 100 таких симуляций и иллюстрирует, что распределение длин синтенных блоков может быть аппроксимировано кривой, соответствующей экспоненциальному распределению. Для получения дополнительной информации см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Экспоненциальное распределение**.

Экспоненциальное распределение предсказывает, что мы должны наблюдать около семи блоков, содержащих 34 гена, и один или два гораздо больших блока, содержащих 100 генов.

Что происходит, когда мы смотрим на гистограмму реальных синтенных блоков человека и мыши? Когда Надо и Тейлор построили эту гистограмму для ограниченных генетических данных, доступных в 1984 году, они заметили, что длины блоков хорошо соответствуют экспоненциальному распределению. В 1990-х годах более точные данные по синтенным блокам еще лучше соответствовали экспоненциальному распределению (рис. 6.5). Дело закрыто – хотя мы не знаем точных рекомбинаций, вызвавших эволюцию нашего генома за последние 75 млн лет, эти рекомбинации следовали «модели случайных разрывов»!

ОСТАНОВИТЕСЬ и задумайтесь. Согласны ли вы с логикой этого аргумента?

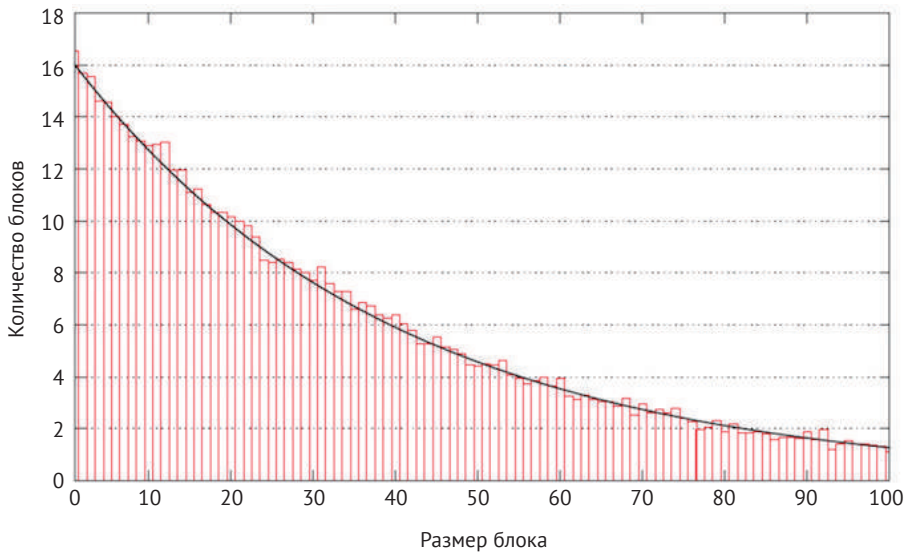


Рис. 6.4 Гистограмма длин синтенных блоков, усредненная по 100 симуляциям, подобранная экспоненциальным распределением. Каждое моделирование случайным образом выбирает 320 реверсий в гипотетическом геноме с 25 000 генов

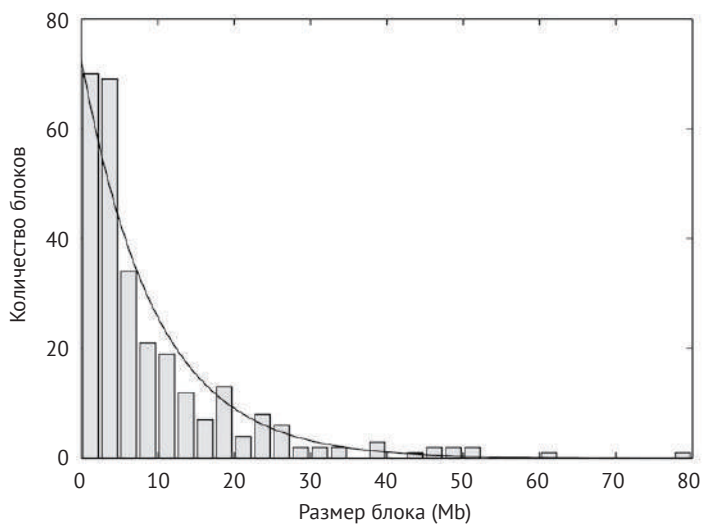


Рис. 6.5 Гистограмма длин синтенных блоков человека и мыши (показаны только синтенные блоки длиной более 1 млн нуклеотидов). Гистограмма соответствует экспоненциальному распределению

Сортировка по реверсиям

Теперь у нас есть доказательства в пользу модели случайных разрывов, но они далеко не окончательные. Чтобы проверить эту модель, давайте начнем строить математическую модель для анализа рекомбинаций. Поэтому вернемся к задаче, на которую мы намекали во введении, а именно поиску минимального числа реверсий, которые могли бы превратить X-хромосому мыши в X-хромосому человека.

ОСТАНОВИТЕСЬ и задумайтесь. С биологической точки зрения почему, по вашему мнению, нам нужно найти минимально возможное количество реверсий?

Мы требуем минимального количества реверсий в соответствии с принципом, называемым **бритвой Оккама**. Столкнувшись с затруднительным положением, мы должны объяснить его, используя простейшую гипотезу, которая согласуется с тем, что мы уже знаем. В этом случае кажется наиболее разумным, что эволюция пойдет по «кратчайшему пути» между двумя видами, т. е. по наиболее **экономному** эволюционному сценарию. Эволюция может не всегда идти по кратчайшему пути, но даже когда это не так, количество шагов в истинном эволюционном сценарии часто приближается к количеству шагов в самом экономном сценарии. Как же тогда найти длину этого кратчайшего пути?

Исследования рекомбинаций генома обычно игнорируют длины синтенных блоков и представляют хромосомы в виде **знаковых перестановок**. Каждый блок помечен числом, которому присваивается положительный или отрицательный знак в зависимости от направления блока; поскольку 0 обычно не имеет знака, перестановки используют индексацию на основе 1. Количество элементов в перестановке со знаком равно ее **длине**. Возвращаясь к 7-ступенчатому реверсному сценарию (рис. 6.2), X-хромосомы человека и мыши могут быть представлены следующими знаковыми перестановками длины 11:

Мышь: (+1 −7 +6 −10 +9 −8 +2 −11 −3 +5 +4).
Человек: (+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11).

В оставшейся части главы мы будем для краткости называть знаковые перестановки просто **перестановками**. Поскольку мы предполагаем, что каждый синтенный блок уникален, мы не допускаем повторения чисел в перестановках (например, (+1 −2 +3 +2) не является перестановкой).

Упражнение. Сколько существует перестановок длины 7?

Мы можем моделировать реверсии, реверсируя элементы в интервале перестановки, а затем меняя знаки любых элементов в реверсированном интерва-

ле. Например, рисунок внизу иллюстрирует, как реверсия меняет перестановку (+1 +2 +3 **+4 +5 +6 +7 +8** +9 +10) на (+1 +2 +3 **–8 –7 – 6 –5 –4** +9 +10). Эту реверсию можно рассматривать как первое нарушение перестановки между +3 и +4, а также между +8 и +9:

(+1 +2 +3 | **+4 +5 +6 +7 +8** | +9 +10).

Затем инвертируется средний сегмент:

(+1 +2 +3 | **–8 –7 –6 –5 –4** | +9 +10),

и, наконец, три сегмента склеиваются вместе, чтобы сформировать новую перестановку:

(+1 +2 +3 **–8 –7 –6 –5 –4** +9 +10)

+1 +2 +3 **+4 +5 +6 +7 +8** +9 +10
+1 +2 +3 **–8 –7 – 6 –5 –4** +9 +10.

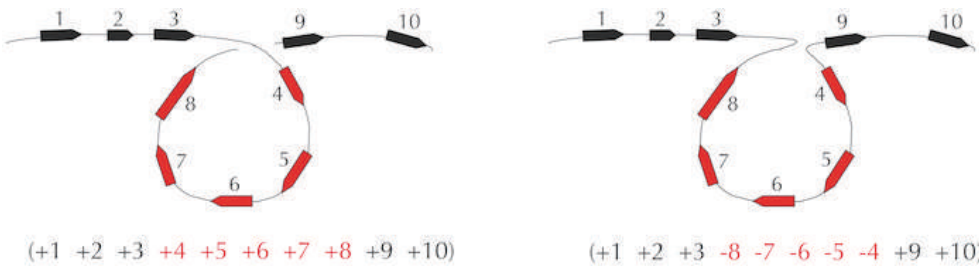


Рис. 6.6 Схема, показывающая, как реверсия разрывает хромосому в двух местах и инвертирует сегмент между двумя точками разрыва. Обратите внимание, что реверсия меняет знак каждого элемента в перевернутом сегменте перестановки

Упражнение. Сколько различных реверсий можно сделать в перестановке длиной 100?

Мы определяем **реверсное расстояние** между перестановками P и Q, обозначаемое $d_{rev}(P, Q)$, как минимальное количество реверсий, необходимых для преобразования P в Q.

Задача реверсного расстояния: *вычислить реверсное расстояние между двумя перестановками.*

Input: две перестановки равной длины.

Output: реверсное расстояние между этими перестановками.

Мы представили Х-хромосому человека как (+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11); такая перестановка, в которой блоки упорядочены от меньшего к большему с положительными направлениями, называется **перестановкой тождества**. Причина, по которой мы использовали перестановку идентичности длины 11 для представления Х-хромосомы человека, заключается в том, что при сравнении двух геномов мы можем маркировать синтенные блоки в одном из геномов как угодно. Маркировка блока, для которой человеческая Х-хромосома является перестановкой идентичности, автоматически индуцирует представление мышиной хромосомы как (+1 –7 +6 –10 +9 –8 +2 –11 –3 +5 +4).

Конечно, вместо этого мы могли бы закодировать Х-хромосому мыши как перестановку идентичности, что вызвало бы кодирование Х-хромосомы человека как (+1 +7 –9 +11 +10 +3 –2 –6 +5 –4 –8) (рисунок ниже).

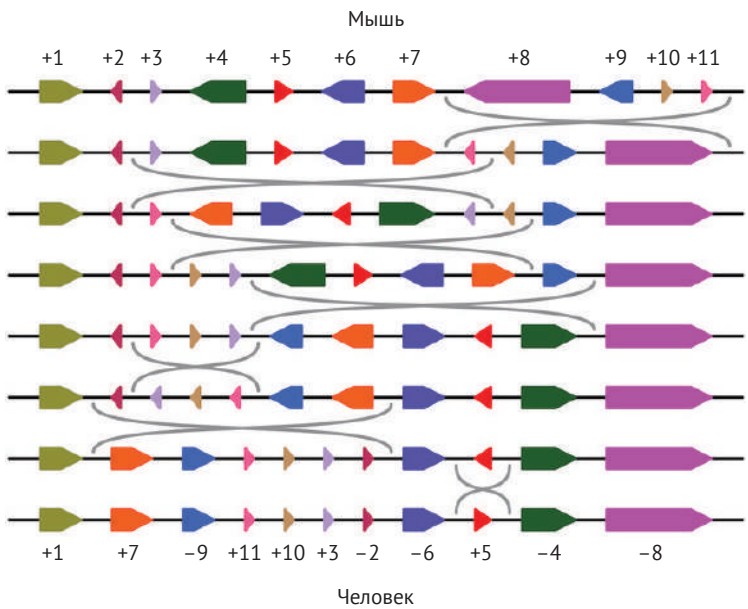


Рис. 6.7 Кодирование Х-хромосомы мыши как перестановку идентичности подразумевает кодирование Х-хромосомы человека как (+1 +7 –9 +11 +10 +3 –2 –6 +5 –4 –8)

Поскольку у нас есть свобода обозначать синтенные блоки как угодно, мы рассмотрим эквивалентную версию задачи о реверсном расстоянии, в которой перестановка Q является тождественной перестановкой (+1 +2 ... +n). Эта вычислительная задача называется **сортировкой по реверсиям**, и мы обозначаем минимальное количество реверсий, необходимых для сортировки P в тождественной перестановке, как $d_{rev}(P)$. История сортировки по реверсиям основана на кулинарном приложении и включает в себя двух знаменитостей

(подробнее см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Билл Гейтс и Дэвид Х. Коэн переворачивают блины**).

Задача сортировки по реверсиям: *вычислить реверсное расстояние между перестановкой и перестановкой тождества.*

Input: перестановка P.

Output: реверсное расстояние $d_{rev}(P)$.

Представляем сортировку (+2 −4 −3 +5 −8 −7 −6 +1) с использованием пяти реверсий:

(+2 −4 −3 +5 −8 −7 −6 +1)
(+2 +3 +4 +5 −8 −7 −6 +1)
(+2 +3 +4 +5 +6 +7 +8 +1)
(+2 +3 +4 +5 +6 +7 +8 −1)
(−8 −7 −6 −5 −4 −3 −2 −1)
(+1 +2 +3 +4 +5 +6 +7 +8).

ОСТАНОВИТЕСЬ и задумайтесь. Сможете ли вы отсортировать эту перестановку, используя меньше реверсий?

Вот еще более быстрая сортировка:

(+2 −4 −3 +5 −8 −7 −6 +1)
(+2 +3 +4 +5 −8 −7 −6 +1)
(−5 −4 −3 −2 −8 −7 −6 +1)
(−5 −4 −3 −2 −1 +6 +7 +8)
(+1 +2 +3 +4 +5 +6 +7 +8).

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующие вопросы.

1. Можно ли отсортировать эту перестановку еще быстрее?
2. При сортировке по реверсиям промежуточные перестановки в приведенном выше примере становятся все более и более «упорядоченными». Можете ли вы придумать количественную меру того, насколько упорядочена перестановка?

Жадный алгоритм сортировки по реверсиям

Давайте посмотрим, сможем ли мы разработать жадную эвристику для аппроксимации $d_{rev}(P)$. Простейшая идея состоит в том, чтобы выполнить реверсии, фиксирующие +1 в первой позиции, за которыми следуют реверсии, фиксирующие +2 во второй позиции, и т. д. Например, элемент 1 уже находится в правильном положении и имеет правильный знак (+) в X-хромосоме мыши, а элемент 2 находится в неправильном положении. Мы можем зафиксировать элемент 1 и переместить элемент 2 в правильное положение, применив одну реверсию.

(+1 −7 +6 −10 +9 −8 +2 −11 −3 +5 +4)
(+1 −2 +8 −9 +10 −6 +7 −11 −3 +5 +4).

Еще одна реверсия переворачивает элемент 2 так, чтобы он имел правильный знак:

(+1 −2 +8 −9 +10 −6 +7 −11 −3 +5 +4)
(+1 +2 +8 −9 +10 −6 +7 −11 −3 +5 +4).

С помощью итераций мы можем последовательно перемещать все более и более крупные элементы на их правильные позиции в перестановке идентичности, следуя приведенным ниже реверсиям. Инвертированный интервал каждой реверсии по-прежнему отображается красным цветом, а элементы, которые были размещены в правильном положении, – синим.

(+1 −7 +6 −10 +9 −8 +2 −11 −3 +5 +4)
(+1 −2 +8 −9 +10 −6 +7 −11 −3 +5 +4)
(+1 +2 +8 −9 +10 −6 +7 −11 −3 +5 +4)
(+1 +2 +3 +11 −7 +6 −10 +9 −8 +5 +4)
(+1 +2 +3 −4 −5 +8 −9 +10 −6 +7 −11)
(+1 +2 +3 +4 −5 +8 −9 +10 −6 +7 −11)
(+1 +2 +3 +4 +5 +8 −9 +10 −6 +7 −11)
(+1 +2 +3 +4 +5 +6 −10 +9 −8 +7 −11)
(+1 +2 +3 +4 +5 +6 −7 +8 −9 +10 −11)
(+1 +2 +3 +4 +5 +6 +7 +8 −9 +10 −11)
(+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 −11)
(+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11).

Предыдущий пример мотивирует жадную эвристику под названием **GreedySorting**. Мы говорим, что элемент k в перестановке $P = (p_1 \ldots p_n)$ **отсортирован**, если $p_k = +k$, и **не отсортирован** в противном случае. Мы называем P ***k*-отсортированным**, если его первые $k - 1$ элементов отсортированы, но если элемент k не отсортирован. Для каждой k-отсортированной перестановки P су-

ществует единственная реверсия, называемая ***k*-сортировочной реверсией**, которая фиксирует первые $k - 1$ элементов P и перемещает элемент k на позицию k. В случае, когда $-k$ уже находится в k-й позиции P, реверсия k-сортировки просто переворачивает $-k$.

Например, при сортировке X-хромосомы мыши, показанной выше, реверсия 2-сортировки преобразует (**+1 −7 +6 −10 +9 −8 +2** −11 −3 +5 +4) в (**+1** −2 +8 −9 +10 −6 +7 −11 −3 +5 +4). В этом случае для сортировки элемента 2 потребовалось одна дополнительная реверсия 2-сортировки, переворачивающая −2. Таким образом, идея **GreedySorting** заключается в применении реверсивной k-сортировки для увеличения значения k. Здесь $|P|$ относится к длине перестановки P.

```
GreedySorting(P)
    approxReversalDistance ← 0
    for k = 1 до |P|
        if элемент k не отсортирован
            применить  k-сортировачную реверсию к P
            approxReversalDistance ← approxReversalDistance + 1
        if k-й элемент P является −k
            применить  k-сортировачную реверсию к P
            approxReversalDistance ← approxReversalDistance + 1
    return approxReversalDistance
```

Упражнение. Сколько реверсий нужно **GreedySorting**, чтобы отсортировать перестановку (+100 +99 … +2 +1)?

В случае X-хромосомы мыши **GreedySorting** требует 11 реверсий, но мы уже знаем, что эту перестановку можно отсортировать с помощью семи реверсий, что заставляет задуматься: насколько хороша эвристика жадной сортировки **GreedySorting**?

Упражнение. Какое максимальное количество реверсий может потребоваться **GreedySorting** для сортировки перестановки длины 100?

Рассмотрим перестановку (−6 +1 +2 +3 +4 +5). Вы можете убедиться, что **GreedySorting** требует десяти шагов для сортировки этой перестановки, и все же ее можно отсортировать, используя всего два обращения!

(**−6 +1 +2 +3 +4 +5**)
(**−5 −4 −3 −2 −1** +6)
(+1 +2 +3 +4 +5 +6)

Этот пример показывает, что эта жадная эвристика дает плохое приближение для реверсного расстояния.

ОСТАНОВИТЕСЬ и задумайтесь. Сможете ли вы найти нижнюю границу для $d_{rev}(P)$? Например, можете ли вы показать, что перестановку мыши (+1 −7 +6 −10 +9 −8 +2 −11 −3 +5 +4) нельзя отсортировать менее чем за семь рекомбинаций?

Точки останова

Что такое точки останова?

Рассмотрим сортировку по реверсиям, показанную ниже. Мы хотели бы количественно определить, как каждая последующая перестановка приближается к идентичности по мере того, как мы применяем последующие реверсии. Для первой реверсии в правой конечной точке инвертированного интервала он меняет последовательные элементы (−11 + 13) на гораздо более желательные (+12 + 13). Менее очевидна работа четвертой реверсии, которая размещает −11 непосредственно слева от −10, так что на следующем шаге последовательные элементы (−11 −10) могут быть частью инвертированного интервала, создавая желаемые последовательные элементы (+10 +11).

	BREAKPOINTS(P)
(+3 +4 +5 −12 −8 −7 −6 +1 +2 +10 +9 −11 +13 +14)	8
(+3 +4 +5 +11 −9 −10 −2 −1 +6 +7 +8 +12 +13 +14)	7
(+1 +2 +10 +9 −11 −5 −4 −3 +6 +7 +8 +12 +13 +14)	6
(+1 +2 +3 +4 +5 +11 −9 −10 +6 +7 +8 +12 +13 +14)	5
(+1 +2 +3 +4 +5 +9 −11 −10 +6 +7 +8 +12 +13 +14)	4
(+1 +2 +3 +4 +5 +9 −8 −7 −6 +10 +11 +12 +13 +14)	3
(+1 +2 +3 +4 +5 +6 +7 +8 −9 +10 +11 +12 +13 +14)	2
(+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11 +12 +13 +14)	0

Рис. 6.8 Сортировка по реверсиям

Интуитивное предположение, которое мы пытаемся реализовать, заключается в том, что последовательные элементы, такие как (+12 +13), желательны, потому что они появляются в том же порядке, что и в перестановке идентичности. Однако также желательны последовательные элементы, такие как (−11 −10), поскольку эти элементы могут быть позже инвертированы в правильном порядке (+10 +11). У пар (+12 +13) и (−11 −10) есть что-то общее; второй элемент равен первому элементу плюс 1. Поэтому мы говорим, что последовательные элементы $(p_i\ p_{i+1})$ в перестановке $P = (p_1 \ldots p_n)$ образуют

смежность (или **примыкание**), если $p_{i+1} - p_i$ равно 1. По определению для любого положительного элемента $k < n$ как $(k\ k+1)$, так и $(-(k+1)\ -k)$ являются смежностями. Если $p_{i+1} - p_i$ не равно 1, то мы говорим, что $(p_i\ p_{i+1})$ является **точкой останова**.

Интуитивно мы можем думать о точке останова как о паре последовательных элементов, которые «выскакивают из общего порядка» по сравнению с перестановкой тождества $(+1\ +2\ \ldots\ +n)$. Например, пара $(+5\ -12)$ является точкой останова, потому что +5 и −12 не являются соседями в перестановке идентичности. Точно так же $(-12\ -8)$, $(-6\ +1)$, $(+2\ +10)$, $(+9\ -11)$ и $(-11\ +13)$ явно не в общем порядке. Но $(+10\ +9)$ также является точкой останова (даже несмотря на то, что она состоит из последовательных целых чисел), потому что ее знаки не совпадают по порядку по сравнению с перестановкой тождества.

ОСТАНОВИТЕСЬ и задумайтесь. Перестановка $(-5\ -4\ -3\ -2\ -1)$ явно не является перестановкой тождества, но где ее точки останова?

Далее мы будем представлять начало и конец перестановки P, добавляя **0** слева от первого элемента и $\boldsymbol{n+1}$ справа от последнего элемента:

$$(\mathbf{0}\ p_1 \ldots p_n\ (\boldsymbol{n+1})).$$

В результате имеется $n + 1$ пара последовательных элементов:

$$(\mathbf{0}\ p_1),\ (p_1\ p_2),\ (p_2\ p_3),\ \ldots,\ (p_{n-1}\ p_n),\ (p_n\ (\boldsymbol{n+1})).$$

Мы используем *Adjacencies*(P) и *Breakpoints*(P) для обозначения числа смежностей и точек останова перестановки P соответственно. На рис. 6.8 показано, как изменяется количество точек останова во время сортировки по разворотам с предыдущего шага. Обратите внимание, что 0 и n + 1 являются заполнителями места и на них не может повлиять разворот. Также обратите внимание, что перестановка в начале этого рисунка имеет восемь точек останова и шесть смежностей.

Счет точек останова

Поскольку любая пара последовательных элементов перестановки образует либо точку останова, либо смежность, мы имеем следующее тождество для любой перестановки P длины n:

$$Adjacencies(P) + Breakpoints(P) = n + 1.$$

Из этой формулы следует, что перестановка n элементов может иметь не более $n + 1$ смежностей.

ОСТАНОВИТЕСЬ и задумайтесь. Сколько рекомбинаций на n элементах имеют $n + 1$ смежностей?

Вы можете убедиться, что перестановка идентичности $(+1\ +2\ \ldots\ +n)$ является единственной перестановкой, для которой все последовательные элементы являются смежными, что означает, что она не имеет точек останова. Заметим также, что перестановка $(-n\ -(n-1)\ \ldots\ -2\ -1)$ имеет смежности для каждой последовательной пары элементов, за исключением двух точек останова $(\mathbf{0}, -n)$ и $(-1\ (\boldsymbol{n+1}))$ на концах перестановки.

Упражнение. Сколько рекомбинаций длины 200 имеют ровно 199 смежностей?

Задача определения количества точек останова: *найти количество точек останова в перестановке.*

Input: перестановка.

Input: количество точек останова в этой перестановке.

☞ Загрузить данные 6.1

ОСТАНОВИТЕСЬ и задумайтесь. Мы определили точку останова между произвольной перестановкой и перестановкой идентичности. Обобщите понятие точки останова между двумя произвольными перестановками и разработайте алгоритм с линейным временем для вычисления этого числа.

Сортировка по реверсиям для устранения точек останова

Реверсии в нашем примере сортировки по реверсиям (показано ниже) уменьшают количество точек останова с 8 до 0. Обратите внимание, что перестановка становится все более и более «упорядоченной» после каждой реверсии по мере уменьшения количества точек останова на каждом шаге. Таким образом, вы можете думать о сортировке по реверсиям как о процессе исключения точек останова – уменьшении количества точек останова в перестановке P из $Breakpoints(P)$ до 0 (рис. 6.8).

ОСТАНОВИТЕСЬ и задумайтесь. Какое максимальное количество точек останова можно устранить за одну реверсию?

Например, есть пять точек останова в диапазоне следующей реверсии перестановки (0 +3 +4 +5 **−12 −8 −7 −6 +1 +2 +10 +9 −11** +13 +14 15):

(−12 −8) (−6 +1) (+2 +10) (+10 +9) (+9 −11).

После реверсирования эти точки останова становятся следующими пятью точками останова:

(+11 −9) (−9 −10) (−10 −2) (−1 +6) (+8 +12).

Поскольку все точки останова внутри и вне диапазона реверсии остаются точками останова после реверсии, единственными точками останова, которые могут быть устранены путем реверсии, являются две точки останова, расположенные на границах реверсированного интервала. Возвращаясь к реверсиям

(+3 +4 +5 −12 −8 −7 −6 +1 +2 +10 +9 −11 +13 +14)
(+3 +4 +5 +11 −9 −10 −2 −1 +6 +7 +8 +12 +13 +14),

точки останова на границах первой реверсии будут (+5 −12) и (−11 +13); реверсия преобразует их в точку останова (+5 +11) и смежность (+12 +13), тем самым уменьшая количество точек останова на 1.

ОСТАНОВИТЕСЬ и задумайтесь. Может ли перестановка (+3 +4 +5 −12 −8 −7 −6 +1 +2 +10 +9 −11 +13 +14), имеющая восемь точек останова, быть отсортирована с тремя реверсиями?

Реверсия может устранить не более двух точек останова, поэтому две реверсии могут устранить не более четырех точек останова, три реверсии могут устранить не более шести точек останова и т. д. Это рассуждение устанавливает следующую теорему.

Теорема о точке останова. Реверсное расстояние $d_{rev}(P)$ больше или равно $Breakpoints(P)/2$.

Было бы хорошо, если бы мы *всегда* могли найти реверсию, которая устраняет две точки останова из перестановки, так как это подразумевало бы простой жадный алгоритм для оптимальной сортировки по реверсиям. К сожалению, это не тот случай. Вы можете убедиться, что нет реверсии, уменьшающей количество точек останова в перестановке P = (+2 +1), которая имеет три точки останова.

Упражнение. Сколько рекомбинаций длины 10 обладают свойством, согласно которому никакая реверсия, примененная к P, не уменьшает $Breakpoints(P)$?

Оказывается, любую перестановку длины n можно отсортировать, используя не более $n + 1$ реверсий, и что перестановка $(+n + (n - 1) \ldots +1)$ требует для сортировки $n + 1$ реверсий. Поскольку эта перестановка имеет $n + 1$ точку останова, существует большой разрыв между нижней границей $(n + 1)/2$, обеспечиваемой теоремой о точке останова, и реверсным расстоянием.

Упражнение. Докажите, что существует кратчайшая последовательность реверсий, сортирующая перестановку, которая никогда не нарушает перестановку в смежности.

Вы скоро увидите, что идея точек останова поможет нам вернуться к нашей первоначальной цели тестирования модели случайных разрывов. А пока мы хотели бы перейти от перестановок, которые могут моделировать только отдельные хромосомы, к более общей мультихромосомной модели. Вы можете быть удивлены тем, что мы переходим, казалось бы, к более сложной модели, прежде чем решать однохромосомный случай, который и без того сложен. Однако оказывается, что нашу новую мультихромосомную модель будет легче анализировать!

Рекомбинации в геномах опухолей

По мере того как мы продвигаемся к более надежной модели сравнения геномов, нам необходимо включать рекомбинации, которые перемещают гены из одной хромосомы в другую. Действительно, за заметным исключением Х-хромосомы, гены из одной хромосомы человека обычно имеют свои аналоги, распределенные по многим хромосомам мыши (и наоборот). Мы надеемся, что в вашей голове звучит ворчливый голос, задающий вопрос: *как рекомбинация генома может повлиять на несколько хромосом?*

Хотя мультихромосомные рекомбинации происходили в ходе эволюции видов на протяжении миллионов лет, оказывается, что мы можем наблюдать их в гораздо более узких временных рамках в раковых клетках, которые демонстрируют множество хромосомных аберраций. Некоторые из этих мутаций не оказывают прямого влияния на развитие опухоли, но многие типы опухолей демонстрируют повторяющиеся рекомбинации, которые запускают рост опухоли путем разрыва генов или изменения их регуляции. Изучая эти рекомбинации, мы можем идентифицировать гены, важные для роста опухоли, что ведет к улучшению диагностики и терапии рака.

На рисунке ниже представлена рекомбинация хромосом 9 и 22 человека при редкой форме рака, называемой **хроническим миелоидным лейкозом (CML)**. В этом типе рекомбинации, называемой **транслокацией**, два участка ДНК вырезаются из концов хромосом 9 и 22, а затем прикрепляются к противоположным хромосомам. Одна из перестроенных хромосом называется **филадельфийской хромосомой**. Эта хромосома объединяет два гена, называемых

ABL и BCR, которые обычно не имеют ничего общего друг с другом. Однако при их соединении в филадельфийской хромосоме эти два гена создают единый **химерный ген**, кодирующий **гибридный белок ABL-BCR**, который участвует в развитии **ХМЛ**.

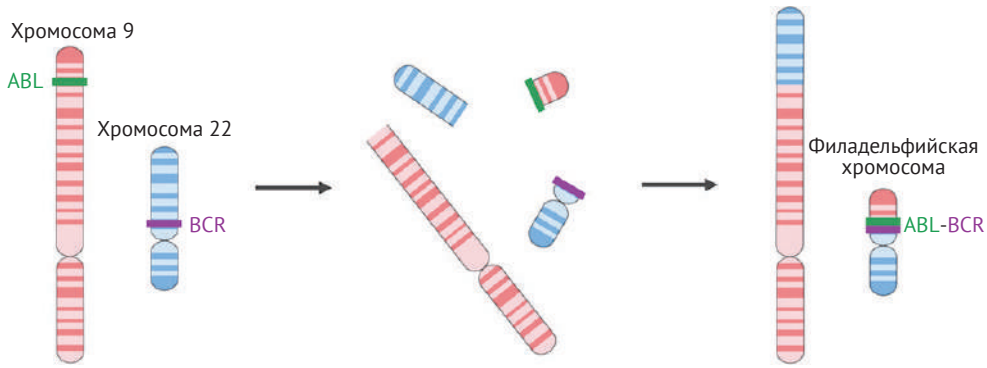


Рис. 6.9 Филадельфийская хромосома образована транслокацией, захватывающей хромосомы 9 и 22. Она объединяет ген ABL и часть гена BCR, образуя химерный ген, который может вызывать CML

Как только ученые поняли первопричину CML, они начали поиск соединения, ингибирующего ABL-BCR, что привело к появлению в 2001 году препарата под названием **гливек**. Гливек – это **таргетная терапия** против CML, которая ингибирует раковые клетки, но не влияет на нормальные клетки и показала отличные клинические результаты. Однако, поскольку препарат нацелен только на слитый белок ABL-BCR, он работает при CML, но не лечит большинство других видов рака. Тем не менее появление гливека укрепило надежды исследователей на то, что поиск специфических рекомбинаций при других формах рака может привести к созданию дополнительных таргетных методов лечения рака.

От монохромосомных к мультихромосомным геномам

Транслокации, слияния и расщепления

Для моделирования транслокаций мы представляем мультихромосомный геном с k хромосомами как перестановку, разделенную на k частей. Например, геном (+1 +2 +3 +4 +5 +6)(+7 +8 +9 +10 +11) состоит из двух хромосом (+1 +2 +3 +4 +5 + 6) и (+7 +8 +9 +10 +11). Транслокация заменяет сегменты разных хромосом, например транслокация (**+1 +2 +3 +4 +5 +6**) и (**+7 +8 +9 +10 +11**) может привести к хромосомам (**+1 +2 +3 +4 +9 +10 +11**) и (**+7 +8 +5 +6**). Вы можете думать о транслокации как о разрыве каждой из двух хромосом

(+1 +2 +3 +4 +5 +6) (+7 +8 +9 +10 +11)

на две части:

(+1 +2 +3 +4) (+5 +6) (+7 +8) (+9 +10 +11),

а затем склеивание полученных сегментов в две новые хромосомы:

(+1 +2 +3 +4 +9 +10 +11) (+7 +8 +5 +6).

Рекомбинации в мультихромосомных геномах не ограничиваются реверсиями и транслокациями. К ним также относятся **слияния** хромосом, при которых две хромосомы сливаются в одну, а также **расщепления**, при которых одна хромосома распадается на две. Например, (+1 +2 +3 +4 +5 +6) и (+7 +8 +9 +10 +11) могут сливаться в одну хромосому (+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11); последующее деление этой хромосомы может привести к двум хромосомам (+1 +2 +3 +4) и (+5 +6 +7 +8 +9 +10 +11). Пять миллионов лет назад, вскоре после разделения линий человека и шимпанзе, слияние двух хромосом (называемых 2A и 2B) у одного из наших предков создало человеческую хромосому 2 и уменьшило количество хромосом с 24 до 23.

ОСТАНОВИТЕСЬ и задумайтесь. *Априори* вполне могло быть так, что предок человека – шимпанзе – имел неповрежденную хромосому 2 и что в результате деления эти две хромосомы разделились на хромосомы 2A и 2B шимпанзе. Как бы вы выбрали один из двух сценариев? Подсказка: у горилл и орангутангов, как и у шимпанзе, тоже 24 хромосомы.

От генома к графу

В дальнейшем будем предполагать, что все хромосомы в геноме кольцевые. Это предположение представляет собой небольшое искажение биологической реальности, поскольку хромосомы млекопитающих линейны. Однако превращение линейной хромосомы в кольцо путем соединения ее концов упростит последующий анализ, не повлияв на наш окончательный вывод.

Теперь у нас есть мультихромосомная геномная модель, а также четыре типа рекомбинаций (реверсии, транслокации, слияния и деления), которые могут трансформировать один геном в другой. Для моделирования геномов с кольцевыми хромосомами мы будем использовать **граф генома**. Представим каждый синтенный блок направленным черным ребром, указывающим его направление, а затем соединим черные ребра, соответствующие соседним синтенным блокам, цветным ненаправленным ребром. На рисунке ниже каждая кольцевая хромосома показана как **альтернативный цикл** красных и черных ребер. В этой модели геном человека может быть представлен с по-

мощью 280 синтенных блоков человек–мышь, распределенных по 23 альтернативным циклам.

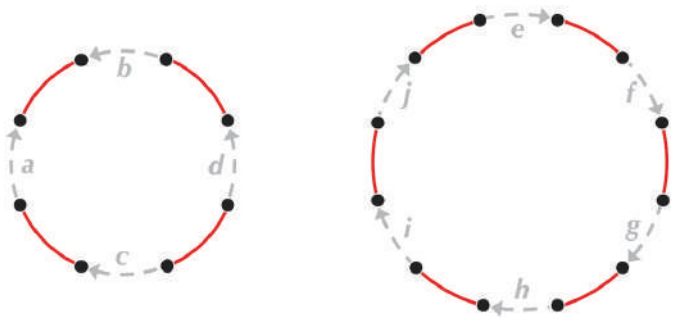


Рис. 6.10 Геном с двумя кольцевыми хромосомами (+*a* −*b* −*d* +*c*) и (+*e* +*f* +*g* +*h* +*i* +*j*). Черные направленные ребра представляют синтенные блоки, а красные ненаправленные ребра соединяют соседние синтенные блоки. Кольцевую хромосому с *n* элементами можно записать 2*n* различными способами; хромосома слева может быть записана как (+*a* −*b* −*d* +*c*), (−*b* −*d* +*c* +*a*), (−*d* +*c* +*a* −*b*), (+*c* +*a* −*b* −*c*), (−*a* −*c* +*d* +*b*) (−*c* +*d* +*b* −*a*), (+*d* +*b* −*a* −*c*) и (+*b* −*a* −*c* +*d*)

ОСТАНОВИТЕСЬ и задумайтесь. Пусть *P* и *Q* будут геномами, состоящими из линейных хромосом, и пусть *P*' и *Q*' будут кольцевыми версиями этих геномов. Можете ли вы преобразовать данную серию реверсий/транслокаций/слияний/расщеплений, превращающих *P* в *Q*, в серию рекомбинаций, превращающих *P*' в *Q*'? А как насчет обратной операции – можете ли вы преобразовать серию рекомбинаций, преобразующих *P*' в *Q*', в серию рекомбинаций, превращающих *P* в *Q*?

Двойные разрывы

Теперь мы сосредоточимся на одной из хромосом в мультихромосомном геноме и рассмотрим реверсию, преобразующую кольцевую хромосому *P* = (+*a* −*b* −*c* +*d*) в *Q* = (+*a* −*b* −*d* +*c*). Мы можем нарисовать *Q* разными способами, в зависимости от того, как мы расположим черные ребра. На рисунке ниже показаны два таких эквивалентных представления.

Хотя *Q* слева (рис. 6.11) является наиболее естественным представлением, мы будем использовать второе представление, потому что его черные ребра расположены по кругу точно в том же порядке, что и в естественном представлении *P* = (+*a* −*b* −*c* +*d*).

Как показано на рис. 6.12, фиксирование черных краев позволяет нам визуализировать эффект реверсии. Как видите, реверсия удаляет («разрывает»)

два красных ребра в P (соединяющих b с c и d с a) и заменяет их двумя новыми красными ребрами (соединяющими b с d и c с a).

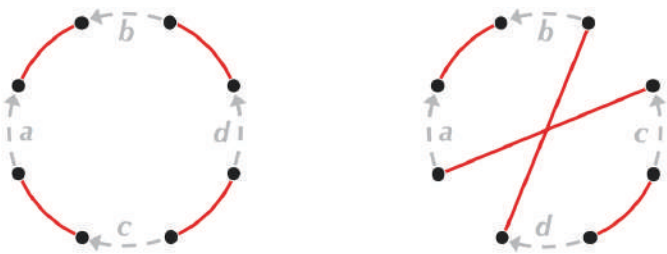


Рис. 6.11 Две эквивалентные схемы кольцевой хромосомы $Q = (+a\ -b\ -d\ +c)$

Рис. 6.12 Реверсия преобразует $P = (+a\ -b\ -c\ +d)$ в $Q = (+a\ -b\ -d\ +c)$. Мы расположили черные ребра Q так, чтобы они имели ту же ориентацию и положение, что и черные ребра в естественном представлении P. Реверсию можно рассматривать как удаление двух красных ребер, помеченных звездочками, и замену их двумя новыми красными ребрами на одних и тех же четырех узлах

На рисунке ниже показано расщепление генома $P = (+a\ -b\ -c\ +d)$ на $Q = (+a\ -b)$ $(-c\ +d)$; реверсия этой операции соответствует слиянию двух хромосом Q с получением P. И операции слияния, и операции расщепления, как и реверсия, соответствуют удалению двух ребер в одном геноме и замене их двумя новыми ребрами в другом геноме.

Рис. 6.13 Расщепление одиночной хромосомы $P = (+a\ -b\ -c\ +d)$ на геном $Q = (+a\ -b)(-c\ +d)$. Обратная операция представляет собой слияние, превращающее две хромосомы Q в одну хромосому путем удаления двух красных ребер Q и замены их двумя другими ребрами

Транслокацию, включающую две *линейные* хромосомы, также можно имитировать путем циркуляризации этих хромосом и последующей замены двух красных ребер двумя другими красными ребрами, как показано на рисунке ниже. Таким образом, мы нашли общую тему, объединяющую четыре различных типа перегруппировок. Все их можно рассматривать как разрыв двух красных ребер графа генома и замену их двумя новыми красными ребрами на тех же четырех узлах. По этой причине мы определяем общую операцию на графе генома, в которой два красных ребра заменяются двумя новыми красными ребрами на тех же четырех узлах, как **двойной разрыв** («двойной разрез», или «2-Break»).

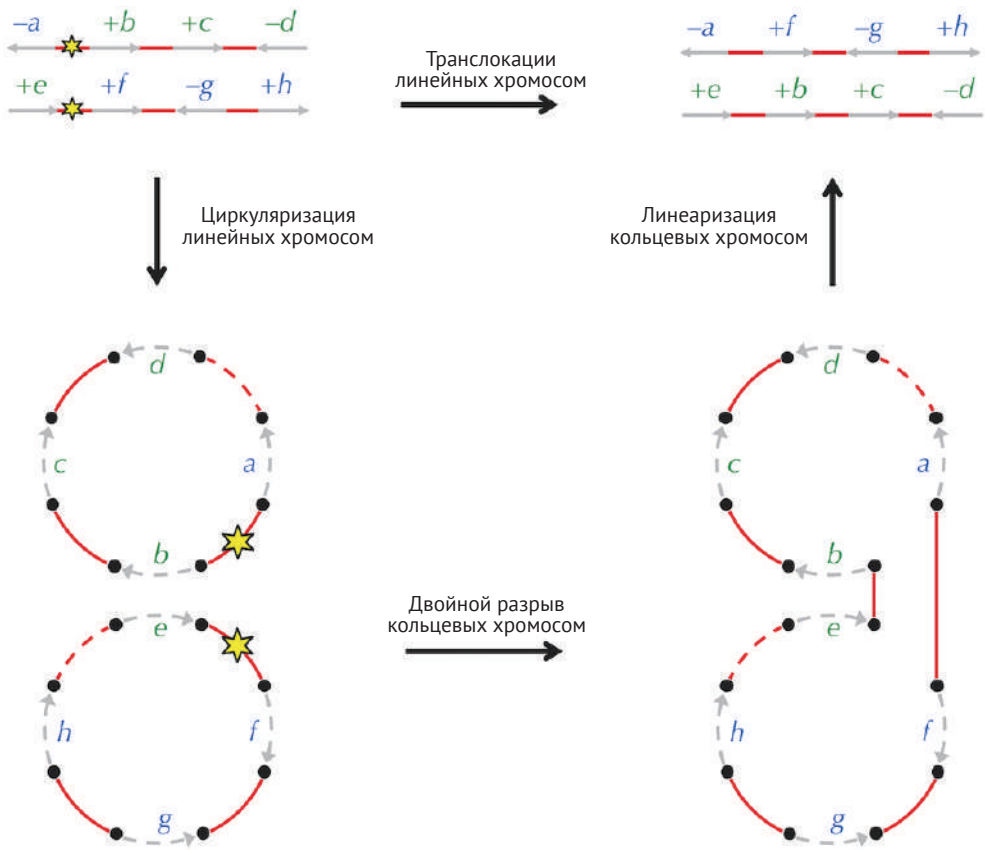


Рис. 6.14 Транслокация линейных хромосом (*–a +b +c –d*) и (*+e +f –g +h*) превращает их в линейные хромосомы (*–a +f –g +h*) и (*+e +b +c –d*). Эту транслокацию также можно осуществить, сначала циркуляризировав эти хромосомы (т. е. соединяя концы каждой хромосомы красным пунктирным ребром), затем применяя двойной разрыв к новым хромосомам и, наконец, превращая полученную кольцевую хромосому в две линейные хромосомы путем удаления пунктирных красных ребер

Мы хотели бы найти кратчайшую последовательность двойных разрывов, преобразующих геном P в геном Q, и мы называем количество операций в кратчайшей последовательности двойных разрывов, преобразующих P в Q, **дистанцией двойного разрыва** между P и Q, обозначается $d(P, Q)$.

Задача дистанции двойного разрыва: *найти дистанцию двойного разрыва между двумя геномами.*

Input: два генома с кольцевыми хромосомами в одном наборе синтенных блоков.

Output: дистанцию двойного разрыва между этими геномами.

Графы точек останова

Чтобы вычислить дистанцию двойного разрыва, мы вернемся к понятию точек разрыва, чтобы построить граф для сравнения двух геномов. Рассмотрим геномы $P = (+a\ -b\ -c\ +d)$ и $Q = (+a\ +c\ +b\ -d)$. Обратите внимание, что мы использовали красный цвет для цветных ребер P и синий цвет для цветных ребер Q. Как и прежде, мы переставляем черные ребра Q так, чтобы они располагались точно так же, как в P (рис. 6.15, в середине). Если мы наложим графы геномов P и Q, то получим трехцветный граф точек останова *BreakpointGraph*(P, Q), показанный на рисунке ниже.

Обратите внимание, что красные и черные ребра в графе точек останова образуют P, а синие и черные ребра образуют Q. Более того, красные и синие ребра в графе точек останова образуют набор альтернативных красно-синих циклов.

ОСТАНОВИТЕСЬ и задумайтесь. Докажите, что красные и синие ребра в любом графе точек останова образуют альтернативные циклы. Подсказка: сколько красных и синих ребер встречается в каждом узле графа точек останова?

Обозначим количество альтернативных красно-синих циклов в *BreakpointGraph*(P, Q) как *Cycles*(P, Q). Для $P = (+a\ -b\ -c\ +d)$ и $Q = (+a\ +c\ +b\ -d)$, *Cycles*(P, Q) = 2, как показано на рисунке. В дальнейшем мы сосредоточимся на альтернативных красно-синих циклах в графах точек останова и часто опускаем черные ребра.

Мы проиллюстрировали построение графа точек останова для однохромосомных геномов, но точно так же можно построить граф точек останова и для многохромосомных геномов (рис. 6.16).

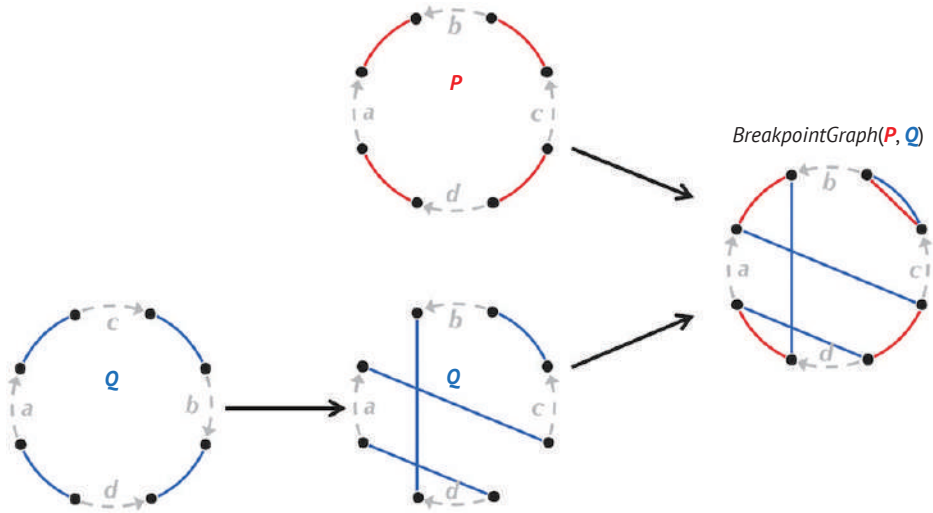


Рис. 6.15 Построение графа точек останова для однохромосомных геномов ***P*** = (+a −b −c + d) и ***Q*** = (+a +c +b −d). После перестановки черных ребер ***Q*** так, чтобы они располагались так же, как в ***P***, граф точек останова *BreakpointGraph*(***P***, ***Q***) формируется путем наложения графов ***P*** и Q. Как показано справа, есть два альтернативных красно-синих цикла на этом графе точек останова

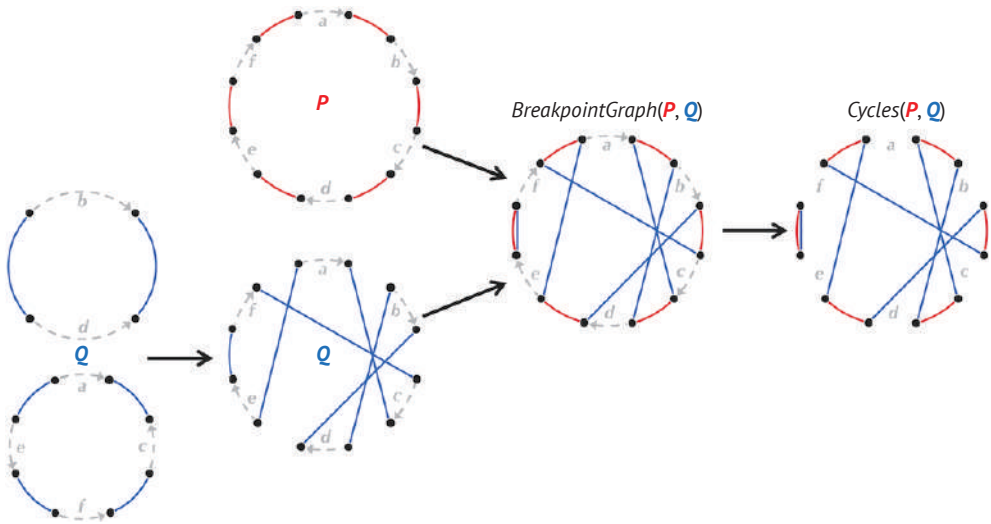


Рис. 6.16 Построение *BreakpointGraph*(***P***, ***Q***) для однохромосомного генома ***P*** = (*+a +b +c +d +e +f*) и двухромосомного генома ***Q*** = (*+a −c −f −e*) (*+b −d*). Внизу, чтобы проиллюстрировать построение графа точек останова, мы сначала переставляем черные ребра ***Q*** так, чтобы они рисовались в том же порядке вдоль цикла, что и в ***P***

ОСТАНОВИТЕСЬ и задумайтесь. При заданном геноме ***P*** какой геном ***Q*** максимизирует *Cycles*(***P***, ***Q***)?

В случае, когда ***P*** и ***Q*** имеют одинаковое количество синтенных блоков, мы обозначаем количество их синтенных блоков как *Blocks*(***P***, ***Q***). Как показано на рисунке ниже, когда ***P*** и ***Q*** идентичны, их граф точек останова состоит из циклов *Blocks*(***P***, ***Q***) длины 2, каждый из которых содержит одно красное и одно синее ребро. Мы называем циклы длины 2 тривиальными циклами, а граф точек останова, образованный идентичными геномами, – тривиальным графом точек останова.

Упражнение. Докажите, что количество циклов в *BreakpointGraph*(***P***, ***Q***) меньше, чем в *Blocks*(***P***, ***Q***), если только ***P*** не равно ***Q***.

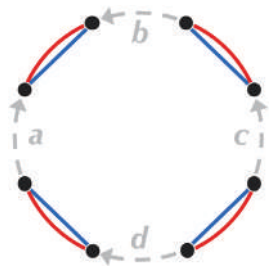


Рис. 6.17 Тривиальный граф точек останова *BreakpointGraph*(***P***, ***P***), образованный двумя копиями генома ***P*** = (+*a* −*b* −*c* +*d*). Граф точек останова любого генома сам по себе состоит только из альтернативных циклов длины 2

Вероятно, вы задаетесь вопросом, чем полезен граф точек останова. Мы можем рассматривать двойной разрыв, преобразующий ***P*** в ***P'***, как операцию над *BreakpointGraph*(***P***, ***Q***), которая дает *BreakpointGraph*(***P'***, ***Q***) (рисунок ниже).

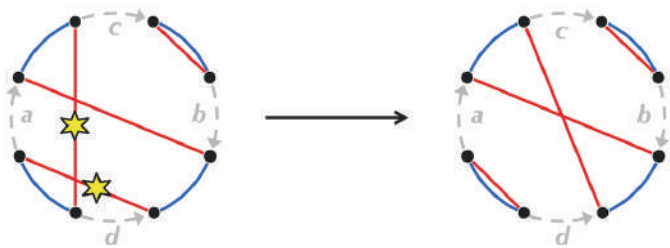


Рис. 6.18 Двойной разрыв (обозначен звездочками), преобразующий геном ***P*** в ***P'***, также преобразует *BreakpointGraph*(***P***, ***Q***) в *BreakpointGraph*(***P'***, ***Q***) для любого генома ***Q***. В этом примере ***P*** = (+*a* −*b* −*c* + *d*), ***P'*** = (+*a* −*b* −*c* −*d*) и ***Q*** = (+*a* +*c* +*b* −*d*)

В более широком смысле мы можем рассматривать серию двойных разрывов, преобразующих ***P*** в ***Q***, как серию двойных разрывов, преобразующих *BreakpointGraph*(***P***, ***Q***) в *BreakpointGraph*(***Q***, ***Q***), тривиальный граф точек останова. Это означает, что решение проблемы двойного разрыва для геномов ***P*** и ***Q*** эквивалентно нахождению кратчайшей серии двойных разрывов, преобразующих *BreakpointGraph*(***P***, ***Q***) в тривиальный граф точек останова. На рисунке ниже показано преобразование графа точек останова с *Cycles*(***P***, ***Q***) = 2 в тривиальный граф точек останова с *Cycles*(***Q***, ***Q***) = 4 с использованием двух двойных разрывов.

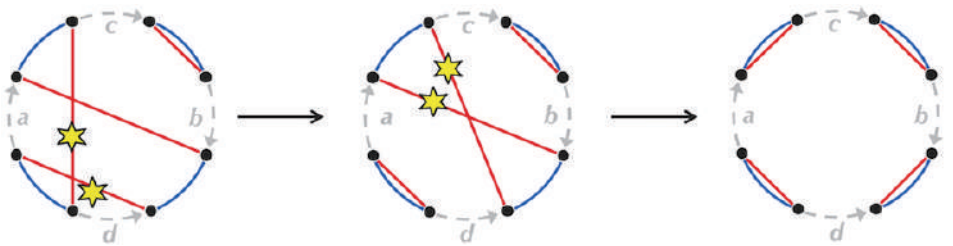


Рис. 6.19 Каждое преобразование с двойным разрывом генома ***P*** в геном ***Q*** соответствует преобразованию *BreakpointGraph*(***P***, ***Q***) в *BreakpointGraph*(***Q***, ***Q***). В показанном примере количество красно-синих циклов на графе увеличивается с *Cycles*(***P***, ***Q***) = 2 до *Cycles*(***Q***, ***Q***) = *Blocks*(***P***, ***Q***) = 4

Поскольку каждое преобразование ***P*** в ***Q*** преобразует *BreakpointGraph*(***P***, ***Q***) в тривиальный граф точек останова *BreakpointGraph*(***Q***, ***Q***), любая сортировка по двойной разрывам увеличивает количество красно-синих циклов на

$$Cycles(\mathbf{Q}, \mathbf{Q}) - Cycles(\mathbf{P}, \mathbf{Q}).$$

ОСТАНОВИТЕСЬ и задумайтесь. Какой вклад в это увеличение может внести каждый двойной разрыв? Другими словами, если ***P'*** получается из ***P*** путем двойного разрыва, насколько больше может быть *Cycles*(***P'***, ***Q***), чем *Cycles*(***P***, ***Q***)?

Вычисление дистанции двойного разрыва

Теорема о точке останова утверждает, что реверсирование, примененное к хромосоме *P*, может уменьшить *Breakpoints*(*P*) не более чем на 2. Теперь мы докажем, что двойной разрыв, примененный к мультихромосомному геному *P*, может увеличить *Cycles*(*P, Q*) не более чем на 1, т. е. для любого двойной

разрыва, преобразующего P в P', и для любого генома Q $Cycles(P', Q)$ не может превышать $Cycles(P, Q) + 1$.

Теорема о цикле. При данных геномах P и Q любой двойнй разрыв, примененный к P, может увеличить $Cycles(P, Q)$ не более чем на 1.

Доказательство. На рисунке ниже представлены три случая, которые иллюстрируют, как двойной разрыв, примененный к P, может повлиять на граф точек останова. Каждый двойной разрыв влияет на два красных ребра, которые либо принадлежат одному и тому же циклу, либо двум разным циклам в $BreakpointGraph(P, Q)$. В первом случае двойной разрыв либо не меняет $Cycles(P, Q)$, либо увеличивает его на 1. Во втором случае он уменьшает $Cycles(P, Q)$ на 1. ■

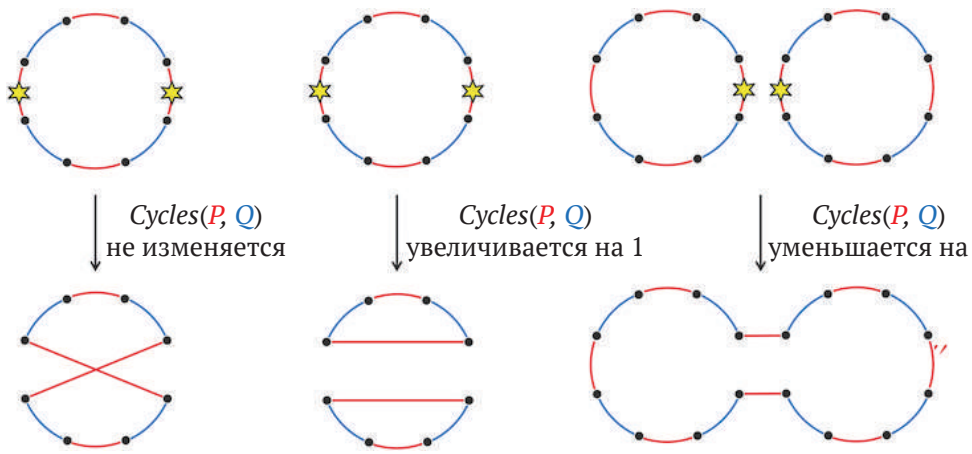


Рис. 6.20 Три случая, иллюстрирующие, как двойной разрыв может повлиять на граф точек останова; серые ребра не показаны

Хотя предыдущее доказательство короткое и интуитивно понятно, это не формальное доказательство, а скорее приглашение изучить рисунок. Если вас интересуют более строгие математические аргументы, прочтите следующее доказательство.

Доказательство. Двойной разрыв добавляет два новых красных ребра и, таким образом, образует не более 2 новых циклов (содержащих два новых красных ребра) в $BreakpointGraph(P, Q)$. В то же время он удаляет два красных ребра и удаляет как минимум один старый цикл (содержащий два старых ребра) из $BreakpointGraph(P, Q)$. Таким образом, количество красно-синих циклов в $BreakpointGraph(P, Q)$ увеличивается не более чем на $\mathbf{2} - \mathbf{1} = 1$, что означает, что $Cycles(P, Q)$ увеличивается не более чем на 1. ■

Напомним, что существуют перестановки, для которых отсутствие реверсий уменьшает количество точек останова, и этот факт разрушил наши надежды на жадный алгоритм сортировки по реверсиям, уменьшающий количество точек

останова на каждом шаге. В случае двойных разрывов (в геномах с кольцевыми хромосомами) мы теперь знаем, что каждый двойной разрыв может увеличить *Cycles*(*P, Q*) не более чем на 1. Но всегда ли можно найти двойной разрыв, увеличивающий *Cycles*(*P, Q*) на 1? Как показывает следующая теорема, ответ положительный.

Теорема о дистанции двойного разрыва. Дистанция двойного разрыва между геномами *P* и *Q* равна *Blocks*(*P, Q*) – *Cycles*(*P, Q*).

Доказательство. Мы называем преобразование *P* в *Q* с помощью двойных разрывов **сортировкой по двойной разрывам**. Ранее мы знали, что любая сортировка по двойному разрыву должна увеличивать количество альтернативных циклов на *Cycles*(*Q, Q*) – Cycles(P, Q), что равно *Blocks*(*P, Q*) – *Cycles*(*P, Q*), поскольку *Blocks*(*P, Q*) = *Cycles*(*Q, Q*). Теорема о циклах подразумевает, что каждый двойной разрыв увеличивает количество циклов в графе точек останова не более чем на 1. Отсюда, в свою очередь, немедленно следует, что *d*(*P, Q*) не меньше, чем *Blocks*(*P, Q*) – *Cycles*(*P, Q*). Если *P* не равно *Q*, то в *BreakPointGraph*(*P, Q*) должен быть **нетривиальный цикл**, т. е. цикл с более чем двумя ребрами. Как показано на рисунке из теоремы о циклах (рис. 6.20), любой нетривиальный цикл в графе точек останова можно разделить на два цикла с помощью двойного разрыва, подразумевая, что мы всегда можем найти двойной разрыв, увеличивающий число красно-синих циклов на 1. Следовательно, *d*(*P, Q*) равно *Blocks*(*P, Q*) – *Cycles*(*P, Q*). ■

Вам, возможно, интересно, как представление графа, которое мы использовали для графов точек останова, может быть преобразовано в список смежности. Ведь мы даже не пометили узлы этого графа! Ознакомьтесь с разделом **ЗАРЯДНАЯ СТАНЦИЯ: От геномов к графу точек останова**, чтобы узнать, как преобразовать геном в граф, с которым проще работать в наших реализациях.

Теперь мы знаем, как вычислить дистанцию двойного разрыва, но мы также хотели бы реконструировать набор двойных разрывов, составляющих кратчайший путь между двумя геномами. Эта задача называется **сортировкой с двойным разрывом**, которую мы представляем вам в качестве упражнения.

Задача сортировки с двойным разрывами: *найти кратчайшее преобразование одного генома в другой с помощью двойных разрывов.*

Input: два генома с кольцевыми хромосомами в одном наборе синтенных блоков.

Output: последовательность геномов, полученная в результате применения кратчайшей последовательности двойных разрывов, преобразующих один геном в другой.

☞ **Загрузить данные 6.2**

Теорема о точках останова гарантирует, что всегда должен быть двойной разрыв, уменьшающий количество красно-синих циклов в графе точек останова на 1. Однако она не говорит нам, как найти такой двойной разрыв. Как это может быть сделано? Посмотрите раздел **ЗАРЯДНАЯ СТАНЦИЯ: Решение задачи сортировки с двойными разрывами**, если вам интересно.

Доказав формулу $d(P, Q) = Blocks(P, Q) - Cycles(P, Q)$ для дистанции двойного разрыва между геномами с несколькими кольцевыми хромосомами, мы задаемся вопросом, можем ли мы найти аналогичную формулу для реверсного расстояния между одиночными линейными хромосомами.

Оказывается, существует полиномиальный алгоритм сортировки рекомбинаций по реверсиям, дающий точную формулу для реверсного расстояния! Хотя этот алгоритм также опирается на понятие графа точек останова, он, к сожалению, слишком сложен, чтобы представлять его здесь (подробнее см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Сортировка линейных рекомбинаций по реверсиям**).

Граф точек останова, построенный на 280 синтенных блоках человек–мышь, содержит 35 альтернативных циклов, так что расстояние двойного разрыва между этими геномами составляет 280 – 35 = 245. Опять же, мы не знаем точно, сколько двойных разрывов произошло в блоке за последние 75 млн лет, но мы уверены, что было *по крайней мере* 245 шагов. Запомните этот факт, так как он окажется важным в следующем разделе.

Горячие точки рекомбинации в геноме человека

Модель случайных разрывов соответствует теореме о дистанции двойного разрыва

Вероятно, с самого начала главы вы ожидали, что в конце концов мы будем выступать против модели случайных разрывов. Но вам все еще может быть неясно, как для этого можно использовать дистанцию двойного разрыва.

Теорема о горячих точках рекомбинации. В геноме человека есть горячие точки рекомбинации.

Доказательство. Вспомним, что если модель случайных разрывов верна, то N реверсий, примененных к хромосоме, дадут $2N$ точек разрыва, поскольку вероятность того, что два соседних участка генома будут использоваться в качестве точки разрыва более чем одной реверсии, очень мала. Эти $2N$ точек разрыва образуют $2N + 1$ синтенных блоков в случае линейной хромосомы и $2N$ синтенных блоков в случае кольцевой хромосомы. Поскольку существует 280 синтенных блоков человек–мышь, должно было быть примерно 280/2 = 140 двойных разрывов на расходящихся эволюционных путях между людьми

и мышами. Однако теорема о дистанции двойного разрыва говорит нам, что на этом эволюционном пути было по крайней мере 245 двойных разрывов.

ОСТАНОВИТЕСЬ и задумайтесь. 245 ≈ 140?

Поскольку 245 намного больше 140, мы пришли к противоречию, следовательно, одно из наших предположений неверно! Но единственное допущение, которое мы сделали в этом доказательстве, было «Если модель случайных разрывов верна…». Таким образом, это допущение является неверным. ■

Предыдущий аргумент, не являющийся математическим доказательством, тем не менее логически обоснован. Он предлагает пример доказательства **от противного**, где мы начинаем с утверждения, которое намереваемся опровергнуть, а затем демонстрируем, что это утверждение не может быть истинным. В результате теоремы о горячих точках рекомбинации мы заключаем, что на пути эволюции человек–мышь было «повторное использование точки останова»; т. е. некоторые точки останова использовались в качестве конечных точек более чем одной рекомбинации генома. Это повторное использование точки останова было обширным, о чем свидетельствует большое соотношение между фактической дистанцией двойного разрыва и дистанцией двойного разрыва, которое дала бы модель случайных разрывов (245/140 = 1,75).

Конечно, наши аргументы должны быть статистически обоснованными, чтобы гарантировать, что расхождение между предсказанием модели случайных разрывов и дистанцией двойного разрыва является значительным. В конце концов, даже несмотря на то, что геномы велики, все еще существует небольшая вероятность того, что случайно выбранные двойной разрывы могут иногда ломать геном более одного раза в небольшом интервале. Необходимый статистический анализ выходит за рамки этой книги.

Модель хрупких разрывов

Но подождите, а как насчет аргумента Надо и Тейлора в пользу модели случайных разрывов? Мы, конечно, не можем игнорировать тот факт, что распределения длин синтенных блоков человека и мыши напоминают экспоненциальные.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы найти что-нибудь неправильное в логике Надо и Тейлора?

Аргумент Надо и Тейлора в пользу модели случайных разрывов иллюстрирует классическую логическую ошибку. Верно, что если разрыв случайный, то гистограмма длин синтенных блоков должна следовать экспоненциальному распределению. Но совершенно другое утверждение – заключить, что только потому, что длины синтенных блоков подобны экспоненциальному распределению, полом-

ка должна быть случайной. Распределение длин синтенных блоков, безусловно, подтверждает модель случайных разрывов, но не доказывает ее правильность.

Тем не менее любая альтернативная гипотеза, которую мы выдвинули для модели случайных разрывов, должна учитывать наблюдение, что распределение длин синтенных блоков для геномов человека и мыши является приблизительно экспоненциальным.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы предложить другую модель эволюции хромосом, которая объясняет горячие точки рекомбинации и согласуется с экспоненциальным распределением длин синтенных блоков?

Противоречие **модели случайных разрывов** привело к альтернативной модели эволюции хромосом **модели хрупких разрывов**, которая была предложена в 2003 году. Эта модель утверждает, что геном каждого млекопитающего представляет собой мозаику из длинных сплошных областей, которые редко подвергаются рекомбинациям, а также коротких хрупких областей, которые служат предполагаемыми очагами рекомбинаций и составляют лишь небольшую часть генома; у людей и мышей эти хрупкие области составляют примерно 3 % генома. Мы не утверждаем, что каждая хрупкая область представляет собой горячую точку рекомбинации (т. е. использовалась в качестве конечной точки нескольких рекомбинаций), но теорема о горячих точках рекомбинации подразумевает, что многие хрупкие области должны быть горячими точками рекомбинации.

Если мы снова применим бритву Оккама, то наиболее разумный способ допустить экспоненциально распределенные длины синтенных блоков – это если сами хрупкие области располагаются в геноме случайным образом; это контрастирует с моделью случайных разрывов, которая постулирует, что точки разрыва рекомбинаций распределяются случайным образом. В самом деле, *случайный* выбор точек разрыва в *случайно* расположенных хрупких областях мало чем отличается от случайного выбора конечных точек рекомбинации во всем геноме. Тем не менее, хотя теперь у нас есть модель, которая соответствует нашим наблюдениям, остается много вопросов. Например, неясно, где расположены хрупкие области, соответствующие горячим точкам рекомбинации, или что в первую очередь вызывает хрупкость генома в том или ином месте.

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующее утверждение: «Экспоненциальное распределение длин синтенных блоков и обширное повторное использование точек останова подразумевают, что модель хрупкого разрыва должна быть верной». Логичен ли этот аргумент?

Суть, которую мы пытаемся подчеркнуть, задавая предыдущий вопрос, заключается в том, что мы никогда не сможем доказать научную теорию, по-

добную модели хрупкого разрыва, таким же образом, как мы доказали одну из математических теорем в этой главе. На самом деле многие биологические теории основаны на аргументах, которые математик счел бы ошибочными; логическая структура, используемая в биологии, сильно отличается от используемой в математике. Возьмем исторический пример: ни Дарвин, ни кто-либо другой никогда не доказывали, что эволюция путем естественного отбора является единственным – или даже наиболее вероятным – объяснением того, как развилась жизнь на Земле!

Мы уже приводили много причин, по которым профессора биологии хотели бы отправить нас в учебный лагерь Биология-101, но теперь нас, вероятно, схватят и бросят в ГУЛАГ вместе со сторонниками теории разумного замысла. (Одна из разновидностей креационизма. – *Прим. ред.*) Однако факт остается фактом: даже дарвинизм не является неопровержимым; в XX веке эта теория была ревизована в неодарвинизм, и нет никаких сомнений в том, что она продолжит свое существование и развитие.

Эпилог. Конструирование синтенных блоков

На протяжении всего нашего обсуждения рекомбинаций генома мы предполагали, что нам заранее даны синтенные блоки. В этом разделе мы опишем один из способов построения синтенных блоков из геномных последовательностей.

Геномные точечные диаграммы и общие k-меры

Биологи иногда представляют повторяющиеся k-меры в строке как набор точек на плоскости; точка с координатами (x, y) представляет идентичные k-меры, встречающиеся в позициях x и y в строке. На рисунке ниже представлены две из таких геномных точечных диаграмм.

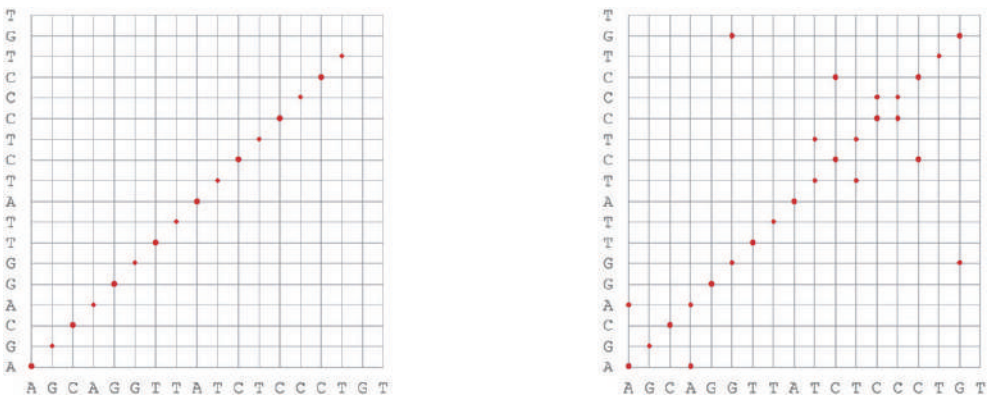


Рис. 6.21 Визуализация повторяющихся k-меров в строке AGCAGGTTATCTCCCTGT для $k = 3$ (слева) и $k = 2$ (справа)

Конечно, поскольку ДНК является двухцепочечной молекулой, мы должны расширить понятие повторяющихся *k*-меров, чтобы учесть повторы, встречающиеся на комплементарной цепи. На рисунке ниже синие точки (*x, y*) указывают на то, что *k*-меры, начинающиеся в позициях *x* и *y* строки, реверсно комплементарны.

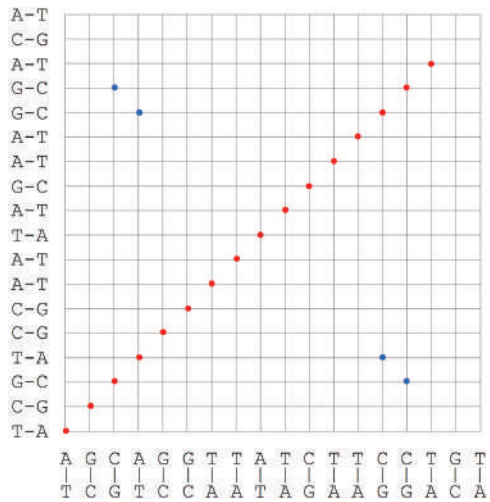


Рис. 6.22 Мы добавляем синие точки к графику в верхнем левом углу, чтобы указать реверсно комплементарные *k*-меры. Например, CCT и AGG являются реверсно комплементарными 3-мерами в AGCAGGTTATCTTCCTGT, что приводит к синей точке в положении (14, 3)

Поиск общих k-меров

Напомним, что синтенный блок определяется множеством сходных генов, встречающихся в одном и том же порядке в двух геномах. Поскольку похожие гены часто имеют одни и те же *k*-меры (при правильно выбранном значении *k*), давайте сначала найдем положения всех *k*-меров, которые являются общими для X-хромосом человека и мыши. Если мы выберем *k* достаточно большим (например, *k* = 30), то маловероятно, что общие *k*-меры представляют ложное сходство. Более вероятное объяснение состоит в том, что они происходят из родственных генов (или общих повторов) в геномах человека и мыши.

Формально мы говорим, что *k*-мер является **общим** для двух геномов, если в каждом геноме присутствует либо *k*-мер, либо его реверсный комплемент. Ниже на рис. 6.23 представлены четыре пары 3-меров (выделены жирным шрифтом), которые являются общими для AAACTCATC и TTTCAAATC; обратите внимание, что вторая пара 3-меров обратно дополняет друг друга.

Общий *k*-мер может быть представлен упорядоченной парой (*x, y*), где *x* – начальное положение *k*-мера в первом геноме, а *y* – начальное положение *k*-мера во втором геноме. Для приведенных выше геномов этими общими *k*-мерами

являются (0,4), (0,0), (4,2) и (6,6). (Обратите внимание, что мы используем индексацию строк с отсчетом, начиная от 0.)

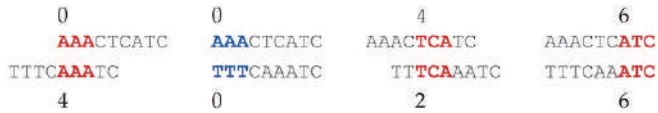


Рис. 6.23 Четыре пары общих 3-меров

Упражнение. Сколько существует общих 2-меров AAACTCATC и TTTCAAATC?

Далее мы можем обобщить геномную точечную диаграмму для анализа общего содержания k-меров в двух геномах. Мы окрашиваем точку (x, y) в красный цвет, если два генома имеют общий k-мер, начинающийся в соответствующих позициях x и y. Мы окрашиваем (x, y) в синий цвет, если два генома имеют реверсно комплементарные k-меры в этих начальных положениях.

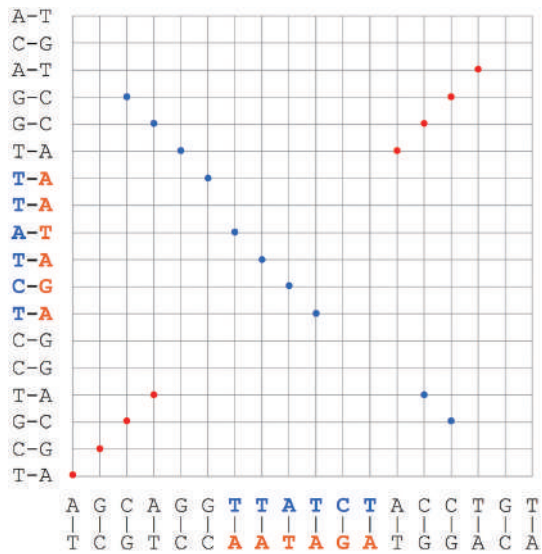


Рис. 6.24 Геномная точечная диаграмма, показывающая общие 3-меры между AGCAGG**TTATCT**ACCTGT и AGCAGG**AGATAA**ACCTGT. Вторая последовательность возникла из первой путем реверсии сегмента **TTATCT**. Каждая точка (x, y) соответствует k-меру, общему для двух геномов. Красные точки указывают на идентичные общие k-меры, тогда как синие точки указывают реверсно комплементарные k-меры. Обратите внимание, что точечный граф имеет шесть ложных синих точек на диаграмме: четыре в верхнем левом углу и две в нижнем правом углу. Вы также заметите, что красные точки могут быть соединены в линейные сегменты с наклоном 1, а синие точки могут быть соединены в линейные сегменты с наклоном –1. Получившиеся три синтенных блока (AGCAGG, **TTATCT** и ACCTGT) соответствуют трем диагоналям (каждая из которых состоит из четырех точек) на геномной точечной диаграмме

Задача общих *k*-меров: *по двум строкам найти все их общие k-меры.*

Input: целое число *k* и две строки.

Output: все *k*-меры, общие для этих строк, в виде упорядоченных пар (*x, y*), соответствующих начальным позициям этих *k*-меров в соответствующих строках.

☞ **Загрузить данные 6.3**

ОСТАНОВИТЕСЬ и задумайтесь. Вычислите ожидаемое количество 30-меров, разделяемых двумя случайными строками, каждая длиной в миллиард нуклеотидов.

Поскольку загрузка длинных хромосом человека и мыши занимает много времени, вместо этого мы решим задачу общих *k*-меров для бактерий *E. coli* и *S. enterica*.

☞ **Загрузить геном *E. Coli* 6.4**

☞ **Загрузить геном *S. Enterica* 6.5**

Упражнение. Сколько общих 30-меров имеют общие геномы *E. coli* и *S. enterica*?

Можно показать, что ожидаемое количество общих 30-меров между двумя случайными последовательностями длиной 5 млн нуклеотидов составляет примерно $2 \cdot (5 \cdot 10^6) 2/430 \approx 1/20\,000$. Тем не менее решение проблемы общих *k*-меров для *E. coli* и *S. enterica* дает более 200 000 пар (*x, y*), соответствующих общим 30-мерам. Удивительно большое количество общих 30-меров указывает на то, что *E. coli* и *S. enterica* являются близкими родственниками, сохранившими многие сходные гены, унаследованные от их общего предка. Однако эти гены могут быть расположены в другом порядке у двух видов: как мы можем сделать вывод о синтенных блоках из общих *k*-меров этих геномов? Геномная точечная диаграмма для *E. coli* и *S. enterica* показана на рис. 6.25.

ОСТАНОВИТЕСЬ и задумайтесь. Видите ли вы синтенные блоки на этом рисунке?

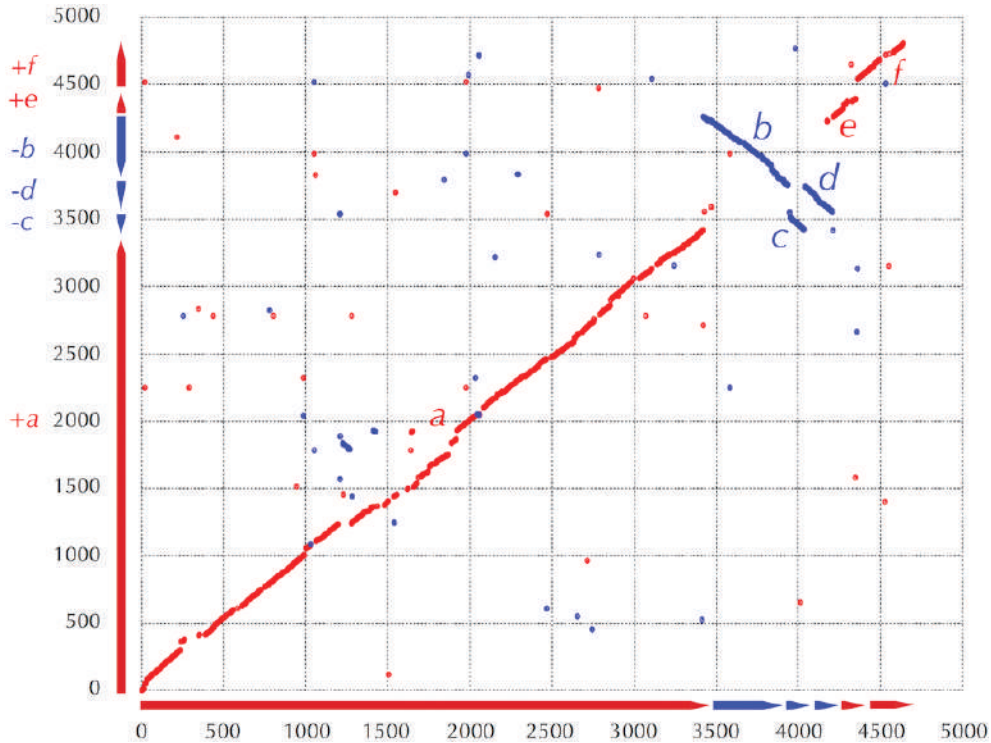


Рис. 6.25 Геномный точечный граф *E. coli* (горизонтальная ось) и *S. enterica* (вертикальная ось) для $k = 30$. Каждая точка (x, y) соответствует k-меру, общему для двух геномов. Красные точки обозначают идентичные общие k-меры, тогда как синие точки указывают точки, реверсно комплементарные общим k-мерам. Каждая ось измеряется в килобазах (тысячах пар оснований)

Построение синтенных блоков из общих k-меров

Геномная точечная диаграмма на рисунке выше указывает на шесть областей сходства в виде точек, которые слипаются в приблизительно диагональные сегменты. Эти сегменты помечены буквами ***a***, ***b***, ***c***, ***d***, ***e*** и ***f*** в соответствии с порядком их появления в геноме *E. coli*; мы игнорируем меньшие диагонали, такие как короткая синяя диагональ, начинающаяся с позиции 1,3 млн в *E. coli* и с позиции 1,9 млн в *S. enterica*. Например, в то время как ***a*** соответствует длинному диагональному сегменту наклона 1, который охватывает примерно первые 3,5 млн позиций в обоих геномах, ***b*** соответствует более короткому диагональному сегменту наклона –1, который начинается незадолго до положения 3,5 млн в *E. coli* и вскоре после позиции 4 млн у *S. enterica*. Хотя ***b*** на точечной диаграмме кажется маленьким, не обманывайтесь масштабом рисунка; ***b*** имеет длину более 100 000 нуклеотидов и содержит почти 100 генов.

Сегменты ***a***, ***b***, ***c***, ***d***, ***e*** и ***f*** дают нам синтенные блоки, которые мы искали. Если спроецировать эти блоки на оси *x* и *y*, то порядок блоков на каждой оси соответствует порядку синтенных блоков в соответствующей бактерии. Порядок синтенных блоков в *E. coli* (по оси абсцисс) следующий: (***+a +b +c +d +e +f***), а порядок блоков *S. enterica* (ось *y*) – (***+a –c –d –b +e +f***). Обратите внимание, что синим буквам в *S. enterica* присвоен отрицательный знак, потому что эти блоки были построены из реверсно комплементарных *k*-меров. Точечный граф также показывает, каковы направления блоков – они соответствуют диагоналям на точечном графе с наклоном 1 (блоки со знаком «+») и наклоном –1 (блоки со знаком «–»).

Поэтому мы представили связь между двумя бактериальными геномами, используя всего шесть синтенных блоков. Конечно, это упрощение потребовало, чтобы мы отбросили некоторые точки на точечном графе, соответствующие крошечным областям подобия, которые не превышают пороговую длину, чтобы считаться синтенными блоками.

Теперь мы готовы построить одиннадцать синтенных блоков человека и мыши, первоначально представленных в начале главы, но, поскольку X-хромосомы человека и мыши довольно длинные, вместо этого мы предоставим вам все позиции (*x, y*), в которых они имеют значительные сходства. На рисунке ниже представлена результирующая геномная точечная диаграмма для X-хромосом человека и мыши, где каждая точка представляет собой длинную аналогичную область, а не общий *k*-мер. Наши глаза сразу находят на этом графе одиннадцать диагоналей, соответствующих синтенным блокам X-хромосомы человека и мыши, – задача решена!

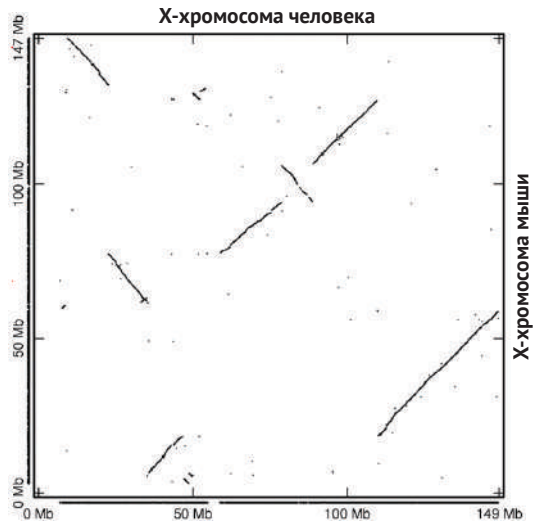


Рис. 6.26 Геномная точечная диаграмма X-хромосом человека и мыши, представляющая все положения (*x, y*), в которых они имеют значительное сходство. Эти положения были использованы в Pevzner & Tesler, 2003, для опровержения модели случайных разрывов. В отличие от предыдущего точечного графа мы не различаем красные и синие точки

Задача о синтенных блоках: *найти диагонали на геномной точечной диаграмме.*

Input: набор точек *DotPlot* на диаграмме.

Output: набор диагоналей в *DotPlot*, представляющих синтенные блоки.

К сожалению, остается неясным, как написать программу, которая делает то, что нашим глазам казалось таким простым; мы надеемся, что вы уже заметили, что задача синтенных блоков не является хорошо сформулированной вычислительной задачей. Как мы уже упоминали, диагонали на точечной диаграмме Х-хромосомы человека и мыши (воспроизведенной выше) не идеальны. Кроме того, внутри диагоналей есть много пробелов, которые не видны человеческому глазу, но станут очевидными, если мы увеличим картинку графа генома. Поэтому неясно, какой метод использует человеческий мозг для преобразования точек в одиннадцать диагоналей на геномной точечной диаграмме.

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем преобразовать способность мозга строить диагонали, которые вы видите выше, в алгоритм, понятный компьютеру?

Синтенные блоки как связные компоненты в графах

Причина, по которой вы можете легко увидеть синтенные блоки на геномной точечной диаграмме, заключается в том, что ваш мозг хорошо группирует соседние точки на изображении. Поэтому, чтобы имитировать этот процесс с помощью компьютера, нам нужно точное понятие кластеризации. Имея набор точек *DotPlot* на плоскости, а также параметр *maxDistance*, построим (неориентированный) **синтенный граф** *SyntenyGraph*(*DotPlot, maxDistance*), соединив две точки *DotPlot* ребром, если расстояние между ними не превышает *maxDistance*.

Каждый граф можно разделить на непересекающиеся связные подграфы, называемые **компонентами связности**. Компоненты связности в *SyntenyGraph*(*DotPlot, maxDistance*) представляют возможные синтенные блоки между двумя геномами. Чем больше значение *maxDistance*, тем меньше компонентов связности, так как ранее изолированные синтенные блоки могут быть объединены в один блок. Этот параметр предлагает компромисс, так как мы хотели бы иметь меньше компонент связности, но не хотим комбинировать ложные синтенные блоки, как показано на рисунке ниже (левая диаграмма).

Синтенный граф для Х-хромосом человека и мыши содержит огромное количество мелких компонент связности (точное количество зависит от нашего выбора параметра *maxDistance*). Однако мы будем игнорировать эти неболь-

шие компоненты связности, поскольку они могут представлять собой ложное сходство. Таким образом, мы вводим параметр *minSize*, представляющий минимальное количество точек в компоненте связности, который мы будем рассматривать как образующий синтенный блок. Наша цель – вернуть все компоненты связности, имеющие узлы как минимум *minSize*.

Параметр *minSize* также предлагает компромисс. Если *minSize* слишком мал, то короткие ложные синтенные блоки усложнят наш анализ. Если *minSize* слишком велик, мы можем отбросить правильные синтенные блоки. Правая диаграмма демонстрирует, что даже после выбора подходящего значения *maxDistance* может не оказаться идеального значения *minSize*; в этом случае тот факт, что один фиктивный синтенный блок имеет тот же размер, что и истинные синтенные блоки, означает, что мы не можем исключить его из рассмотрения. На практике есть надежда, что правильные синтенные блоки будут достаточно большими, чтобы предотвратить эту проблему.

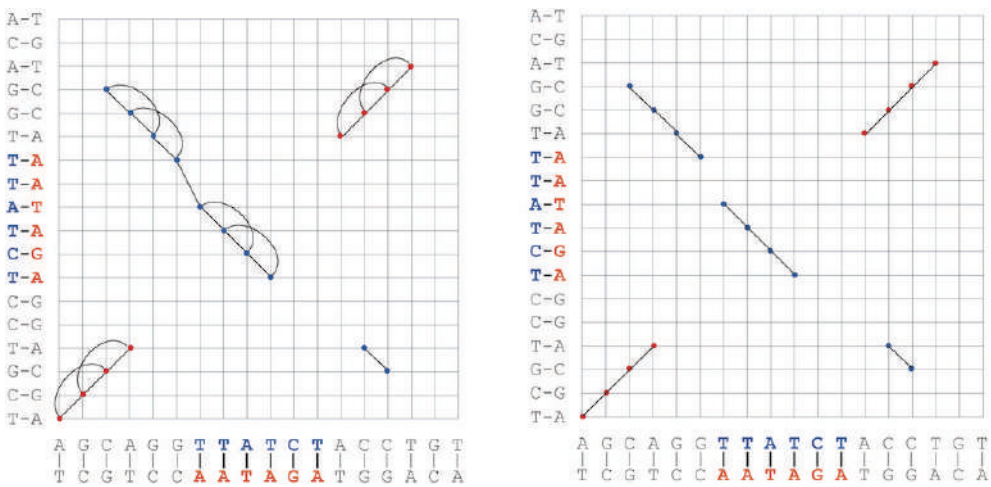


Рис. 6.27 (Слева) Граф *SyntenyGraph*(*DotPlot*, 3), построенный из геномного точечного графа AGCAGGTTATCTCCCTGT и AGCAGGAGATAACCCTGT для $k = 3$. Обратите внимание, что ложный синтенный блок в левом верхнем углу, к сожалению, был объединен с правильным синтенным блоком. (Справа) Граф *SyntenyGraph*(*DotPlot*, 3), позволяющий различать неправильные и правильные синтенные блоки, которые ранее были подвергнуты конкатенации (склеиванию). Однако обратите внимание, что, даже если мы установим *minSize* равным 3, мы сможем отбросить маленький ложный синтенный блок в правом нижнем углу, но не сможем – более крупный в левом верхнем углу, поскольку он имеет тот же размер, что и правильные синтенные блоки

Когда мы строим синтенный граф для Х-хромосом человека и мыши, мы находим огромное количество мелких компонент связности (точное количество зависит от нашего выбора параметра *maxDistance*). Однако мы будем игнорировать эти небольшие компоненты связности, поскольку они могут представлять

собой ложное сходство. Таким образом, мы вводим параметр *minSize*, представляющий минимальное количество точек в компоненте связности, которую мы будем рассматривать как формирующую синтенный блок. Наша цель – вернуть все компоненты связности, имеющие узлы как минимум *minSize*.

SyntenyBlocks(*DotPlot, maxDistance, minSize*)
построить *SyntenyGraph*(*DotPlot, maxDistance*)
найти компоненты связности в *SyntenyGraph*(*DotPlot, maxDistance*)
output компоненты связности, содержащие по крайней мере *minSize* узлов, в качестве кандидатов на синтенные блоки

Как показано на рис. 6.28, алгоритм **SyntenyBlocks** имеет тенденцию разбивать одну диагональ (как воспринимается человеческим глазом) на несколько диагоналей из-за промежутков, которые превышают параметр *maxDistance*. Однако такое разбиение не является задачей, так как разорванные диагонали можно впоследствии объединить в единый (агрегированный) синтенный блок.

ОСТАНОВИТЕСЬ и задумайтесь. Мы определили синтенные блоки как большие компоненты связности в *SyntenyGraph*(*DotPlot*, *maxDistance*), но не описали, как определить, где эти синтенные блоки расположены в исходных геномах. Используя предыдущий рисунок в качестве подсказки, разработайте алгоритм для поиска этой информации.

Теперь вы должны быть готовы решить сложную задачу и обнаружить, что выбор параметров является одним из темных секретов исследований в области биоинформатики.

Заключительная задача. Используя набор анкеров, указывающих области сходства в геномах человека и мыши, постройте синтенные блоки для геномов человека и мыши и рассчитайте дистанцию двойного разрыва между этими геномами, используя построенные вами синтенные блоки. Как меняется это расстояние в зависимости от параметров *maxDistance* и *minSize*?

☞ **Загрузить данные «Анкеры человек–мышь» 6.6**

Примечание к набору данных. Каждая строка соответствует анкеру или короткому интервалу, сходному в геномах человека и мыши. Восемь элементов в строке представляют:

1) индекс анкера;
2) номер хромосомы человека, на которой находится анкер;

3) исходное положение анкера в хромосоме человека;
4) конечную позицию анкера в хромосоме человека;
5) номер хромосомы мыши, на которой находится анкер;
6) исходное положение анкера в хромосоме мыши;
7) конечную позицию анкера в хромосоме мыши;
8) ориентацию анкера («+» или «–»).

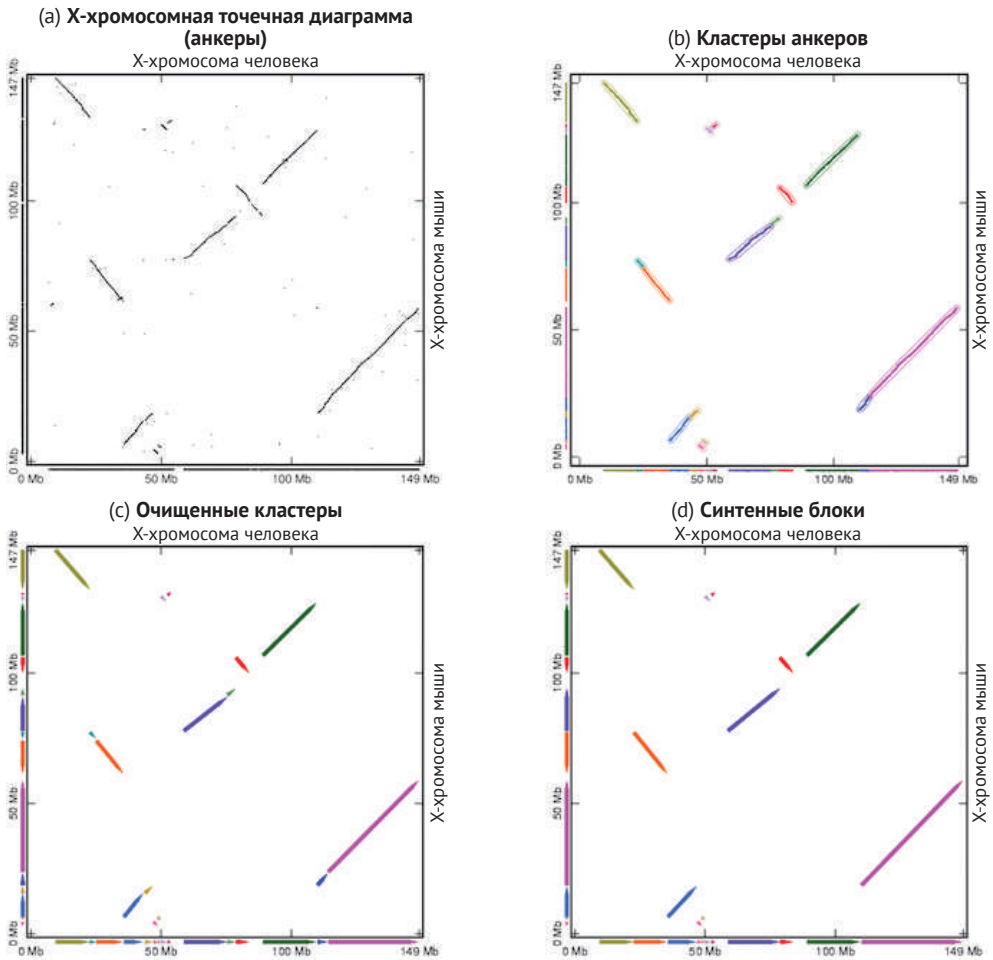


Рис. 6.28 От локальных сходств до синтенных блоков. (a) Геномная точечная диаграмма для X-хромосом человека и мыши. (b) Кластеры (компоненты связности) точек на геномной точечной диаграмме формируются путем построения синтенного графа. (c) Очищенные (выпрямленные) кластеры из синтенного графа превращают каждый кластер из части (b) в точную диагональ наклона ±1. (d) Агрегированные синтенные блоки. Проекция синтенных блоков на осях *x* и *y* приводит к расположению синтенных блоков в соответствующих геномах человека и мыши (+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11) и (+1 –7 +6 –10 +9 –8 +2 –11 –3 +5 +4)

Зарядные станции

От геномов к графу точек останова

Наша цель – подсчитать количество циклов в графе точек останова и, следовательно, решить задачу дистанции двойного разрыва. Однако сначала нужно будет получить удобное представление геномов в виде графов. В основном тексте мы представили кольцевую хромосому, превратив каждый синтенный блок в направленное ребро, а затем соединив соседние синтенные блоки в хромосоме ненаправленными красными ребрами. Хотя это дало нам способ визуализации геномов, не сразу понятно, как представить этот граф со списком смежности.

Для данного генома P мы будем представлять его синтенные блоки не буквами, а целыми числами от 1 до $n = |P|$. Например, $(+a\ -b\ -c\ +d)$ будет представлено как (+1 −2 −3 +4), как показано на рисунке ниже.

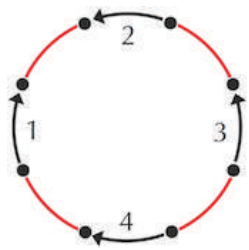


Рис. 6.29 Геном P, представленный в виде графа

Далее мы преобразуем направленные черные ребра P в неориентированные следующим образом. Для направленного ребра, помеченного целым числом x, мы назначаем узел в «голове» этого ребра как x_h, а узел в «хвосте» этого ребра – как x_t. Например, на рисунке ниже (слева) мы заменим направленное ребро с пометкой «2» на неориентированное ребро, соединяющее узлы 2_t и 2_h (справа). Это приводит к циклической последовательности узлов (1_t, 1_h, 2_h, 2_t, 3_h, 3_t, 4_t, 4_h).

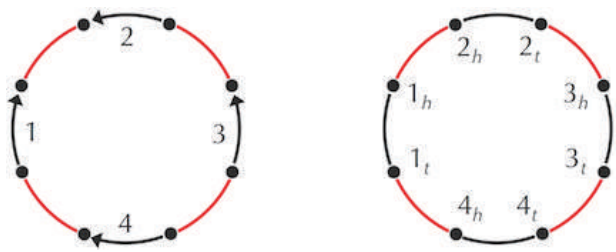


Рис. 6.30 Преобразованный граф генома P

Наконец, чтобы еще больше упростить анализ этого графа, вместо использования x_h и x_t для обозначения головы и хвоста синтенного блока x мы будем использовать целые числа $2x$ и $2x - 1$ соответственно. Как показано ниже, при таком кодировании исходный геном (+1 –2 –3 +4) преобразуется в циклическую последовательность узлов (1, 2, 4, 3, 6, 5, 7, 8). Обратите внимание, что эта кодировка применяет индексацию на основе 1; вы могли бы так же легко пометить узлы 0–7, что дало бы циклическую последовательность (0, 1, 3, 2, 5, 4, 6, 7).

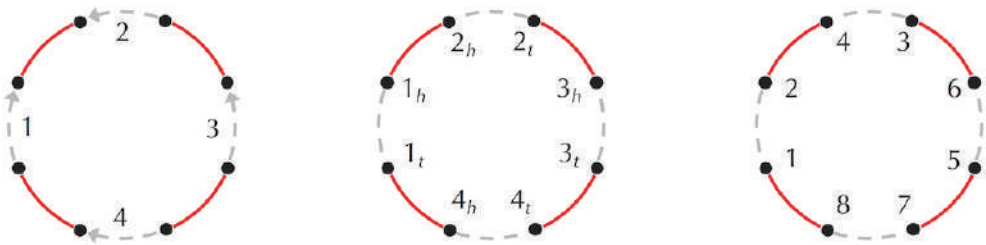


Рис. 6.31 Переобозначение графа *P*

ОСТАНОВИТЕСЬ и задумайтесь. Обратимо ли это преобразование? Другими словами, если бы мы дали вам циклическую последовательность узлов, пронумерованных от 1 до $2n$, смогли бы вы реконструировать хромосому с n синтенными блоками, из которой мы эту последовательность получили?

Следующий псевдокод обходит промежуточный этап назначения узлов «голова» и «хвост», чтобы преобразовать одну круговую хромосому *Chromosome* = (*Chromosome*$_1$, …, *Chromosome*$_n$) в цикл, представленный в виде последовательности целых чисел *Nodes* = (*Nodes*$_1$, …, *Nodes*$_{2n}$).

```
ChromosomeToCycle(Chromosome)
    for j ← 1 до|Chromosome|
        i ← Chromosome_j
        if i > 0
            Nodes_{2j-1} ←2i – 1
            Nodes_{2j} ← 2i
        else
            Nodes_{2j-1} ← –2i
            Nodes_{2j} ←-2i – 1
    return Nodes
```

Этот процесс на самом деле обратим, как описано в следующем псевдокоде.

```
CycleToChromosome(Nodes)
    for j ← 1 до |Nodes|/2
        if Nodes_{2j−1} < Nodes_{2j}
            Chromosome_j ← Nodes_{2j} /2
        else
            Chromosome_j ← −Nodes_{2j−1} /2
    return Chromosome
```

ChromosomeToCycle генерирует последовательность узлов хромосомы, но явно не добавляет ребра. Любой геном P с n синтенными блоками будет иметь черные ненаправленные ребра $BlackEdges(P) = (1, 2), (3, 4), \ldots, (2n - 1, 2n)$.

Теперь определим $ColoredEdges(P)$ как множество цветных ребер в графе P. Для приведенного примера (рис. 6.31, справа) множество $ColoredEdges(P)$ содержит ребра (2, 4), (3, 6), (5, 7) и (8, 1).

Следующий алгоритм конструирует $ColoredEdges(P)$ для генома P. В этом псевдокоде мы будем предполагать, что n-элементный массив $(a_1,\ldots,a_n)$ имеет невидимый $(n + 1)$-й элемент, равный его первому элементу, т. е. $a_{n+1} = a_1$.

```
ColoredEdges(P)
    Edges ← пустой набор
    for каждой хромосомы Chromosome в P
        Nodes ← ChromosomeToCycle(Chromosome)
        for j ← 1 до |Chromosome|
            добавить новое ребро (Nodes_{2j}, Nodes_{2j−1}) к Edges
    return Edges
```

Цветные ребра в графе точек останова P и Q задаются $ColoredEdges(P)$ вместе с $ColoredEdges(Q)$. Обратите внимание, что некоторые ребра в этих двух наборах могут соединять одни и те же два узла, что приводит к тривиальным циклам.

Хотя теперь мы готовы решить задачу о дистанции двойного разрыва, позже мы сочтем полезным реализовать функцию, преобразующую граф генома обратно в геном.

```
GraphToGenome(GenomeGraph)
    P ← пустой набор хромосом
    for каждого цикла Nodes в GenomeGraph
        Nodes ← последовательность узлов в этом цикле (начиная с узла 1)
        Chromosome ← CycleToChromosome(Nodes)
        добавить Chromosome к P
    return P
```

Решение задачи сортировки по двойным разрывам

Примечание Этот раздел использует некоторые обозначения из раздела **ЗАРЯДНАЯ СТАНЦИЯ: От геномов до графа точек останова**.

На рис. 6.32 (вверху) показано, как двойной разрыв заменяет цветные ребра (1, 6) и (3, 8) в графе генома двумя новыми цветными ребрами (1, 3) и (6, 8). Обозначим эту операцию как *2-Break*(1, 6, 3, 8). Обратите внимание, что порядок узлов в этой функции имеет значение, поскольку операция *2-Break*(1, 6, 8, 3) будет представлять другой двойной разрыв, который заменяет (1, 6) и (3, 8) на (1, 8) и (6, 3) (нижний двойной разрыв на рисунке внизу).

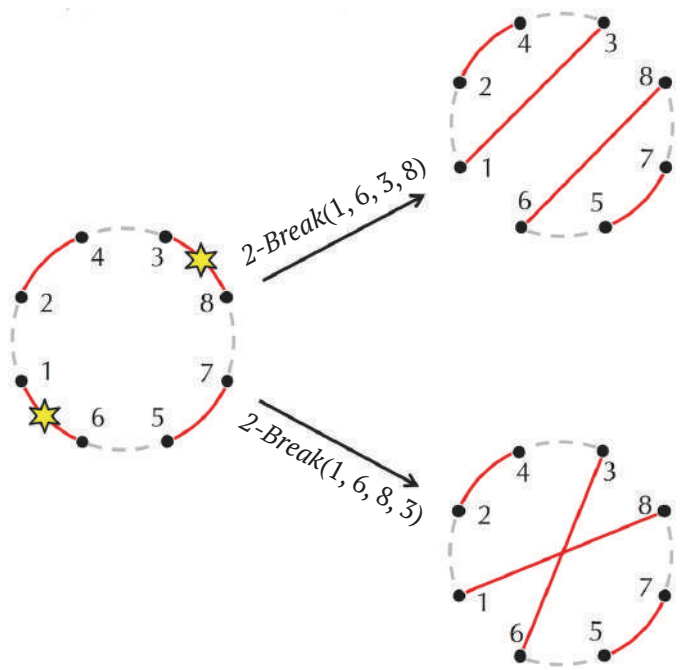


Рис. 6.32 Применение операции *2-Break* к графу генома

Следующий псевдокод описывает, как *2-Break*(i_1, i_2, i_3, i_4) преобразует граф генома.

```
2-BreakOnGenomeGraph(GenomeGraph, i1, i2, i3, i4)
    удалить цветные ребра (i1, i2) и (i3, i4) из GenomeGraph
    добавить цветные ребра (i1, i3) and (i2, i4) к GenomeGraph
    return GenomeGraph
```

Мы можем расширить этот псевдокод до двойной разрыва, определенного на геноме *P*.

2-BreakOnGenome(P, i_1, i_2, i_3, i_4)
  GenomeGraph ← *BlackEdges*(P) и *ColoredEdges*(P)
  GenomeGraph ← **2-BreakOnGenomeGraph**(*GenomeGraph*, i_1, i_2, i_3, i_4)
  P ← **GraphToGenome**(*GenomeGraph*)
  return P

Теперь мы готовы найти ряд промежуточных геномов в кратчайшем преобразовании P в Q с помощью двойных разрывов. Идея нашего алгоритма, называемого **ShortestRearrangementScenario**, состоит в том, чтобы найти двойной разрыв, который увеличит количество красно-синих циклов в графе точек останова на 1. Для этого, как показано на рисунке ниже, мы выбираем произвольное синее ребро в нетривиальном альтернативном красно-синем цикле и делаем двойной разрыв на двух красных ребрах, граничащих с этим синим ребром, чтобы разбить красно-синий цикл на два цикла (по крайней мере один из которых тривиален).

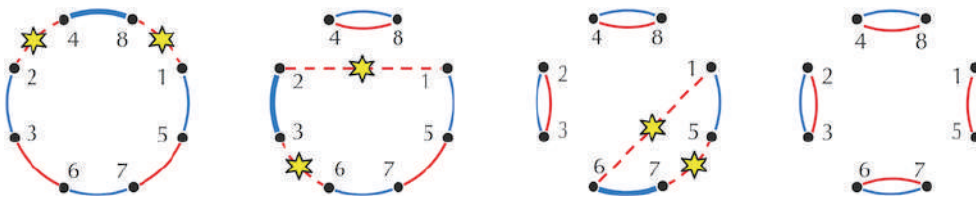


Рис. 6.33 Кратчайшее преобразование с двумя разрывами (слева направо) графа точек останова $P = (+a\ -b\ -c\ +d)$ и $Q = (+a\ +b\ -d\ -c)$. Произвольное синее ребро, выбранное **ShortestRearrangementScenario** на каждом шаге, выделено жирным шрифтом, а красные ребра по обе стороны от него заштрихованы (звездочками обозначены двойные разрывы)

Упражнение. Классифицируйте каждый из трех двойных разрывов, показанных выше, как реверсию, слияние или расщепление.

Наш псевдокод для задачи сортировки с двойными разрывами показан ниже. Этот псевдокод использует концепцию ребра, смежного узлу v, если v является одной из конечных точек ребра.

ShortestRearrangementScenario(P, Q)
  output P
  RedEdges ← *ColoredEdges*(P)
  BlueEdges ← *ColoredEdges*(Q)
  BreakpointGraph ← граф, сформированный из *RedEdges* и *BlueEdges*
  while *BreakpointGraph* содержит нетривиальный цикл *Cycle*

```
        (i1, i2, i3, i4) ← путь, начинающийся с произвольного красного ребра
в нетривиальном красно-синем цикле
        RedEdges ← RedEdges с удаленными ребрами (i1, i2) и (i3, i4)
        RedEdges ← RedEdges с добавленными ребрами (i1, i4) и (i2, i3)
        BreakpointGraph ← граф, сформированный RedEdges и BlueEdges
        P ← 2-BreakOnGenome(P, i1, i2, i3, i4)
        output P
```

Сопутствующие материалы

Почему генный состав Х-хромосом так консервативен?

Хотя Х-хромосомы млекопитающих содержат множество генов, связанных с половым размножением, большинство примерно из 1000 генов на Х-хромосоме не имеют никакого отношения к полу. В идеале они должны экспрессироваться (т. е. транскрибироваться и в конечном итоге транслироваться) примерно в одинаковых количествах у самок и самцов. Но, поскольку у самок две Х-хромосомы, а у самцов только одна, казалось бы, у них все гены на Х-хромосоме должны иметь вдвое больший уровень экспрессии, чем у самок. Этот дисбаланс может привести к проблеме в сложной клеточной системе сдержек и противовесов, лежащей в основе экспрессии генов.

Необходимость сбалансировать экспрессию генов у мужчин и женщин привела к эволюционному возникновению **дозовой компенсации**, или инактивации одной Х-хромосомы у женщин для выравнивания экспрессии генов между полами. Из-за дозовой компенсации содержание гена Х-хромосомы высоко консервативно у разных видов млекопитающих, потому что, если ген выпрыгивает из Х-хромосомы, его экспрессия может удвоиться, создавая генетический дисбаланс.

Открытие геномных рекомбинаций

После того как Стёртевант обнаружил рекомбинации генома у дрозофилы в 1921 году, произошел еще один прорыв, когда было обнаружено, что слюнные железы дрозофилы содержат **политенные клетки**. При нормальном клеточном делении каждая дочерняя клетка получает одну копию генома. Однако в ядрах политенных клеток репликация ДНК происходит многократно в отсутствие клеточного деления. Полученные хромосомы затем соединяются вместе в гораздо более крупные «суперхромосомы», называемые **политенными хромосомами**.

Политенные хромосомы плодовой мушки служат практической цели, используя дополнительную ДНК для увеличения производства транскриптов генов, для производства большего количества липкой слюны. Но ценность политенных хромосом у человека, пожалуй, больше. Когда Стёртевант и его

сотрудник Феодосий Добржанский изучили политенные хромосомы под микроскопом, они смогли своими глазами увидеть работу рекомбинаций в запутанных мутантных хромосомах. В 1938 году Стёртевант и Добржанский опубликовали знаменательную статью с эволюционным деревом, представляющим сценарий рекомбинации с семнадцатью реверсиями для различных видов дрозофилы, первое в истории эволюционное дерево, построенное на основе молекулярных данных.

Экспоненциальное распределение

Схема Бернулли – это случайный эксперимент с двумя возможными исходами: «успех» (с вероятностью p) и «неудача» (с вероятностью $1 - p$). Геометрическое распределение – это распределение вероятностей, лежащее в основе случайной величины X, представляющей количество испытаний Бернулли, необходимых для достижения первого успеха:

$$Pr(X = k) = (1 - p)^{k-1}p.$$

Процесс Пуассона – это вероятностный процесс с непрерывным временем, подсчитывающий количество событий в заданном временном интервале, если предположить, что события происходят независимо и с постоянной скоростью. Например, процесс Пуассона предлагает хорошую модель времени прибытия для пассажиров, прибывающих на большую железнодорожную станцию. Если предположить, что количество пассажиров, прибывающих за очень малый интервал времени ε, равно $\lambda \cdot \varepsilon$ (где λ – константа), то нас интересует вероятность $F(X)$ того, что никто не прибудет на станцию в течение интервала времени X. Экспоненциальное распределение описывает время между событиями в процессе Пуассона.

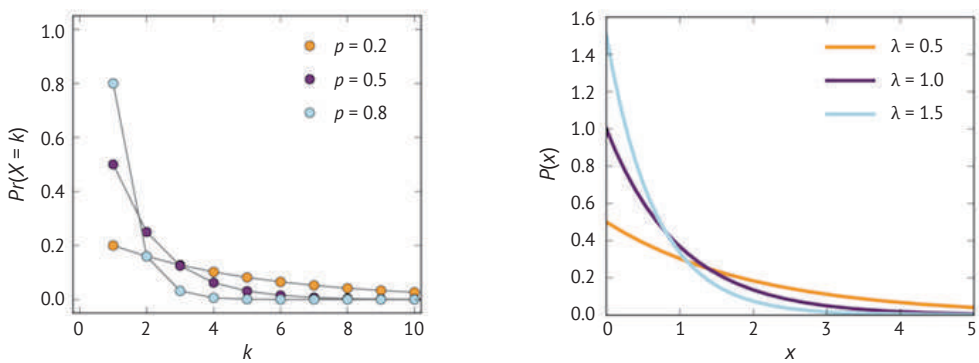


Рис. 6.34 Функции плотности вероятности геометрического (слева) и экспоненциального (справа) распределения для трех различных значений p и λ соответственно

ОСТАНОВИТЕСЬ и задумайтесь. Видите ли вы какое-либо сходство между процессом Пуассона и экспериментом Бернулли, или между экспоненциальным и геометрическим распределениями?

Экспоненциальное распределение является просто непрерывным аналогом геометрического распределения. Точнее, процесс Пуассона характеризуется **параметром скорости** λ таким, что количество событий k во временном интервале продолжительностью ε соответствует **распределению вероятности Пуассона.**:

Сортировка блинов Билла Гейтса и Дэвида Х. Коэна

Прежде чем биологи столкнулись с задачами рекомбинации генома, математики поставили **задачу сортировки блинов**, возникшую из следующей гипотетической головоломки официанта.

> *Шеф-повар у нас неаккуратный, и когда он готовит стопку блинов, они выходят разного размера. Поэтому, когда я доставляю их клиенту, по дороге к столу я их переставляю (чтобы самые маленькие оказались сверху и так далее, вплоть до самых больших внизу), хватая несколько сверху и переворачивая их снова, повторяя это (изменяя число, которое я переворачиваю) столько раз, сколько необходимо. Если есть n блинов, какое максимальное количество переворотов мне придется применить, чтобы переставить их в нужном порядке?*

Формально **реверсия префикса** – это реверсия, которое инвертирует префикс или начальный интервал перестановки. **Задача сортировки блинов** соответствует сортировке беззнаковых перестановок по реверсиям префикса. Например, показанная ниже серия перестановок префиксов игнорирует знаки и представляет собой сортировку **беззнаковой рекомбинации** (1 7 6 10 9 8 2 11 3 5 4) в тождественную беззнаковую перестановку (1 2 3 4 5 6 7 8 9 10 11):

```
( 1  7  6 10  9  8  2 11  3  5  4)
(11  2  8  9 10  6  7  1  3  5  4)
( 4  5  3  1  7  6 10  9  8  2 11)
(10  6  7  1  3  5  4  9  8  2 11)
( 2  8  9  4  5  3  1  7  6 10 11)
( 9  8  2  4  5  3  1  7  6 10 11)
( 6  7  1  3  5  4  2  8  9 10 11)
( 7  6  1  3  5  4  2  8  9 10 11)
( 2  4  5  3  1  6  7  8  9 10 11)
( 5  4  2  3  1  6  7  8  9 10 11)
( 1  3  2  4  5  6  7  8  9 10 11)
( 3  1  2  4  5  6  7  8  9 10 11)
( 2  1  3  4  5  6  7  8  9 10 11)
( 1  2  3  4  5  6  7  8  9 10 11)
```

Когда мы ищем кратчайшую серию перестановок префиксов, сортирующих перестановку со знаком, задача называется **задачей сортировки подгоревших блинов** (каждый блин «сожжен» с одной стороны, что дает ему две возможные ориентации).

ОСТАНОВИТЕСЬ и задумайтесь. Докажите, что любую беззнаковую перестановку длины n можно отсортировать, используя не более $2 \cdot (n - 1)$ перестановок префикса. Докажите, что любую знаковую перестановку длины n можно отсортировать, используя $3 \cdot (n - 1) + 1$ реверсий префикса.

В середине 1970-х Билл Гейтс, студент Гарвардского университета, и Христос Пападимитриу, профессор Гейтса, предприняли первую попытку решить задачу о переворачивании блинов и доказали, что любая перестановка длины n может быть отсортирована не более чем с $5/3 \cdot (n + 1)$ реверсий префикса, таким образом получив результат, который не был улучшен в течение трех десятилетий. Дэвид Х. Коэн работал над задачей переворачивания подгоревших блинов в Беркли, прежде чем он оставил компьютерные науки, чтобы стать сценаристом «Симпсонов» и в конечном итоге продюсером «Футурамы». Вместе с Мануэлем Блюмом он продемонстрировал, что задачу о переворачивании подгоревших блинов можно решить, используя не более $2 \cdot (n - 1)$ перестановок префикса.

Сортировка линейных перестановок по реверсиям

В основном тексте мы определили граф точек останова для кольцевых хромосом, но эту структуру можно легко распространить и на линейные хромосомы. На рис. 6.35 Х-хромосомы человека и мыши изображены в виде альтернативных красно-черных и сине-черных путей (первая и вторая схемы). Эти два пути накладываются на третью схему, чтобы сформировать граф точек останова, который имеет пять альтернативных красно-синих циклов.

ОСТАНОВИТЕСЬ и задумайтесь. Докажите следующий аналог теоремы о цикле для перестановок: «При заданных перестановках P и Q любое изменение, примененное к P, может увеличить $Cycles(P, Q)$ не более чем на 1».

В то время как количество тривиальных циклов равно $Blocks(Q, Q)$ в тривиальном графе точек останова круговой перестановки, тривиальный граф точек останова линейной перестановки имеет $Blocks(Q, Q) + 1$ тривиальный цикл. Поскольку теорема о цикле верна для линейных перестановок, возможно, реверсное расстояние $d_{rev}(P, Q)$ равно $Blocks(P, Q) + 1 - Cycles(P, Q)$ для линейных

хромосом? В конце концов, для Х-хромосом человека и мыши *Blocks*(*P, Q*) + 1 − *Cycles*(*P, Q*) равно 11 + 1 − 5 = 7, что, как мы уже знаем, является реверсным расстоянием между Х-хромосомами человека и мыши.

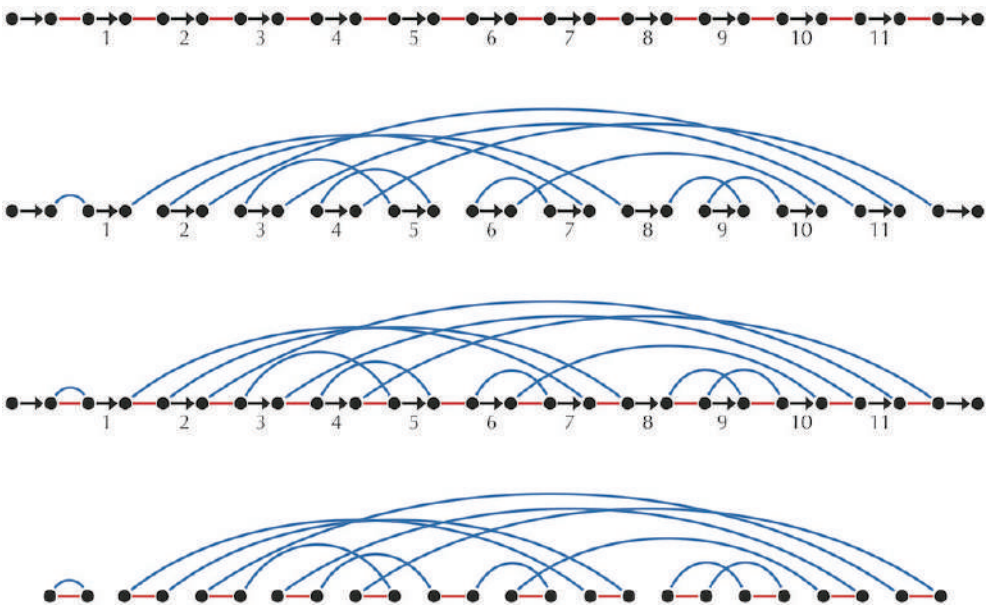


Рис. 6.35 (Схема 1) Альтернативный путь из красных и черных ребер, представляющих Х-хромосому человека (+1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11). (Схема 2) Альтернативный путь из синих и черных ребер, представляющий Х-хромосому мыши (+1 −7 +6 −10 +9 −8 +2 −11 −3 +5 +4). (Схема 3) Граф точек разрыва Х-хромосом мыши и человека получен путем наложения красно-черных и сине-черных путей из первых двух схем. (Схема 4) Чтобы выделить пять альтернативных красно-синих циклов на графе точек останова, черные ребра удалены

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы изменить доказательство теоремы о дистанции двойного разрыва, чтобы доказать, что $d_{rev}(P, Q) = Blocks(P, Q) + 1 - Cycles(P, Q)$ для линейных перестановок P и Q?

Вы можете проверить, что если $P = (+2\ +1)$ и $Q = (+1\ +2)$, то $d_{rev}(P, Q) \neq Blocks(P, Q) + 1 - Cycles(P, Q)$. Это делает маловероятным, что мы сможем разработать простой алгоритм для вычисления реверсного расстояния.

Однако нижняя граница $d_{rev}(P, Q) \geqslant Blocks(P, Q) + 1 - Cycles(P, Q)$ очень хорошо аппроксимирует реверсное расстояние между линейными перестановками.

Это интригующее представление поставило вопрос о том, близка ли эта граница к точной формуле. В 1999 году Ханненхалли и Певзнер нашли эту формулу, определив два специальных типа структур графа точек останова, называемых «барьерами» и «крепостями». Обозначив количество барьеров и крепостей в *BreakpointGraph*(*P, Q*) через *Hurdles*(*P, Q*) и *Fortresses*(*P, Q*) соответственно, они доказали, что реверсное расстояние определяется выражением:

$$d_{rev}(P, Q) = Blocks(P, Q) + 1 - Cycles(P, Q) + Hurdles(P, Q) + Fortresses(P, Q).$$

Используя эту формулу, они разработали полиномиальный алгоритм вычисления $d_{rev}(P, Q)$. Тем не менее *Hurdles*(*P, Q*) и *Fortresses*(*P, Q*) малы для подавляющего большинства перестановок, поэтому нижняя граница *Blocks*(*P, Q*) + 1 − *Cycles*(*P, Q*) является хорошей аппроксимацией реверсного расстояния на практике.

Библиографические примечания

Альфред Стёртевант был первым, кто обнаружил рекомбинации при сравнении порядка генов у плодовых мушек (Sturtevant, 1921[1]). Вместе с Феодосием Добжанским Стёртевант стал пионером в анализе геномных перестановок, опубликовав важную статью, в которой был представлен сценарий рекомбинаций для многих видов плодовых мушек (Sturtevant and Dobzhansky, 1936[2]). Модель случайных разрывов была предложена Ohno, 1973[3], развита в дальнейшем Nadeau and Taylor, 1984[4], и опровергнута Pevzner and Tesler, 2003b[5].

Понятие графа точек останова было предложено Bafna and Pevzner, 1996[6]. Полиномиальный алгоритм сортировки по реверсиям был разработан Hannenhalli and Pevzner, 1999[7]. Алгоритм построения синтенных блоков, представленный в этой главе, был описан Pevzner and Tesler, 2003a[8]. Операция двойного разрыва (2-Break) была введена Yancopoulos, Attie, and Friedberg, 2005[9], под названием «двойной разрез и соединение».

Первый алгоритмический анализ «задачи о переворачивании блинов» был описан Gates and Papadimitriou, 1979[10]. Первый алгоритмический анализ задачи о переворачивании блинов был описан Cohen and Blum, 1995[11].

---

1 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1084859/.
2 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1076803/.
3 https://www.nature.com/articles/244259a0.
4 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC344928/.
5 https://www.pnas.org/content/100/13/7672.
6 https://dl.acm.org/doi/10.1137/S0097539793250627.
7 https://dl.acm.org/doi/10.1145/300515.300516.
8 https://www.ncbi.nlm.nih.gov/pmc/articles/PMC430962/.
9 https://www.ncbi.nlm.nih.gov/pubmed/15951307.
10 https://www.sciencedirect.com/science/article/pii/0012365X79900682.
11 https://www.sciencedirect.com/science/article/pii/0166218X94000093.

Задача множественной перестановки генома была рассмотрена Ma et al., 2008[1], а также Alekseyev and Pevzner, 2009[2]. Zhao and Bourque, 2009[3], заметили, что совпадающие дупликации могут запускать рекомбинацию генома.

[1] https://www.pnas.org/content/105/38/14254.
[2] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2675983/.
[3] https://genome.cshlp.org/content/19/5/934.full.html.

Глава 7

Какое животное заразило нас коронавирусом?

Эволюционная реконструкция

Самая быстрая вспышка

Проблемы в отеле «Метрополь»

21 февраля 2003 года китайский врач по имени Лю Цзяньлунь прилетел в Гонконг на свадьбу и поселился в номере 911 отеля «Метрополь». На следующий день он почувствовал себя слишком больным, чтобы присутствовать на свадьбе, и был госпитализирован. Две недели спустя доктор Лю умер.

На смертном одре Лю сказал врачам, что недавно лечил больных в провинции Гуандун, Китай, где смертельное, очень заразное респираторное заболевание заразило сотни людей. Китайское правительство кратко упомянуло об этом инциденте во Всемирной организации здравоохранения, но пришло к выводу, что вероятным виновником была обычная бактериальная инфекция. К тому времени, когда кто-то осознал серьезность болезни, было уже слишком поздно, чтобы остановить вспышку. 23 февраля мужчина, который останавливался напротив доктора Лю в отеле «Метрополь», отправился в Ханой и умер, заразив 80 человек. 26 февраля женщина выписалась из «Метрополя», вернулась в Торонто и умерла после того, как там началась вспышка. 1 марта третий гость был госпитализирован в больницу Сингапура, где в течение двух недель возникло еще 16 случаев заболевания.

Учтите, что «черной смерти», унесшей жизни более трети всех европейцев в XIV веке, понадобилось четыре года, чтобы добраться из Константинополя в Киев. Или то, что ВИЧ понадобилось два десятилетия, чтобы обогнуть земной шар. Напротив, эта загадочная новая болезнь пересекла Тихий океан в течение недели после проникновения в Гонконг.

Пока чиновники здравоохранения готовились к последствиям самой быстро распространяющейся пандемии в истории человечества, началась паника. Предприятия были закрыты, больных пассажиров сняли с самолетов, а китайские чиновники пригрозили казнить инфицированных пациентов, нарушивших карантин.

Международные поездки, возможно, способствовали быстрому распространению болезни, но международное сотрудничество в конечном итоге сдержало ее. За несколько недель биологи идентифицировали вирус, вызвавший эпидемию, и секвенировали его геном. В процессе борьбы с загадочной новой болезнью она получила название: **«тяжелый острый респираторный синдром»**, или **SARS** («атипичная пневмония»).

Эволюция SARS

Вирус, вызывающий атипичную пневмонию, принадлежит к семейству вирусов, называемых коронавирусами, которые названы в честь латинского слова *corona*, потому что вирусная частица напоминает солнечную корону (рисунок справа). Коронавирусы поражают дыхательные пути млекопитающих и птиц и обычно вызывают лишь незначительные проблемы, такие как

простуда. До атипичной пневмонии никто не верил, что коронавирус может нанести такой ущерб.

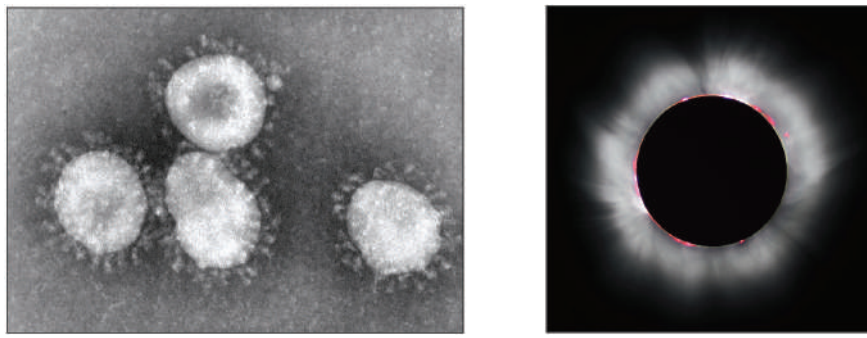


Рис. 7.1 (Слева) Частицы коронавируса.
(Справа) Солнечное затмение с видимой солнечной короной

Коронавирусы, вирусы гриппа и HIV (ВИЧ) являются **РНК-вирусами**, что означает, что они содержат РНК вместо ДНК. Репликация РНК имеет более высокую частоту ошибок, чем репликация ДНК, поэтому РНК-вирусы способны быстрее мутировать в новые штаммы. Быстрая мутация РНК-вирусов объясняет, почему вакцины от гриппа меняются каждый год и почему существует множество различных подтипов ВИЧ.

Исследователи SARS первоначально выдвинули гипотезу, что, подобно ВИЧ и гриппу, **коронавирус SARS** (сокращенно **SARS-CoV**) перешел от животных к людям. Сначала они назвали птиц вероятными подозреваемыми из-за сходства между вспышкой атипичной пневмонии и птичьим гриппом, формой гриппа, возникающей у кур, которая редко передается людям, и она даже более смертоносна, чем атипичная пневмония, убивая более половины людей, которых заражает. Тем не менее, когда в апреле 2003 года исследователи секвенировали геном SARS-CoV длиной 29 751 нуклеотид, стало очевидно, что SARS не произошел от птиц, потому что его геном не был похож на птичьи коронавирусы.

К осени 2003 года исследователи секвенировали множество штаммов SARS-CoV у пациентов из разных стран, но многие вопросы так и остались без ответа. Как SARS-CoV преодолел видовой барьер на пути к человеку? Когда и где это произошло? Как SARS распространился по миру и кто кого заразил?

Каждый из этих вопросов о атипичной пневмонии в конечном итоге связан с задачей построения **эволюционного дерева** (также известного как **филогения**). В качестве другого примера, построив эволюционное древо вирусов приматов, связанных с HIV (рисунок ниже), ученые пришли к выводу, что ВИЧ передался человеку в пяти отдельных случаях (подробнее см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Когда ВИЧ перешел от приматов к человеку?**). Но какой алгоритм использовали исследователи для построения этой филогении?

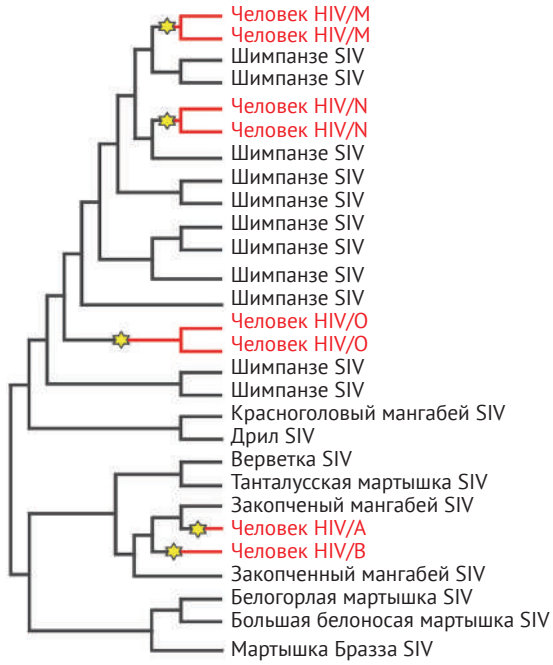


Рис. 7.2 ВИЧ включает в себя пять различных вирусных семейств, обозначенных как A, B, M, N и O, при этом семейство M ответственно за 95 % всех ВИЧ-инфекций. Пять семейств являются разными ответвлениями эволюционного древа вируса иммунодефицита обезьян (SIV), который поражает приматов. Звездочки обозначают переход вирусов от приматов к человеку. Семейства A и B произошли от закопченных мангабеев, тогда как семейства M, N и O произошли от шимпанзе

Преобразование матриц расстояний в эволюционные деревья

Построение матрицы расстояний из геномов коронавируса

Чтобы определить, как SARS перешел от животных к человеку, ученые начали секвенировать коронавирусы разных видов, чтобы определить, какой из них больше всего похож на SARS-CoV. Однако создание множественного выравнивания полных вирусных геномов сложно, потому что вирусные гены часто рекомбинируют, подвергаются вставкам и делециям. По этой причине ученые сосредоточились только на одном из шести генов SARS-CoV. Этот ген кодирует спайковый гликопротеин, известный как **спайковый белок**, который идентифицирует рецепторы и связывается с ними на клеточной мембране хозяина.

В SARS-CoV спайковый белок имеет длину 1255 аминокислот и довольно слабое сходство со спайковыми белками других коронавирусов. Однако даже

этих тонких сходств оказалось достаточно для построения множественного выравнивания спайковых белков у различных коронавирусов.

После построения множественного выравнивания генов из n разных видов биологи часто преобразуют это выравнивание в **матрицу расстояний** D размерностью nxn. Во многих случаях $D_{i,j}$ представляет собой количество различающихся символов между генами, представляющими строки i и j выравнивания (рисунок ниже). Однако матрицы расстояний могут быть построены с использованием множества различных функций расстояний, чтобы соответствовать различным приложениям. Например, $D_{i,j}$ также может представлять расстояние редактирования между генами i-го и j-го видов. Или матрица расстояний для n геномов может быть построена из расстояний с двумя разрывами между каждой парой геномов.

Вид	Выравнивание	Матрица состояний			
		Шимпанзе	**Человек**	**Тюлень**	**Кит**
Шимпанзе	ACGTAGGCCT	0	3	6	4
Человек	ATGTAAGACT	3	0	7	5
Тюлень	TCGAGAGCAC	6	7	0	2
Кит	TCGAAAGCAT	4	5	2	0

Рис. 7.3 Множественное выравнивание гипотетических последовательностей ДНК четырех видов вместе с матрицей расстояний, полученной путем подсчета количества различающихся символов между каждой парой строк в этом множественном выравнивании

Независимо от того, какую функцию расстояния мы используем, чтобы D была матрицей расстояний, она должна удовлетворять трем свойствам. Она должна быть **симметричной** (для всех i и j $D_{i,j} = D_{j,i}$), **неотрицательной** (для всех i и j $D_{i,j} \geqslant 0$) и удовлетворять **неравенству треугольника** (для всех i, j и k, $D_{i,j} + D_{j,k} \geqslant D_{i,k}$).

Упражнение. Докажите, что если $D_{i,j}$ равно количеству различающихся символов между строками i и j в множественном выравнивании, то D является симметричной, неотрицательной и удовлетворяет неравенству треугольника.

К концу 2003 года биоинформатики секвенировали множество коронавирусов, взятых у различных животных и пациентов с атипичной пневмонией, а затем вычислили соответствующую матрицу расстояний. Им нужно было использовать эту информацию, чтобы построить филогению коронавируса и выяснить происхождение и распространение эпидемии атипичной пневмонии.

Эволюционные деревья в виде графов

Возможно, вы заметили, что дерево ВИЧ, которое мы представили ранее, имеет структуру графа. Кроме того, на рисунке ниже показано представление филогении всей жизни в виде графа.

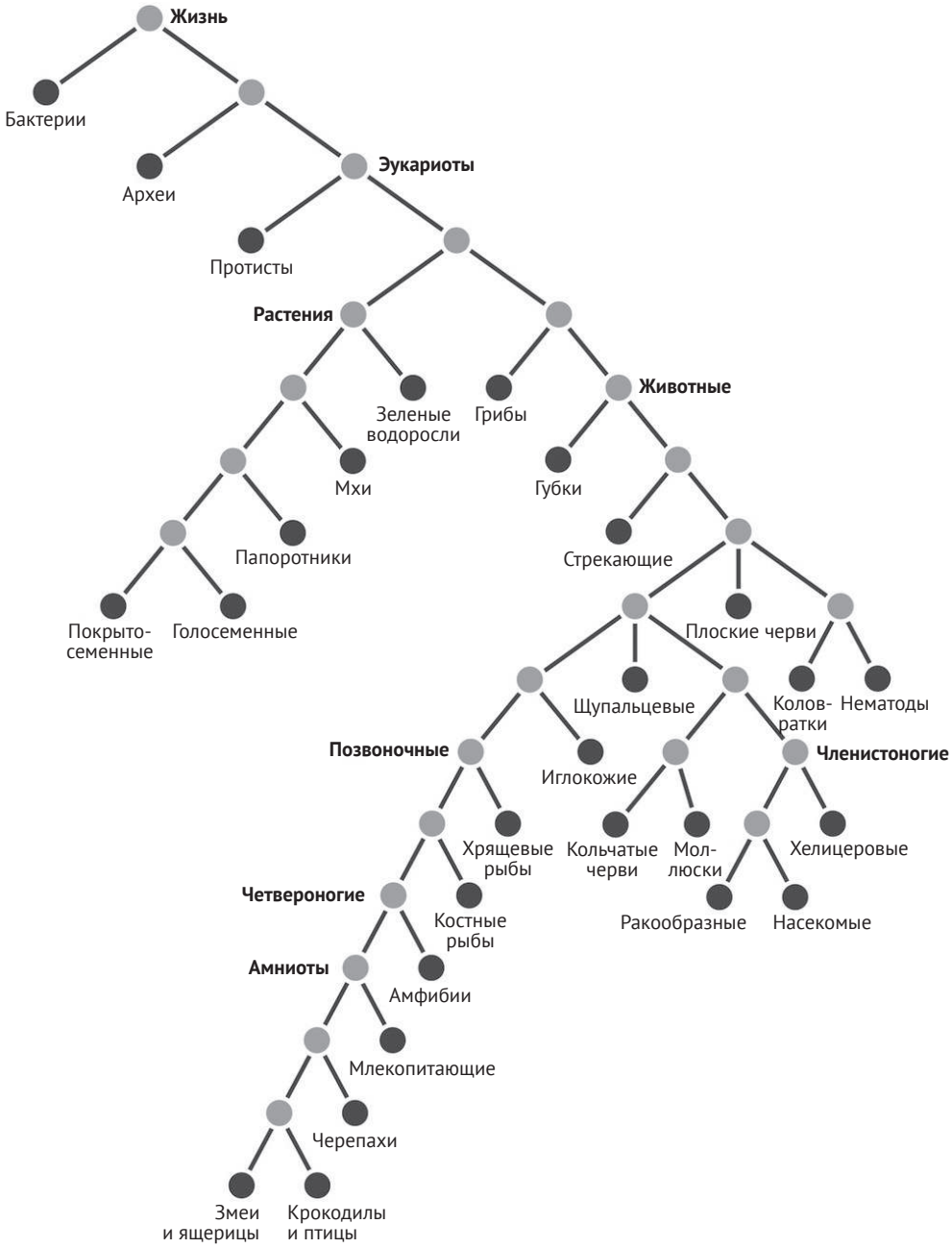


Рис. 7.4 Связный граф без циклов, моделирующий эволюционное дерево жизни на Земле. Современные виды показаны более темными узлами

Графы, используемые для моделирования филогений, имеют два общих свойства. Они связные (т. е. в любой узел можно попасть из любого другого узла) и не содержат циклов. По этой причине мы определим **дерево** как связный граф без циклов.

На рисунке ниже представлено нескольких дополнительных примеров; более темные узлы – это **листья**, или узлы, имеющие степень 1 (степень узла – это количество ребер, соединенных с этим узлом). Узлы большей степени называются **внутренними узлами**.

Упражнение. Докажите следующие утверждения:

- каждое дерево, имеющее не менее двух узлов, имеет не менее двух листьев;
- каждое дерево с n узлами имеет $n - 1$ ребер;
- существует ровно один путь, соединяющий каждую пару узлов дерева. Подсказка: что произойдет, если есть два разных пути, соединяющих пару узлов? Что произошло бы, если бы не было путей, соединяющих пару узлов?

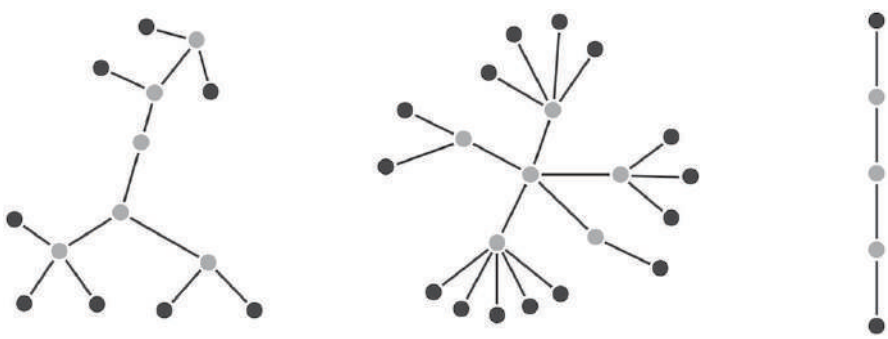


Рис. 7.5 Деревья бывают самых разных форм. В каждом из трех показанных деревьев листья нарисованы темнее, чем внутренние узлы

Еще раз взгляните на форму дерева жизни, показанную выше. Вы увидите, что современные виды были отнесены к листьям этого дерева. Внутренние узлы представляют из себя неизвестные виды предков.

Для данного листа j существует только один узел, соединенный с j ребром, который мы называем **родителем** j, обозначая *Parent*(j). Ребро, соединяющее лист с его родителем, называется **веткой**.

В **корневом дереве** один узел обозначается как специальный узел, называемый **корнем**, и ребра в дереве автоматически наследуют неявную ориентацию от корня, который размещается вверху или слева от дерева (рисунок ниже). Эта ориентация ребра моделирует время: предок всех видов в дереве находится в корне, и эволюция идет от корня наружу через дерево. Деревья без обозначенного корня называются **некорневыми**.

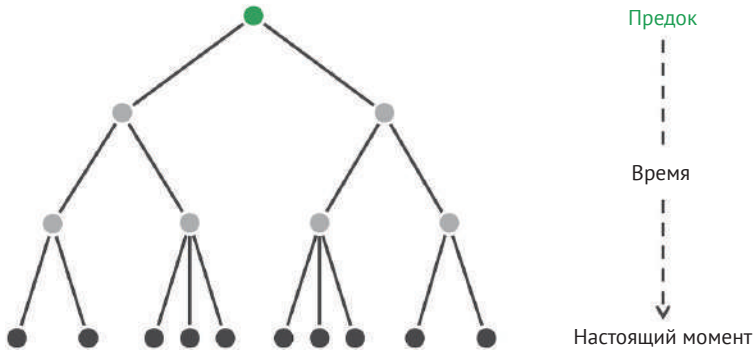


Рис. 7.6 Корневое дерево; корень (представляющий предка всех видов в дереве) указан зеленым цветом в верхней части дерева. Наличие корня подразумевает ориентацию ребер в дереве – от корня

ОСТАНОВИТЕСЬ и задумайтесь. Где бы вы поместили корень в филогении HIV, с которой мы сталкивались ранее (рис. 7.2)?

В этой главе мы будем анализировать корневые деревья, когда попытаемся вывести узел, соответствующий предку всех видов в эволюционном дереве; в противном случае будем анализировать некорневые деревья. На рис. 7.7 показано некорневое дерево вирусов ВИЧ, созданное из набора данных, отличного от того, который использовался для создания рисунка, представленного выше. Предлагая два дополнительных подтипа ВИЧ, он показывает, что классификация HIV по пяти семействам не высечена на камне.

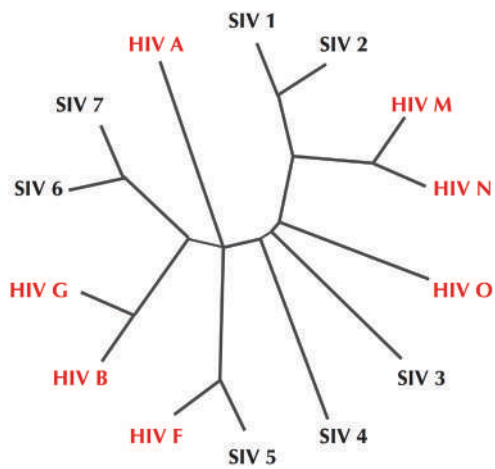


Рис. 7.7 Некорневое дерево вирусов HIV и SIV, которое предлагает дополнительные семейства вирусов F и G в дополнение к семействам вирусов A, B, M, N и O, показанным на рисунке выше

Построение филогении по расстояниям

Сначала мы сосредоточимся на получении некорневого дерева из матрицы расстояний. Листья этого дерева должны соответствовать видам, представленным матрицей (с внутренними узлами, соответствующими неизвестным видам предков). Чтобы отразить эволюционное расстояние между видами в дереве, мы присваиваем каждому ребру неотрицательную длину, представляющую расстояние между организмами, соединенными ребром, как показано на рисунке ниже.

Вид	Выравнивание	Матрица состояний			
		Шимпанзе	**Человек**	**Тюлень**	**Кит**
Шимпанзе	ACGTAGGCCT	0	3	6	4
Человек	ATGTAAGACT	3	0	7	5
Тюлень	TCGAGAGCAC	6	7	0	2
Кит	TCGAAAGCAT	4	5	2	0

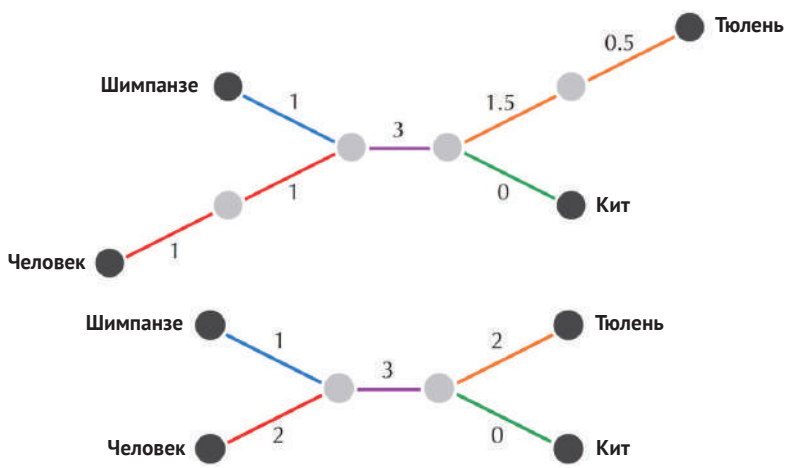


Рис. 7.8 Матрица расстояний, построенная из множественного выравнивания, а также два некорневых дерева, соответствующих этой матрице расстояний

В этой главе мы определяем длину пути в дереве как сумму длин его ребер (а не как количество ребер на пути). В результате эволюционное расстояние между двумя современными видами, соответствующими листьям i и j дерева T, равно длине уникального пути, соединяющего i и j, обозначаемого $d_{i,j}(T)$.

Задача расстояния между листьями: *вычислить расстояния между листьями взвешенного дерева.*

Input: целое число n, за которым следует список смежности взвешенного дерева с n листьями.

Output: матрица $(d_{i,j})$ размерностью nxn, где $d_{i,j}$ – длина пути между листьями i и j.

☞ **Загрузить данные 7.1**

Задачу о расстоянии между листьями решить просто, но мы хотели бы решить обратную задачу, в которой нужно построить некорневое дерево, моделирующее заданную матрицу расстояний. Мы говорим, что взвешенное некорневое дерево T **соответствует** матрице расстояний D, если $d_{i,j}(T) = D_{i,j}$ для каждой пары листьев i и j (на рис. 7.9 показаны два дерева, соответствующие матрице расстояний, с которой мы уже сталкивались ранее).

Вид	Выравнивание	Матрица состояний			
		Шимпанзе	**Человек**	**Тюлень**	**Кит**
Шимпанзе	ACGTAGGCCT	0	3	6	4
Человек	ATGTAAGACT	3	0	7	5
Тюлень	TCGAGAGCAC	6	7	0	2
Кит	TCGAAAGCAT	4	5	2	0

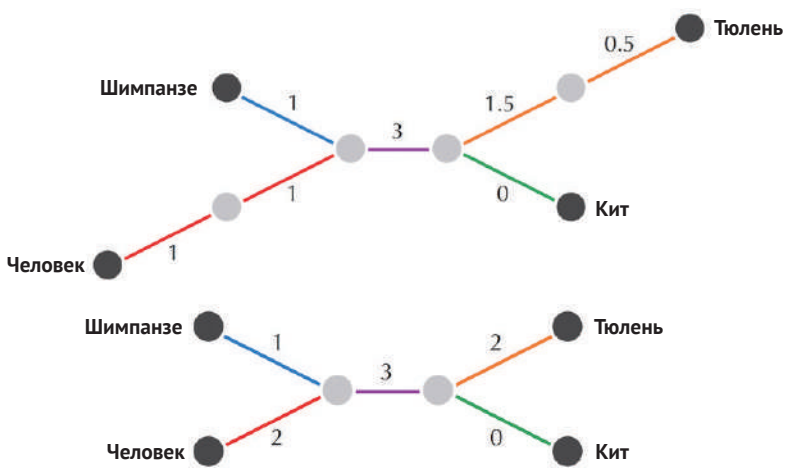


Рис. 7.9 Матрица расстояний и два эволюционных дерева, ей соответствующих

Задача построения филогении по расстояниям:
реконструировать эволюционное дерево, соответствующее матрице расстояний.

Input: матрица расстояний.

Output: дерево, соответствующее этой матрице расстояний.

ОСТАНОВИТЕСЬ и задумайтесь. Всегда ли есть решение задачи построения филогении, основанной на расстояниях?

Не всякая матрица расстояний имеет соответствующее ей дерево. Дополнительные сведения см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Поиск дерева, соответствующего матрице расстояний**.

Поэтому мы называем **аддитивной** матрицу расстояний, если существует дерево, которое соответствует этой матрице, и **неаддитивной** в противном случае (пример неаддитивной матрицы 4×4 показан ниже).

	v_1	v_2	v_3	v_4
v_1	0	3	4	3
v_2	3	0	4	5
v_3	4	4	0	2
v_4	3	5	2	0

Рис. 7.10 Пример неаддитивной матрицы

Термин «аддитивный» используется потому, что длины всех ребер на пути между листьями i и j в дереве, соответствующем матрице D, складываются с $D_{i,j}$.

Оба приведенных выше дерева соответствуют матрице расстояний, поэтому было бы неплохо иметь представление о «каноническом» дереве, соответствующем матрице расстояний. Распространяя определения, введенные в предыдущей главе, на неориентированные графы, мы говорим, что путь в дереве **неветвящийся**, если каждый узел, кроме начального и конечного узлов пути, имеет степень, равную 2. Неветвящийся путь называется **максимальным**, если это не подпуть еще более длинного неветвящегося пути. Если мы заменим каждый максимальный неветвящийся путь одним ребром, длина которого равна длине пути, то дерево в средней части рисунка станет деревом внизу (рис. 7.9). В общем случае после такого преобразования узлов степени 2 не остается; дерево, удовлетворяющее этому свойству, называется **простым деревом**.

Оказывается, если матрица аддитивна, то существует единственное простое дерево, соответствующее этой матрице. Поэтому в задаче филогении, основанной на расстоянии, мы будем использовать термин «*Tree*(D)» для обозначения простого дерева, соответствующего аддитивной матрице расстояний D. Таким образом, наш вопрос состоит в том, как построить *Tree*(D) из D.

Упражнение. Докажите, что каждое простое дерево с n листьями имеет не более $n - 2$ внутренних узлов.

На пути к алгоритму построения филогении по расстоянию

В поисках соседних листьев

Естественным первым шагом для решения проблемы филогении по расстоянию было бы обеспечение того, чтобы два ближайших вида по отношению к матрице расстояний D соответствовали соседям в дереве (D). Другими словами, минимальное значение $D_{i,j}$ должно соответствовать листьям i и j, имеющим одного родителя. В оставшейся части этой главы, когда мы говорим о минимальном элементе матрицы, мы имеем в виду минимальный **недиагональный** элемент, т. е. значение $D_{i,j}$ такое, что $i \neq j$.

Теорема. Каждое простое дерево, имеющее не менее четырех узлов, имеет пару соседних листьев.

Доказательство. Для заданного простого дерева T не менее чем с четырьмя узлами рассмотрим путь $P = (v_1, \ldots, v_k)$, который имеет максимальное число узлов среди всех путей в T. Поскольку T связное, k должно быть не менее 3. Кроме того, узлы v_1 и v_k должны быть листьями, так как в противном случае мы могли бы расширить путь P до более длинного пути. Поскольку T простое, каждый внутренний узел T имеет степень не менее 3. Таким образом, узел v_2, который является родителем v_1, должен иметь по крайней мере три смежных узла: v_1, v_3 и еще один узел ω.

Мы утверждаем, что ω – лист, из чего следует, что листья v_1 и ω – соседи. Предположим противное: если бы ω не был листом, то в силу простоты T узел ω был бы смежным с другим узлом u. В результате мы могли бы образовать путь $P' = (u, \omega, v_2, v_3, \ldots, v_k)$, содержащий $k + 1$ узлов и противоречащий нашему исходному предположению, что P имеет максимальное число узлов. Таким образом, ω должен быть листом, что означает, что v_1 и ω являются соседями, что и требовалось доказать. ■

На рис. 7.11 показано, что для соседних листьев i и j, имеющих общий родительский узел m, для каждого другого листа k в дереве выполняется следующее равенство:

$$d_{k,m} = \frac{(d_{i,m} + d_{k,m}) + (d_{j,m} + d_{k,m}) - (d_{i,m} + d_{j,m})}{2} = \frac{d_{i,k} + d_{j,k} - d_{i,j}}{2}.$$

Поскольку i, j и k – листья, мы можем вычислить расстояние $d_{k,m}$ между узлами k и m через элементы аддитивной матрицы расстояний D:

$$d_{k,m} = \frac{(D_{i,k} + D_{j,k} - D_{i,j})}{2}.$$

В случае, когда родитель m имеет степень 3 (как на рисунке выше), удаление листьев i и j из дерева превращает m в лист и, таким образом, уменьшает общее

количество листьев (рис. 7.12). Эта операция эквивалентна удалению строк i и j, а также столбцов i и j из D с последующим добавлением новой строки и столбца, соответствующих их родительскому m, где расстояния от m до других листьев вычисляются в соответствии с приведенной выше формулой.

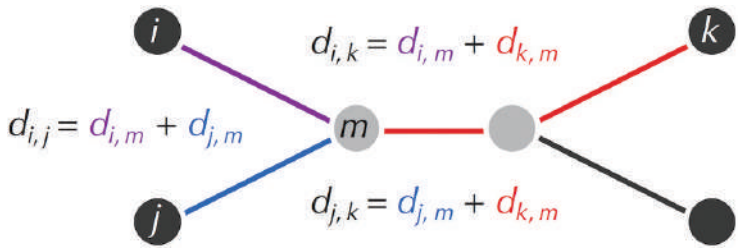


Рис. 7.11 Для соседних листьев i и j и их родительского узла m $d_{k,m} = \frac{d_{i,k} - d_{j,k} - d_{i,j}}{2}$ для каждого другого листа k в дереве

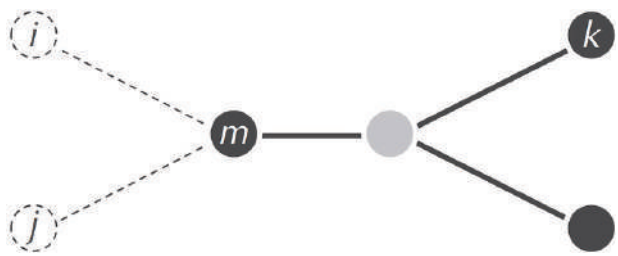


Рис. 7.12 Удаление листьев i и j из дерева превращает m в лист (мы предполагаем, что m имеет степень 3). Расстояния от этого нового листа до любого другого листа k можно пересчитать как $d_{k,m} = \frac{(D_{i,k} + D_{j,k} - D_{i,j})}{2}$

Упражнение. Мы только что описали, как уменьшить размер дерева, а также размерность матрицы расстояний D, если родительский узел (m) имеет степень 3. Разработайте аналогичный метод в случае, когда степень m больше, чем 3.

Это рассуждение подразумевает рекурсивный алгоритм для задачи построения филогении по расстояниям:

- найдите пару соседних листьев i и j, выбрав минимальный элемент $D_{i,j}$ в матрице расстояний;
- замените i и j их родителем и пересчитайте расстояния от этого родителя до всех остальных листьев, как описано выше;
- решите задачу филогении по расстояниям для меньшего дерева;
- добавьте ранее удаленные листья i и j обратно в дерево.

Упражнение. Примените этот рекурсивный метод к аддитивной матрице расстояний, показанной на рисунке ниже. (Решите это упражнение вручную.)

	v_1	v_2	v_3	v_4
v_1	0	13	21	22
v_2	13	0	12	13
v_3	21	12	0	13
v_4	22	13	13	0

Рис. 7.13 Аддитивная матрица расстояний

Вычисление длины ветвей

Если вы попытались выполнить предыдущее упражнение, то вы, вероятно, сошли с ума. Причина в том, что на первом шаге предложенного нами алгоритма мы предполагали, что минимальный элемент аддитивной матрицы расстояний соответствует соседним листьям. Однако, как показано на рисунке ниже, это предположение не обязательно верно! Таким образом, нам нужен новый подход к задаче построения филогении по расстоянию, поскольку обнаружение коронавируса животных, находящегося на наименьшем расстоянии от SARS-CoV, может быть не лучшим способом определения животного резервуара SARS.

	v_1	v_2	v_3	v_4
v_1	0	13	21	22
v_2	13	0	12	13
v_3	21	12	0	13
v_4	22	13	13	0

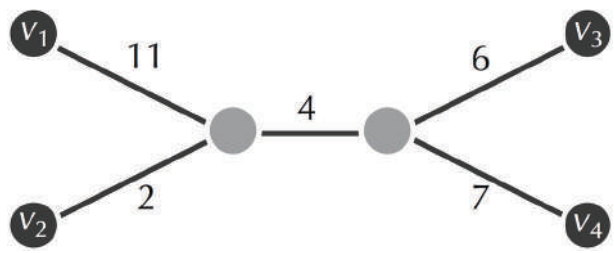


Рис. 7.14 Матрица расстояний вместе с простым деревом, соответствующим этой матрице расстояний. Два ближайших листа этого дерева (*j* и *k*) не являются соседями

Предложенный нами рекурсивный метод, возможно, потерпел неудачу, но использование рекурсии было хорошей идеей, поэтому мы рассмотрим другой рекурсивный алгоритм. Вместо того чтобы искать *пару* соседей в *Tree*(*D*), мы

уменьшим размер дерева, обрезав его листья по одному. Конечно, мы не знаем *Tree*(*D*), поэтому нужно каким-то образом обрезать листья в *Tree(D)*, анализируя матрицу расстояний.

В качестве первого шага к построению *Tree*(*D*) мы решим более скромную задачу вычисления длин ветвей *Tree*(*D*). Итак, для данного листа *j* в дереве мы обозначаем длину ветви, соединяющей *j* с его родителем, как *LimbLength*(*j*). Ребра, не являющиеся ветвями, должны соединять два внутренних узла и поэтому называются **внутренними ребрами**.

Задача длины ветви: *вычислить длину ветви в дереве, определяемом аддитивной матрицей расстояний.*

Input: аддитивная матрица расстояний *D* и целое число *j*.

Output: *LimbLength*(*j*), длина ветви, соединяющей лист *j* с его родителем в *Tree*(*D*).

Чтобы вычислить *LimbLength*(*j*) для заданного листа *j*, обратите внимание, что, поскольку *Tree*(*D*) простое, мы знаем, что *Parent*(*j*) имеет степень не менее 3 (если только *Tree(D)* не имеет только двух узлов). Таким образом, мы можем думать о *Parent*(*j*) как о делении других узлов *Tree(D)* как минимум на три **поддерева** или меньшие деревья, которые остались бы, если бы мы удалили *Parent*(*j*) вместе с любыми ребрами, соединяющими его с другими узлами (рис. 7.15). Поскольку *j* – лист, он должен сам принадлежать поддереву; мы называем это поддерево T_j. Это приводит нас к следующему результату.

Теорема о длине ветви. Для аддитивной матрицы *D* и листа *j* длина ветви (*j*) равна минимальному значению $(D_{i,j} + D_{j,k} - D_{i,k})/2$ по всем листам *i* и *k*.

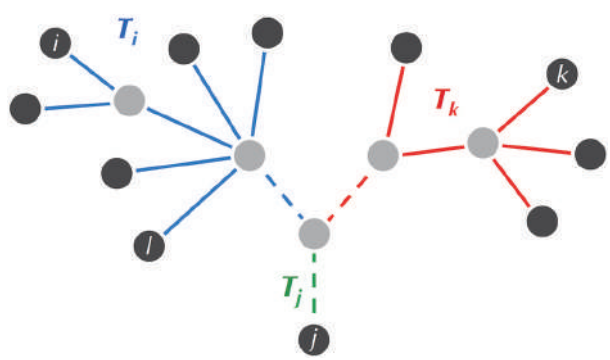


Рис. 7.15 Простое дерево с выбранными листьями *i*, *j*, *k* и *l*. Удаление родителя *j* (вместе с тремя пунктирными ребрами, соединяющими его с другими узлами) разделило бы это дерево на три поддерева, ребра которых показаны разными цветами. Листья *i* и *l* принадлежат T_i, тогда как лист *k* принадлежит T_k. Лист *j* принадлежит T_j, который содержит один узел

Доказательство теоремы о длине ветвей. Заданная пара листьев может принадлежать одному и тому же поддереву или разным поддеревьям на рис. 7.15. Итак, сначала предположим, что листья i и k принадлежат разным поддеревьям T_i и T_k. Поскольку $Parent(j)$ находится на пути, соединяющем i с k, отсюда следует, что

$$d_{i,j} = d_{i,Parent(j)} + LimbLength(j);$$
$$d_{j,k} = d_{k,Parent(j)} + LimbLength(j).$$

Сложение этих двух уравнений дает

$$d_{i,j} + d_{j,k} = d_{i,Parent(j)} + d_{k,Parent(j)} + 2 \cdot LimbLength(j).$$

Поскольку $d_{i,Parent(j)} + d_{k,Parent(j)}$ равно $d_{i,k}$, получаем

$$LimbLength(j) = \frac{d_{i,j} + d_{j,k} - d_{i,k}}{2} = \frac{D_{i,j} + D_{j,k} - D_{i,k}}{2}.$$

С другой стороны, предположим, что листья i и l принадлежат одному и тому же поддереву (рис. 7.15). Тогда путь из i в l не проходит через $Parent(j)$, поэтому справедливо неравенство

$$d_{i,Parent(j)} + d_{l,Parent(j)} \geqslant d_{i,l}.$$

Объединив это с уравнением

$$d_{i,j} + d_{j,l} = d_{i,Parent(j)} + d_{l,Parent(j)} + 2 \cdot LimbLength(j),$$

получим, что

$$LimbLength(j) = \frac{d_{i,j} + d_{j,l} - (d_{i,Parent(j)} + d_{l,Parent(j)})}{2} \leq \frac{d_{i,j} + d_{j,l} - d_{i,l}}{2} = \frac{D_{i,j} + D_{j,l} - D_{i,l}}{2}.$$

В результате этого обсуждения $LimbLength(j)$ должно быть меньше или равно $(D_{i,j} + D_{j,k} - D_{i,k})/2$ для любого выбора листьев i и k. Поскольку мы всегда можем найти листья i и k, принадлежащие разным поддеревьям (почему?), отсюда следует, что длина ветви $LimbLength(j)$ равна минимальному значению $(D_{i,j} + D_{j,k} - D_{i,k})/2$ по всем вариантам i и k, что и требовалось доказать. ■

Теперь у нас есть алгоритм решения проблемы длины ветви. Для каждого j мы можем вычислить $LimbLength(j)$, найдя минимальное значение $(D_{i,j} + D_{j,k} - D_{i,k})/2$ по всем парам листьев i и k.

Упражнение. Алгоритм, предложенный ранее, вычисляет $LimbLength(j)$ за время $O(n^2)$ (для матрицы расстояний nxn). Разработайте алгоритм, который вычисляет $LimbLength(j)$ за время $O(n)$.

Аддитивная филогения

Обрезка дерева

Поскольку теперь мы знаем, как найти длину любой ветви в *Tree*(*D*), можно рекурсивно построить *Tree*(*D*), используя алгоритм, показанный на рис. 7.16 и описанный далее.

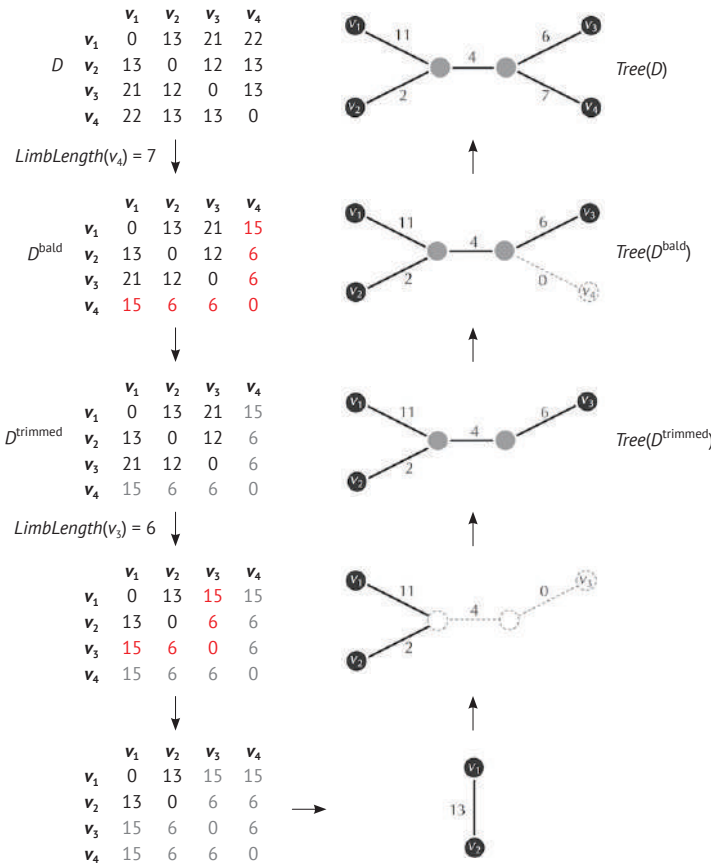


Рис. 7.16 Преобразование аддитивной матрицы расстояний в простое дерево, соответствующее этой матрице. В левой части сначала мы вычисляем *LimbLength*(v_4) = 7, а затем вычитаем 7 из недиагональных элементов в последней строке и столбце *D*, чтобы получить D^{bald} (обновленные значения показаны красным). Удаление этой строки и столбца дает матрицу расстояний $D^{trimmed}$ размерностью 3×3. Мы находим, что *LimbLength*(v_3) = 6 в $D^{trimmed}$, и вычитаем 6 из недиагональных элементов в третьей строке и столбце. Выделение этой строки и столбца серым цветом дает матрицу расстояний 2×2. С правой стороны мы можем подогнать эту матрицу расстояний 2×2 к дереву, состоящему из одного ребра. Находя точки крепления удаленных ветвей (показаны слева), мы реконструируем *Tree*($D^{trimmed}$), *Tree*(D^{bald}), а затем *Tree*(*D*).

Во-первых, представьте, что мы уже знаем *Tree*(*D*) и выбираем произвольный лист *j*. Обрежем ветвь *j*, уменьшив ее длину на *LimbLength*(*j*). Поскольку мы не знаем *Tree*(*D*), нам нужно представить обрезку листа *j* в терминах матрицы расстояний *D*. Для этого сначала вычтем *LimbLength*(*j*) из каждого недиагонального элемента в строке *j* и столбце *j* матрицы *D*, чтобы получить матрицу D^{bald}, для которой ветвь *j* стала **пустой ветвью** или ветвью длины 0 (рис. 7.16). Далее будем считать, что с дерева полностью исчезла пустая ветка. С точки зрения матрицы расстояний игнорирование пустой ветви означает удаление строки *j* и столбца *j* из *D* для получения меньшей матрицы расстояний D^{trimmed} размерностью $(n - 1)\times(n - 1)$.

Теперь мы можем за четыре шага рекурсивно найти *Tree*(*D*):

- выбираем произвольный лист *j*, вычисляем *LimbLength*(*j*) и строим матрицу расстояний D^{trimmed};
- решаем задачу филогении по расстояниям для D^{trimmed};
- определяем точку в *Tree*(D^{trimmed}), где лист *j* должен быть присоединен к *Tree*(D);
- добавляем ветку длины *LimbLength*(*j*), растущую из этой точки крепления в *Tree*(D^{trimmed}), чтобы сформировать *Tree*(*D*).

ОСТАНОВИТЕСЬ и задумайтесь. При добавлении листа *j* обратно в дерево (D^{trimmed}) как бы вы нашли его точку присоединения?

Прикрепление ветви

Чтобы найти точку присоединения листа *j* в *Tree*(D^{trimmed}), рассмотрим *Tree*(D^{bald}), который совпадает с *Tree*(*D*), за исключением того, что *LimbLength*(*j*) = 0. Из теоремы о длине ветви мы знаем, что в *Tree*(D^{bald}) должны быть листья *i* и *k* такие, что

$$\frac{D_{i,j}^{\text{bald}} + D_{j,k}^{\text{bald}} - D_{i,k}^{\text{bald}}}{2} = 0,$$

что подразумевает, что

$$D_{i,k}^{\text{bald}} = D_{i,j}^{\text{bald}} + D_{j,k}^{\text{bald}}.$$

Таким образом, точка крепления листа *j* должна находиться на расстоянии $D_{i,j}^{\text{bald}}$ от листа *i* на пути, соединяющем *i* и *k* в обрезанном дереве. Эта точка присоединения может находиться в существующем узле, и в этом случае мы подключаем *j* к этому узлу. С другой стороны, точка присоединения для *j* может располагаться вдоль ребра, и в этом случае мы помещаем новый узел в точку присоединения и соединяем с ним *j*.

Алгоритм реконструкции филогении по расстоянию

Предыдущее рассуждение приводит к следующему рекурсивному алгоритму, называемому **AdditivePhylogeny**, для нахождения простого дерева, соответствующего аддитивной матрице расстояний D размерностью $n \times n$. Мы предполагаем, что вы уже реализовали программу **Limb**(D, j), которая вычисляет $LimbLength(j)$ для листа j по матрице расстояний D. Вместо того чтобы выбирать произвольный лист j из дерева (D) для обрезки, **AdditivePhylogeny** выбирает лист n (соответствующий последней строке и столбцу D).

```
AdditivePhylogeny(D)
    n ← количество строк в D
    if n = 2
        return дерево, состоящее из единственного ребра длины D_{1,2}
    limbLength ← Limb(D, n)
    for j ← 1 до n – 1
        D_{j,n} ← D_{j,n} – limbLength
        D_{n,j} ← D_{j,n}
    (i, k) ← два листа, такие, что D_{i,k} = D_{i,n} + D_{n,k}
    x ← D_{i,n}
    D ← D со строкой n и столбцом n удалены
    T ← AdditivePhylogeny(D)
    v ← (потенциально новый) узел в T на расстоянии x от i на пути между i and k
    добавить лист n обратно в T, сделав ветвь (v, n) длины limbLength
    return T
```

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующие вопросы об алгоритме **AdditivePhylogeny**.

- Каково время его выполнения?
- Хотя может показаться, что **AdditivePhylogeny** построит дерево для любой матрицы, это не так. Что пойдет не так, если вы примените **AdditivePhylogeny** к неаддитивной матрице расстояний на рис. 7.10?
- Измените **AdditivePhylogeny**, чтобы разработать алгоритм, проверяющий, является ли данная матрица расстояний аддитивной. Затем примените этот тест к матрице расстояний для спайковых белков коронавируса (см рис. 7.17 ниже). Является ли эта матрица аддитивной? (Примечание: существует еще более простой способ проверки аддитивности. Дополнительные сведения см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Условие четырех точек**.)

Построение эволюционного дерева коронавирусов

К концу 2003 года биоинформатики секвенировали множество коронавирусов различных птиц и млекопитающих, из которых мы получили приведенную ниже матрицу расстояний на основе множественного выравнивания спайковых белков.

	Корова	Свинья	Лошадь	Мышь	Собака	Кошка	Индейка	Цивета	Человек
Корова	0	295	300	524	1077	1080	978	941	940
Свинья	295	0	314	487	1071	1088	1010	963	966
Лошадь	300	314	0	472	1085	1088	1025	965	956
Мышь	524	487	472	0	1101	1099	1021	962	965
Собака	1076	1070	1085	1101	0	818	1053	1057	1054
Кошка	1082	1088	1088	1098	818	0	1070	1085	1080
Индейка	976	1011	1025	1021	1053	1070	0	963	961
Цивета	941	963	965	962	1057	1085	963	0	16
Человек	940	966	956	965	1054	1080	961	16	0

Рис. 7.17 Матрица расстояний, основанная на попарном выравнивании спайковых белков из коронавирусов, извлеченных из различных животных. Расстояние между каждой парой последовательностей было рассчитано как общее количество несовпадений и вставок в их оптимальном выравнивании

Хотя теперь вы понимаете опасность вывода о том, что минимальный элемент матрицы расстояний соответствует паре соседей, здравый смысл подсказывает нам, взглянув на матрицу расстояний коронавируса, что цивета должна быть животным резервуаром атипичной пневмонии. Эта информация натолкнула исследователей на мысль о том, что причиной вспышки атипичной пневмонии могла быть неправильная подготовка мяса пальмовых цивет (см. фото ниже) в провинции Гуандун в Китае.

Рис. 7.18 Пальмовая цивета

Тем не менее, прежде чем спешить с таким выводом, вы можете прочитать **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Являются ли летучие мыши истоянико эпидемии атипичной пневмонии?**, чтобы понять, почему историю межвидовой передачи вируса часто трудно проследить. Фактически некоторые исследования показали, что люди сначала заразились SARS от летучих мышей, которые позже передали вирус пальмовым циветам, а те передали болезнь обратно людям. Цивета была идентифицирована как животное-резервуар атипичной пневмонии в 2003 году отчасти потому, что вирусы атипичной пневмонии от других потенциальных подозреваемых, включая летучих мышей, еще не были секвенированы.

Поскольку матрица расстояний для SARS-подобных коронавирусов неаддитивна, мы схитрим и немного изменим ее, сделав ее аддитивной, чтобы к ней можно было применить **AdditivePhylogeny** (рисунок ниже).

	Корова	Свинья	Лошадь	Мышь	Собака	Кошка	Индейка	Цивета	Человек
Корова	0	295	306	497	1081	1091	1003	956	954
Свинья	295	0	309	500	1084	1094	1006	959	957
Лошадь	306	309	0	489	1073	1083	995	948	946
Мышь	497	500	489	0	1092	1102	1014	967	965
Собака	1081	1084	1073	1092	0	818	1056	1053	1051
Кошка	1091	1094	1083	1102	818	0	1066	1063	1061
Индейка	1003	1006	995	1014	1056	1066	0	975	973
Цивета	956	959	948	967	1053	1063	975	0	16
Человек	954	957	946	965	1051	1061	973	16	0

Рис. 7.19 Модификация матрицы расстояний спайкового белка, превращающая ее в аддитивную

Упражнение. Постройте простое дерево, соответствующее следующей матрице расстояний.

Использование метода наименьших квадратов для построения приблизительных филогений

Если матрица расстояний D размерностью $n \times n$ неаддитивна, тогда мы будем искать взвешенное дерево T, расстояния между листьями которого аппроксимируют элементы в D. С этой целью мы хотели бы, чтобы T минимизировало **среднее квадратичное отклонение (дисперсию)**. $Discrepancy(T, D)$, которая задается формулой

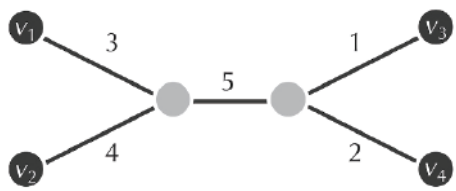
$$Discrepancy(T, D) = \sum_{1 \le i < j \le n} (d_{i,j}(T) - D_{i,j})^2.$$

Задача построения филогении на основе метода наименьших квадратов: *по заданной матрице расстояний найти дерево, минимизирующее сумму квадратичных отклонений.*

Input: матрица расстояний D размерностью $n \times n$.

Output: взвешенное дерево T, минимизирующее *Disrepancy*(T, D) по всем взвешенным деревьям с n листьями.

Рассмотрим следующее помеченное дерево T и матрицу расстояний D. Вычислите *Disrepancy*(T, D).



	v_1	v_2	v_3	v_4
v_1	0	3	4	3
v_2	3	0	4	5
v_3	4	4	0	2
v_4	3	5	2	0

Рис. 7.20 Дерево T и матрица расстояний D

Упражнение. Пусть T будет деревом, показанным ниже.

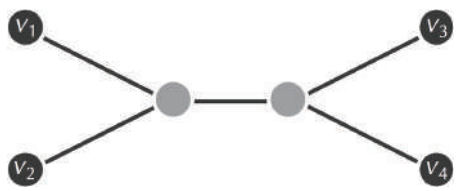


Рис. 7.21 Дерево T

Для приведенной ниже неаддитивной матрицы расстояний D 4×4 (рис. 7.22) найдите длины ребер в этом дереве, которые минимизируют *Disrepancy*(T, D). Подсказка: это не длина, указанная ранее ☺.

Оказывается, для конкретного дерева T легко найти длины ребер, минимизирующие *Discrepancy*(T, D). Тем не менее наша способность минимизировать сумму квадратов отклонений для конкретного дерева не означает, что мы можем эффективно решить задачу филогении по расстояниям методом наи-

меньших квадратов, поскольку количество различных деревьев очень быстро растет по мере увеличения количества листьев в дереве. На самом деле филогения, основанная на методе наименьших квадратов, оказывается *NP*-полной, и поэтому мы должны оставить надежду на разработку быстрого алгоритма для нахождения дерева, которое лучше всего соответствует неаддитивной матрице. В следующих двух разделах мы рассмотрим эвристики построения деревьев из неаддитивных матриц, которые решают эту задачу приближенно.

	v_1	v_2	v_3	v_4
v_1	0	3	4	3
v_2	3	0	4	5

Рис. 7.22 Матрица расстояний *D*.

Ультраметрические эволюционные деревья

Биологи часто предполагают, что каждый внутренний узел в эволюционном дереве соответствует виду, который претерпел **видообразование**, разделившее один вид на два вида-потомка. Обратите внимание, что каждый внутренний узел в дереве на рисунке ниже (соответствующий событию видообразования) имеет степень 3. Поэтому мы определяем **некорневое бинарное дерево** как дерево, в котором каждый узел имеет степень, равную 1 или 3.

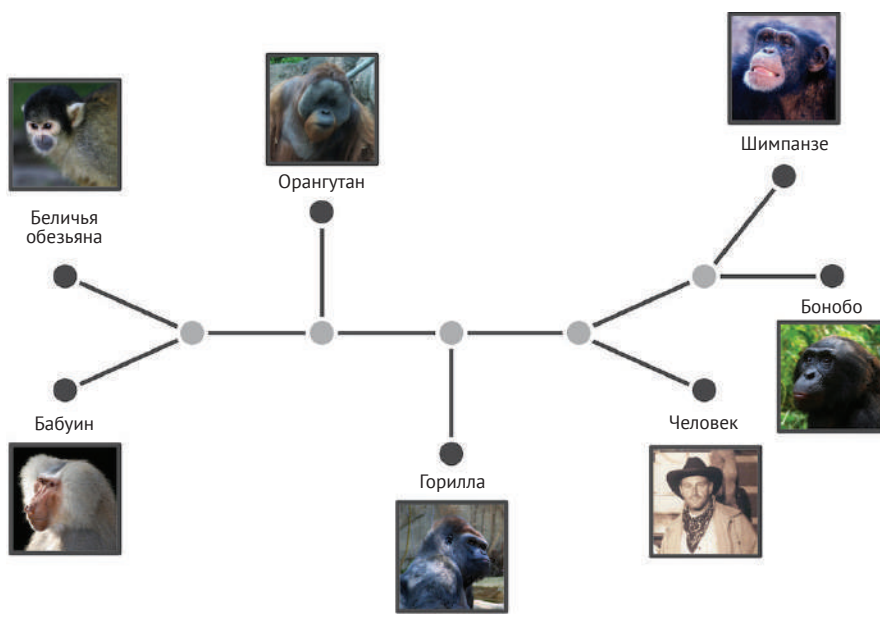


Рис. 7.23 Бинарное некорневое дерево, моделирующее филогению приматов

Упражнение. Докажите, что каждое некорневое двоичное дерево с n листьями имеет $n - 2$ внутренних узла (и, следовательно, $2n - 3$ ребра).

Корневое бинарное дерево – это некорневое бинарное дерево, у которого есть корень (степени 2), расположенный на одном из его ребер; другими словами, мы заменяем ребро (v, ω) корнем и рисуем ребра, соединяющие корень с каждым из v и ω.

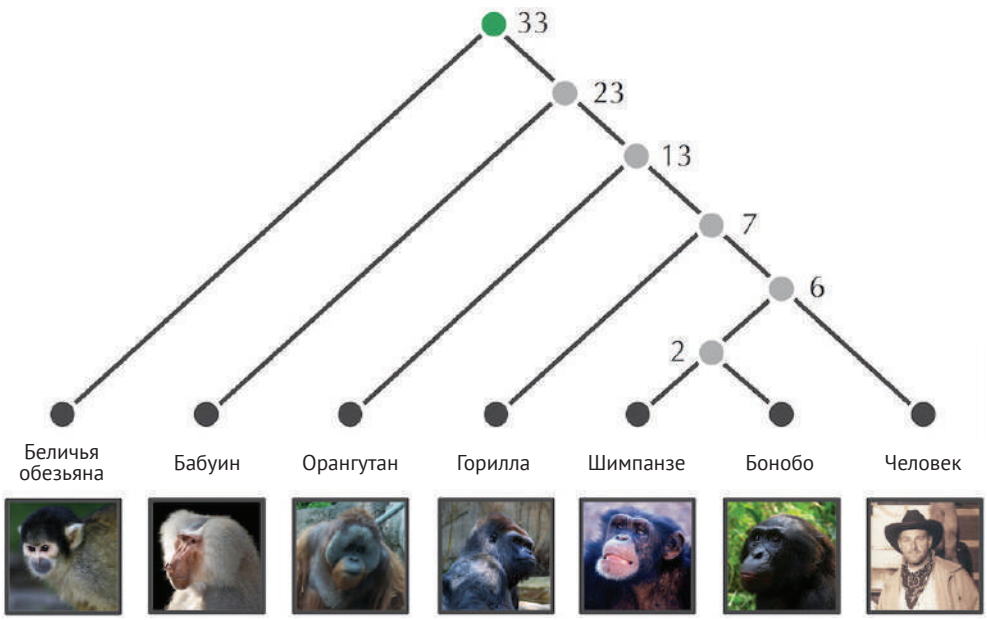


Рис. 7.24 Если поместить корень на ветвь беличьей обезьяны, получится корневое бинарное дерево. Число в каждом узле соответствует количеству миллионов лет назад, когда произошло расхождение в этом узле. Изображение предоставлено: Джули Лэнгфорд (беличья обезьяна), Андре Карват (бабуин), Франс де Вааль (шимпанзе), Кабир Баки (горилла)

Если бы у нас были **молекулярные часы**, измеряющие время эволюции, то мы могли бы присвоить возраст каждому узлу v в корневом бинарном дереве (обозначаемом как $Age(v)$), где все листья дерева имеют возраст 0, поскольку они соответствуют видам в настоящее время. Затем мы могли бы определить вес ребра (v, ω) в дереве как разность $Age(v) - Age(\omega)$. Следовательно, длина пути между корнем и любым узлом будет равна разнице между их возрастами. Такое дерево, в котором расстояние от корня до любого листа одинаково, называется ультраметрическим (рисунок ниже).

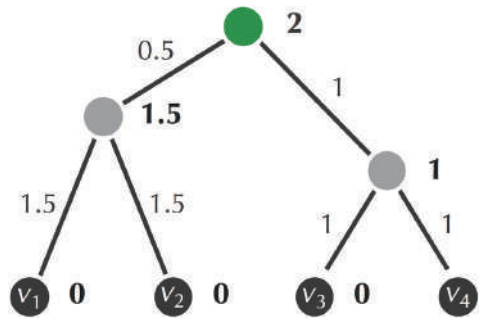


Рис. 7.25 Ультраметрическое дерево. Каждому узлу был присвоен возраст (выделен полужирным шрифтом), возраст каждого листа равен нулю. Длина каждого ребра равна разнице между возрастами узлов, которые соединяет ребро, а расстояние от корня до любого листа равно 2, возрасту корня

Наша цель – получить ультраметрическое дерево, соответствующее заданной матрице расстояний (даже если оно делает это только приблизительно). **UPGMA** (что означает невзвешенный метод парных групп со средним арифметическим) – это простая эвристика кластеризации, которая вводит гипотетические молекулярные часы для построения ультраметрического эволюционного дерева. Вы можете узнать больше о кластеризации в следующей главе.

Имея матрицу D размерностью $n \times n$, **UPGMA** (как показано на рис. 7.26) сначала формирует n тривиальных кластеров, каждый из которых содержит один лист. Затем алгоритм находит пару «ближайших» кластеров. Чтобы уточнить понятие ближайших кластеров, **UPGMA** определяет расстояние между кластерами C_1 и C_2 как среднее попарное расстояние между элементами C_1 и C_2:

$$D_{C_1,C_2} = \frac{\sum_{i \in C_1} \sum_{j \in C_2} D_{i,j}}{|C_1| \cdot |C_2|}.$$

В этом уравнении $|C|$ обозначает количество листьев в кластере C.

Как только **UPGMA** определяет пару ближайших кластеров C_1 и C_2, он объединяет их в кластер C с $|C_1| + |C_2|$ элементами, а затем создает узел для C, который соединяется с каждым из C_1 и C_2 направленным ребром. Возраст C устанавливается как $D_{C_1,C_2}/2$. Затем **UPGMA** повторяет этот процесс объединения двух ближайших кластеров до тех пор, пока не останется только один кластер, соответствующий корню.

Упражнение. Докажите, что после объединения кластеров C_i и C_j в кластер C_{new} расстояние между C_{new} и другим кластером C_m равно $(D_{C_i,C_m} \cdot |C_i| + D_{C_j,C_m} \cdot |C_j|)/(|C_i| + |C_j|)$.

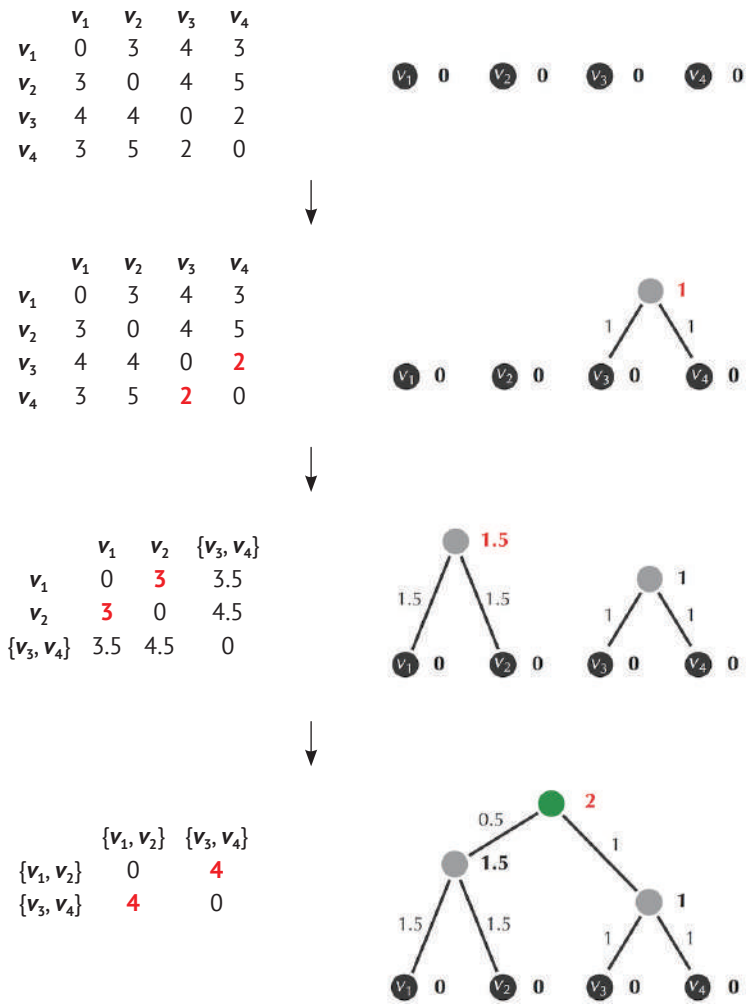


Рис. 7.26 Реконструкция дерева с помощью **UPGMA** для неаддитивной матрицы расстояний, с которой мы сталкивались ранее. **UPGMA** начинается с формирования одного кластера для каждого листа. На каждом шаге этот метод идентифицирует два ближайших кластера C_1 и C_2, объединяет их в новый узел C и соединяет C с C_1 и C_2 направленными ребрами. Возраст C устанавливается равным $D_{C_1,C_2}/2$. Затем мы повторяем этот процесс до тех пор, пока не останется только один кластер, который должен быть корнем

Псевдокод для **UPGMA** показан ниже.

UPGMA(*D, n*)
Clusters ← *n* кластеров с единственным элементом, обозначенных как 1, ..., *n*
постройте граф *T* с *n* изолированными узлами, обозначенными единичными

```
    элементами 1, ... , n
    for каждого узла v в графе T
        Age(v) ← 0
    while существует более одного кластера
        найдите два ближайших кластера C_i и C_j
        объедините C_i и C_j в новый кластер C_new с |C_i| + |C_j| еэлементами
        добавьте новый узел, обозначенный кластером C_new до T
        соедините узел C_new с C_i и C_j направленными ребрами
        Age(C_new) ← D_{C_i,C_j}/2
        удалите строки и столбцы матрицы D в соответствии с C_i и C_j
        удалите C_i и C_j из Clusters
        добавьте строку/столбец к D для C_new, вычислив D(C_new, C) для каждого C
        в Clusters
        добавьте C_new к Clusters
    корень ← узел в T, соответствующий оставшемуся кластеру
    for каждого ребра (v, ω) в T
        длина (v, ω) ← Age(v) – Age(ω)
    return T
```

Упражнение. Примените алгоритм **UPGMA** к (неаддитивной) матрице расстояний от коронавируса, воспроизведенной ниже.

	Корова	**Свинья**	**Лошадь**	**Мышь**	**Собака**	**Кошка**	**Индейка**	**Цивета**	**Человек**
Корова	0	295	300	524	1077	1080	978	941	940
Свинья	295	0	314	487	1071	1088	1010	963	966
Лошадь	300	314	0	472	1085	1088	1025	965	956
Мышь	524	487	472	0	1101	1099	1021	962	965
Собака	1076	1070	1085	1101	0	818	1053	1057	1054
Кошка	1082	1088	1088	1098	818	0	1070	1085	1080
Индейка	976	1011	1025	1021	1053	1070	0	963	961
Цивета	941	963	965	962	1057	1085	963	0	16
Человек	940	966	956	965	1054	1080	961	16	0

UPGMA предлагает шаг вперед по сравнению с **AdditivePhylogeny**, поскольку он может анализировать неаддитивные матрицы расстояний. На рисунке ниже показан результат применения **UPGMA** к матрице расстояний коронавируса. Однако первым шагом, который делает **UPGMA**, является объединение двух листьев i и j с минимальным расстоянием $D_{i,j}$ в один кластер. Но мы уже видели, что наименьший элемент матрицы расстояний не обязательно соответствует паре соседних листьев! Это вызывает беспокойство, поскольку если **UPGMA** генерирует неправильные деревья из аддитивных матриц, то это неидеальная эвристика для построения эволюционного дерева из них. Можем ли мы найти алгоритм, который всегда идентифицирует соседние листья в ад-

дитивной матрице расстояний, но также хорошо работает и на неаддитивной матрице расстояний?

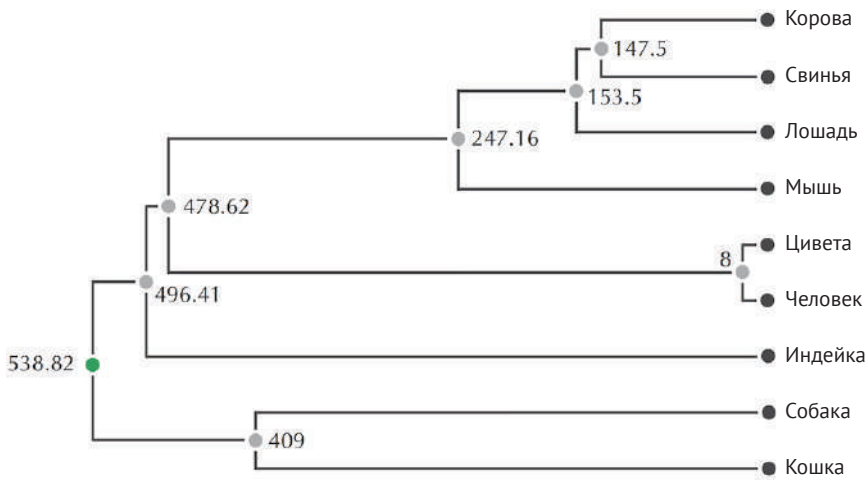


Рис. 7.27 Ультраметрическое дерево коронавирусов, созданное **UPGMA** с использованием матрицы расстояний коронавируса. Корень показан зеленым цветом

Алгоритм объединения соседей

Преобразование матрицы расстояний в матрицу объединения соседей

В 1987 году Наруя Сайтоу и Масатоши Ней разработали **алгоритм объединения соседей** для реконструкции эволюционного дерева. Имея аддитивную матрицу расстояний, этот алгоритм, который мы называем **NeighborJoining**, находит пару соседних листьев и заменяет их одним листом, тем самым уменьшая размер дерева. Таким образом, **NeighborJoining** может рекурсивно построить дерево, соответствующее аддитивной матрице. Этот алгоритм также обеспечивает эвристику для неаддитивных матриц расстояний, которая хорошо работает на практике.

Основная идея **NeighborJoining** заключается в том, что, хотя поиск минимального элемента в матрице расстояний D не гарантирует получение пары соседей в $Tree(D)$, мы можем преобразовать D в другую матрицу, минимальный элемент которой дает пару соседей. Во-первых, для заданной матрицы расстояний D размерностью $n \times n$ мы определяем $TotalDistance_D(i)$ как сумму $\sum_{1 \le k \le n} D_{i,k}$ расстояний от листа i до всех остальных листьев. **Матрица объединения соседей** D^* (см. ниже) определяется таким образом, что для любых i и j $D^*_{i,i} = 0$ и

$$D^*_{i,i} = (n - 2) \cdot D_{i,i} - TotalDistance_D(i) - TotalDistance_D(j).$$

D		*i*	*j*	*k*	*l*	$TotalDistance_D$
	i	0	13	21	22	56
	j	13	0	12	13	38
	k	21	12	0	13	46
	l	22	13	13	0	48

D		*i*	*j*	*k*	*l*
	i	0	−68	−60	−60
	j	−68	0	−60	−60
	k	−60	−60	0	−68
	l	−60	−60	−68	0

Рис. 7.28 Матрица объединения соседей

NeighborJoining (показано выше) – широко используемый метод реконструкции эволюционного дерева; статья, в которой он был представлен, является одной из самых цитируемых во всей науке – она имеет более 30 000 ссылок. Однако этот алгоритм не является интуитивным: приведенная выше формула для вычисления матрицы D^*, вероятно, покажется вам колдовством. На самом деле, несмотря на безупречную интуицию, Сайтоу и Ней так и не доказали, что их алгоритм правильно решает задачу филогении по расстояниям для аддитивных матриц! Исследователям потребовался еще год, чтобы доказать следующую теорему, чье (длинное) доказательство можно найти в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Почему алгоритм объединения соседей находит соседние листья?**.

Теорема объединения соседей. Для аддитивной матрицы D наименьший элемент $D^*_{i,j}$ ее матрицы объединения соседей D^* показывает пару соседних листьев i и j в $Tree(D)$.

Если $n = 2$, то $NeighborJoining(D, n)$ выдает дерево, состоящее из одного ребра длины $D_{1,2}$. Если $n > 2$, то он выбирает минимальный элемент в матрице объединения соседей, заменяет соседние листья i и j новым листом m, а затем вычисляет расстояние от m до любого другого листа k по формуле

$$D_{k,m} = (1/2)(D_{k,i} + D_{k,j} - D_{i,j}),$$

что мотивировано **AdditivePhylogeny**. Это уравнение позволяет заменить матрицу D размера $n \times n$ на матрицу D размерностью $(n - 1) \times (n - 1)$, в которой i и j заменены на m. Рекурсивно применяя $NeighborJoining$ к D', мы получаем эволюционное дерево с $n - 1$ листом. Затем мы добавляем две ветви, начиная с вершины m, одну заканчивая листом i, а другую заканчивая листом j. Мы устанавливаем

$$\Delta_{i,j} = (TotalDistance_D(i) - TotalDistance_D(j)) / (n - 2)$$

и назначаем

$$LimbLength(i) = (1/2)\ (D_{i,j} + \Delta_{i,j})$$
$$LimbLength(j) = (1/2)\ (D_{i,j} - \Delta_{i,j}).$$

Примечание Чтобы узнать, откуда берутся эти формулы, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Вычисление длин ветвей в алгоритме объединения соседей**.

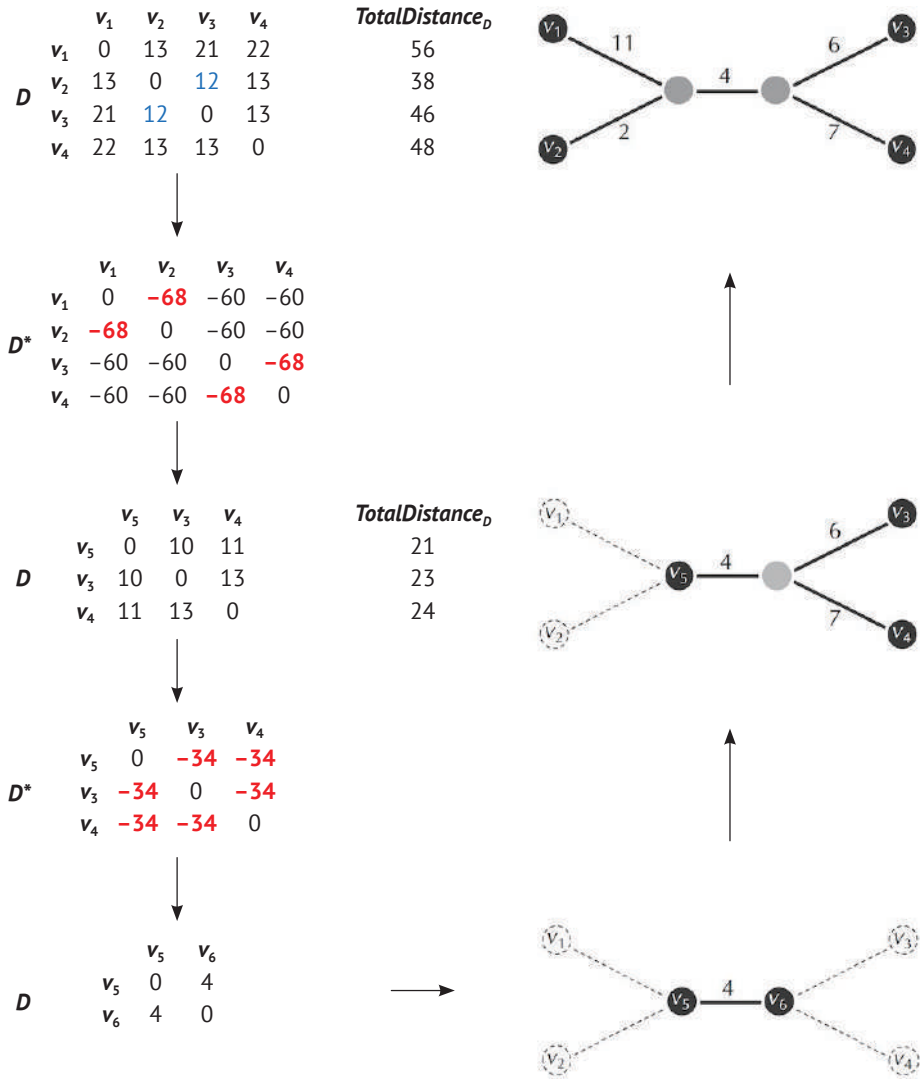


Рис. 7.29 (Вверху слева) Аддитивная матрица расстояний D 4×4, которую мы рассмотрели ранее, вместе с массивом $TotalDistance_D$. D_{v_2,v_3} (показаны синим цветом) – минимальный элемент D, но, как оказалось, листья v_2 и v_3 не являются соседями в $Tree(D)$. Двигаясь вниз по левой стороне, мы строим матрицу объединения соседей D^* и находим, что $D^*_{v_1,v_2}$ (красный) является минимальным элементом D. Затем мы преобразуем исходную матрицу расстояний 4×4 в матрицу расстояний 3×3 с помощью замены v_1 и v_2 одним листом v_5 и обновления расстояний от v_5 до других листьев как $D_{v_3,v_5} = (1/2)(D_{v_3,v_1} + D_{v_3,v_2} - D_{v_1,v_2}) = (1/2)(21 + 12 - 13) = 10$ и $D_{v_4,v_5} = (1/2)(D_{v_4,v_1} + D_{v_4,v_2} - D_{v_1,v_2}) = (1/2)(22 + 13 - 13) = 11$. Все элементы являются минимальными элементами в новой матрице D^*, поэтому мы произвольно выбираем v_3 и v_4 в качестве соседей для замены одним листом v_6. Полученная матрица 2×2 соответствует дереву с единственным ребром, соединяющим v_5 и v_6. Затем мы продвигаемся вверх по правой стороне, добавляя пары соседей обратно в дерево на каждом шаге, используя ранее вычисленные формулы для длин ветвей. (Вверху справа) Дерево $Tree(D)$, соответствующее исходной матрице D

Упражнение. Докажите, что если D аддитивна, то для любых i и j между 1 и n оба $(1/2)\ (D_{i,j} + \Delta_{i,j})$ и $(1/2)\ (D_{i,j} - \Delta_{i,j})$ неотрицательны.

Приведенный ниже псевдокод суммирует алгоритм объединения соседей.

```
NeighborJoining(D)
    n ← количество строк в D
    if n = 2
        T ← дерево, состоящее из единственного ребра длины D_{1,2}
        return T
    D* ← матрица объединения с соведями, построенная из матрицы расстояний D
    Найдите элементы i и j такие, что D*_{i,j} – минимальный недиагональный
    элемент D*
    Δ ← (TotalDistance_D(i) – TotalDistance_D(j)) / (n – 2)
    limbLength_i ← (1/2)(D_{i,j} + Δ)
    limbLength_j ← (1/2)( D_{i,j} – Δ)
    добавьте новую строку/столбец m к D такую, что D_{k,m} = D_{m,k} = (1/2)( D_{k,i} + D_{k,j} –
    D_{i,j}) для любого k
    D ← D с удаленными строками i и j
    D ← D с удаленными столбцами i и j
    T ← NeighborJoining(D)
    добавьте две новые ветви (соединяющие узел m с листьями i и j) к дереву T
    добавьте длину limbLength_i к Limb(i)
    добавьте длину limbLength_j к Limb(j)
    return T
```

Упражнение. Перед реализацией алгоритма объединения соседей попробуйте применить его к аддитивным и неаддитивным матрицам расстояний, показанным ниже.

	v_1	v_2	v_3	v_4
v_1	0	13	21	22
v_2	13	0	12	13
v_3	21	12	0	13
v_4	22	13	13	0

	v_1	v_2	v_3	v_4
v_1	0	3	4	3
v_2	3	0	4	5
v_3	4	4	0	2
v_4	3	5	2	0

Рис. 7.30 Аддитивная и неаддитивная матрицы расстояний

Упражнение. Примените **NeighborJoining** к неаддитивной матрице расстояний коронавируса, воспроизведенной ниже.

	Корова	Свинья	Лошадь	Мышь	Собака	Кошка	Индейка	Цивета	Человек
Корова	0	295	300	524	1077	1080	978	941	940
Свинья	295	0	314	487	1071	1088	1010	963	966
Лошадь	300	314	0	472	1085	1088	1025	965	956
Мышь	524	487	472	0	1101	1099	1021	962	965
Собака	1076	1070	1085	1101	0	818	1053	1057	1054
Кошка	1082	1088	1088	1098	818	0	1070	1085	1080
Индейка	976	1011	1025	1021	1053	1070	0	963	961
Цивета	941	963	965	962	1057	1085	963	0	16
Человек	940	966	956	965	1054	1080	961	16	0

Анализ коронавирусов с помощью алгоритма объединения соседей

На рисунке ниже показано дерево соседствующих коронавирусов, выделенных от разных животных, по неаддитивной матрице расстояний, представленной выше. (Веса ребер округляются до ближайшего целого числа.)

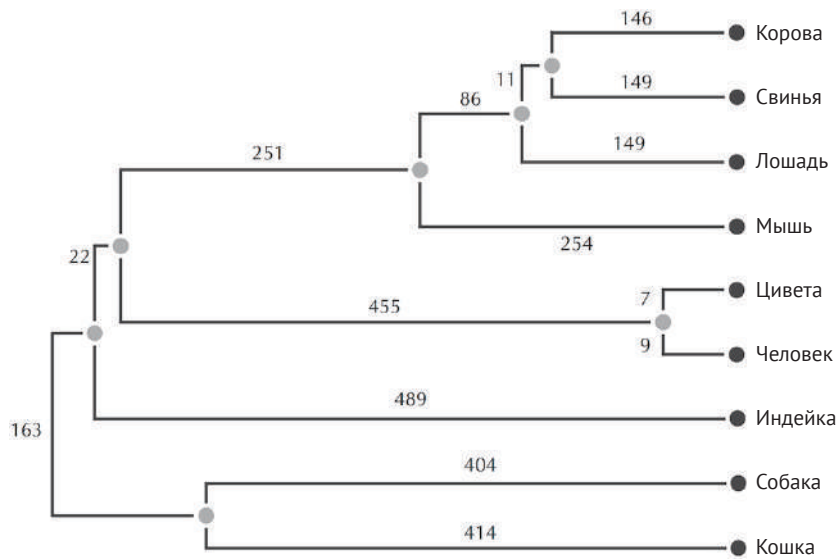


Рис. 7.31 Дерево соседствующих коронавирусов

Мы также можем применить **NeighborJoining** к матрице расстояний вариантов SARS-CoV, выделенных от различных носителей-людей, в дополнение к коронавирусу пальмовой циветы (рис. 7.32).

Штамм SARS-CoV, обозначенный на этом рисунке как «Ханой», был взят у Карло Урбани, итальянского врача, работавшего во Всемирной организации здравоохранения. В феврале 2003 года Урбани вызвали в больницу Ханоя для

осмотра пациента, который заболел тем, что местные врачи сочли тяжелым случаем гриппа; эти врачи боялись, что это может быть птичий грипп. На самом деле пациентом был американец, который всего неделю назад останавливался напротив Лю Цзяньлуня в отеле «Метрополь». К счастью, Урбани быстро понял, что это не грипп, и стал первым врачом, поднявшим тревогу среди

	Гуаньчжоу: 16 декабря 2002	Чжуньшань: 16 декабря 2002	Гуаньчжоу: 24 января 2003	Гуаньчжоу: 31 января 2003	Гуаньчжоу: 18 февраля 2003	Гонконг: 18 февраля 2003	Ханой: 26 февраля 2003	Торонто: 17 февраля 2003	Гонконг: 15 марта 2003	Пальмовая цивета
Гуаньчжоу	0	4	12	8	9	9	12	12	11	3
Чжуньшань	4	0	10	6	7	7	10	10	9	3
Гуаньчжоу	12	10	0	4	5	3	2	2	1	11
Гуаньчжоу	8	6	4	0	3	1	4	4	3	7
Гуаньчжоу	9	7	5	3	0	2	5	5	4	8
Гонконг	9	7	3	1	2	0	3	3	2	8
Ханой	12	10	2	4	5	3	0	2	1	11
Торонто	12	10	2	4	5	3	2	0	1	11
Гонконг	11	9	1	3	4	2	1	1	0	10
Пальмовая цивета	3	3	11	7	8	8	11	11	10	0

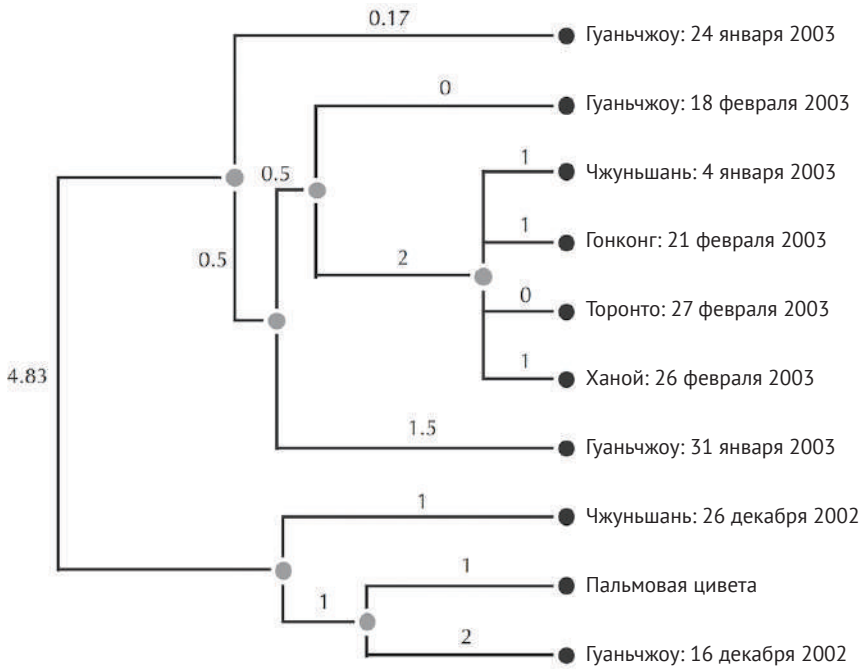


Рис. 7.32 (Вверху) Матрица расстояний, основанная на попарном выравнивании спайковых белков из штаммов SARS-CoV, выделенных у разных пациентов, а также коронавируса, взятого у пальмовой циветы. Расстояние между каждой парой последовательностей рассчитывали как общее количество несовпадений и вставок в их оптимальном попарном выравнивании. (Внизу) Эволюционное дерево этих вирусов, построенное с помощью алгоритма объединения соседей

чиновников здравоохранения. Однако вместо того, чтобы покинуть Ханой, он потребовал, чтобы его оставили там для наблюдения за карантинными процедурами. В ссоре с женой, которая ругала его за то, что он рисковал жизнью, чтобы лечить больных, Урбани ответил: «Зачем я здесь? Отвечать на электронные письма, ходить на коктейльные вечеринки и перебирать бумажки?» В конечном итоге месяц спустя Урбани погиб от атипичной пневмонии. Но его жертва помогла начать масштабную всемирную борьбу с этой болезнью, которая, возможно, спасла миллионы жизней.

Ограничения методов реконструкции эволюционного дерева по расстояниям

Хотя реконструкция дерева по расстояниям успешно разрешила вопросы о происхождении и распространении атипичной пневмонии, многие эволюционные противоречия не могут быть разрешены с помощью матрицы расстояний. Например, когда мы преобразуем каждую пару строк множественного выравнивания в значение расстояния, мы теряем информацию, содержащуюся в выравнивании. В результате методы, основанные на расстоянии, не позволяют нам реконструировать последовательности спайковых белков предков (соответствующие внутренним узлам филогении спайковых белков), что может привести нас к мысли, что такая молекулярная палеонтология невозможна. Следовательно, лучшим методом реконструкции эволюционного дерева было бы каким-то образом использовать выравнивание напрямую, без предварительного преобразования его в матрицу расстояний.

Реконструкция эволюционного дерева по признакам

Таблицы признаков

Пятьдесят лет назад биологи строили филогении не из последовательностей ДНК и белков, а из анатомических и физиологических особенностей, называемых признаками. Например, при анализе эволюции беспозвоночных одним из часто используемых признаков является наличие или отсутствие крыльев, а другим – количество ног (от 0 до более чем 300 у некоторых многоножек). Эти два признака приводят к таблице признаков размерностью 3×2, показанной на рис. 7.33 для трех видов.

В общем, каждая строка в таблице признаков $n \times m$ представляет **вектор признаков**, содержащий значения m признаков, соответствующих одному из n существующих видов. Наша цель, грубо говоря, состоит в том, чтобы построить эволюционное дерево, в котором листья, соответствующие современным видам с похожими векторами признаков, расположены рядом друг с другом в дереве. Мы также хотели бы присвоить m значений признаков каждому

внутреннему узлу в дереве, чтобы наилучшим образом объяснить признаки видов предков.



	крылья	ноги
крылатый палочник	Да	6
бескрылый палочник	Нет	6
гигантская многоножка	Нет	42

Рис. 7.33 (Вверху) Крылатые (слева) и бескрылые (в центре) палочники, у каждого из которых по шесть ног, и гигантская многоножка (справа), у которой 42 ноги. (Внизу) Таблица признаков 3×2 описывает два признака (крылья и ноги) у этих трех беспозвоночных. Изображение предоставлено: Drägüs, пользователь Викисклада (крылатые палочники), Л. Шьямал (бескрылые палочники), Катка Немчокова (гигантская многоножка)

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы преобразовать приведенное выше расплывчатое описание в хорошо сформулированную вычислительную задачу, которая моделирует реконструкцию дерева по таблице признаков?

От анатомических к генетическим признакам

В 1965 году Эмиль Цукеркандл и Линус Полинг опубликовали статью «Молекулы как документы эволюционной истории», утверждая, что последовательности ДНК представляют собой гораздо более информативный источник данных, чем анатомические и физиологические признаки. Сегодня эта идея кажется очевидной, особенно после того, как мы потратили половину главы на построение эволюционных деревьев из матриц расстояний, сгенерированных из последовательностей ДНК. Однако предложение Цукеркандла и Полинга поначалу было скептически встречено многими биологами, которые считали, что анализ ДНК не может иметь ту же силу, как анатомическое сравнение. Популярный сейчас довод был выдвинут, когда Цукеркандл и Полинг обнаружили, что аминокислотная последовательность бета-гемоглобина человека очень похожа на последовательность горилл, что побудило Цукеркандла написать в 1963 году:

> *С точки зрения структуры гемоглобина получается, что горилла – это просто ненормальный человек.*

Тем не менее удивительное сходство между белками гемоглобина у разных приматов противоречило явным анатомическим различиям между приматами. В результате ведущий биолог-эволюционист Гейлорд Симпсон немедленно ответил Цукеркандлу:

> *...это, конечно, ерунда. На что действительно указывает сравнение, так это на то, что гемоглобин – плохой выбор, и ничего не говорит нам о признаках или говорит очевидную ложь.*

Несмотря на такую яростную первоначальную критику, к 1970-м годам генетический анализ стал доминирующим методом в эволюционных исследованиях. Действительно, анализ последовательностей ДНК ответил на эволюционные вопросы, которые не удалось разрешить на основе анализа анатомических признаков, используемого ранее. Ранние примеры включают классификацию гигантской панды и определение происхождения человека (подробности см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Гигантская панда: медведь или енот?** и **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Откуда пришли люди?**). В результате у большинства экспертов-эволюционистов не было иного выбора, кроме как пересмотреть свои убеждения и стать экспертами в молекулярной эволюции.

По иронии судьбы современные эволюционные исследования часто рассматривают множественное выравнивание $n \times m$ как таблицу признаков $n \times m$, где каждый столбец представляет отдельный признак. Наша цель – построить дерево, листья которого соответствуют строкам этого выравнивания, а внутренние узлы – наследственным последовательностям в соответствии с наиболее экономным эволюционным сценарием. Прежде чем строго определить наиболее экономный сценарий, мы опишем один пример того, как эта алгоритмическая процедура решила давнюю загадку эволюции насекомых.

Сколько раз эволюция изобретала крылья для насекомых?

Крылья обеспечили насекомым революционную адаптацию, позволив им убегать от хищников и расселяться по новым территориям, что привело к появлению множества новых видов насекомых. Тем не менее, несмотря на эволюционные преимущества, обеспечиваемые крыльями, некоторые насекомые, по-видимому, лучше приспособлены к выживанию без них. На самом деле почти все крылатые виды имеют бескрылых родственников, принадлежащих к тому же роду, а некоторые отряды насекомых бескрылы целиком, включая блох и вшей.

Приобретение крыльев, казалось бы, представляет собой эволюционную задачу, потому что для полета необходимы сложные физиологические взаимодействия. В результате мы бы пришли к выводу, что крылья развились у насекомых только один раз. Этот аргумент аналогичен **принципу необратимости Долло**, гипотезе, предложенной палеонтологом XIX века Луи Долло. В соответствии с этим принципом, когда вид теряет сложный орган, например крылья, этот орган не появляется вновь в точно такой же форме у потомков вида.

ОСТАНОВИТЕСЬ и задумайтесь. Что вы думаете об этом аргументе, касающемся крыльев насекомых?

До недавнего времени биологи следовали принципу Долло в отношении крыльев насекомых, полагая, что повторная эволюция крыльев практически невозможна, потому что неиспользуемые гены полета у бескрылых насекомых могут свободно накапливать мутации, в конечном итоге превращаясь в нефункциональные псевдогены. Однако в 2003 году Майкл Уайтинг изучил различных крылатых и бескрылых палочников со всего мира и опроверг этот аргумент. Он секвенировал сегмент длиной примерно 2000 нуклеотидов (ген **рибосомной РНК 18S**) этих палочников и построил эволюционное дерево по этим последовательностям. Из этой филогении он сделал вывод, что крылья заново изобретались по крайней мере три раза и терялись по крайней мере четыре раза в ходе эволюции палочников (рис. 7.34).

ОСТАНОВИТЕСЬ и задумайтесь. Возможно ли вывести эволюционный сценарий из приведенного ниже дерева, в котором крылья изобретаются менее четырех раз?

Работа Уайтинга указывает на сложности, присущие попыткам вывести эволюционные деревья из анатомических признаков. Если бы вас попросили разработать основанный на признаках алгоритм построения филогении для коллекции всех палочников, то одним из первых шагов, который вы, вероятно, предприняли бы, было бы сгруппировать всех крылатых насекомых на противоположной стороне дерева, подальше от всех бескрылых. Однако повторная эволюция крыльев насекомых означает, что такой подход ошибочен. Анатомические признаки имеют еще меньше силы, когда мы перемещаемся в микроскопический мир: только представьте, что вы пытаетесь построить филогению на основе непосредственного наблюдения за коронавирусами!

Задача минимального показателя экономии

Пометим каждый лист эволюционного дерева строкой множественного выравнивания и попытаемся вывести строки, помечающие внутренние узлы и соответствующие последовательности предков-кандидатов. Однако нам нужно будет разработать **оценочную функцию** (функцию счета), чтобы количественно определить, насколько хорошо такое помеченное дерево соответствует заданному множественному выравниванию. Далее для простоты будем считать, что множественное выравнивание содержит только замены и не содержит вставок. На практике исследователи могут начать с множественного выравнивания, содержащего вставки, а затем удалить все столбцы, содержащие вставки.

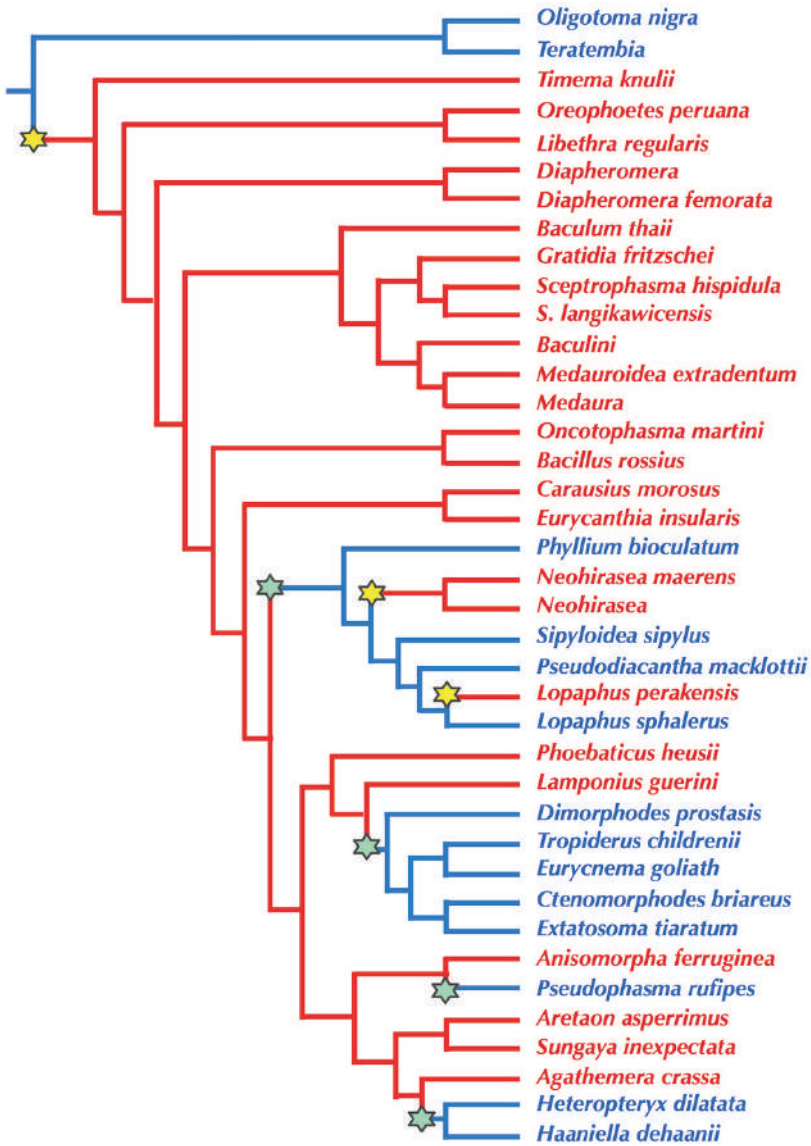


Рис. 7.34 Эволюционное древо крылатых (синие) и бескрылых (красные) палочников, построенное по генам 18S рибосомой РНК. Переходы от крылатых к бескрылым видам показаны желтыми звездочками; переходы от бескрылых к крылатым – зелеными звездочками. 18S рибосомные РНК эволюционируют медленно, что делает их идеальными для реконструкции древних расхождений

Интуитивно счет эволюционного дерева – это общее количество мутаций, необходимых для объяснения строк во всех узлах дерева. Для дерева T, в котором каждый узел помечен строкой длины m, мы установим длину ребра (v, ω)

равной количеству замен (расстоянию Хэмминга) между строками, помечающими v и ω. **Показатель экономии** T представляет собой сумму длин его ребер (рисунок ниже).

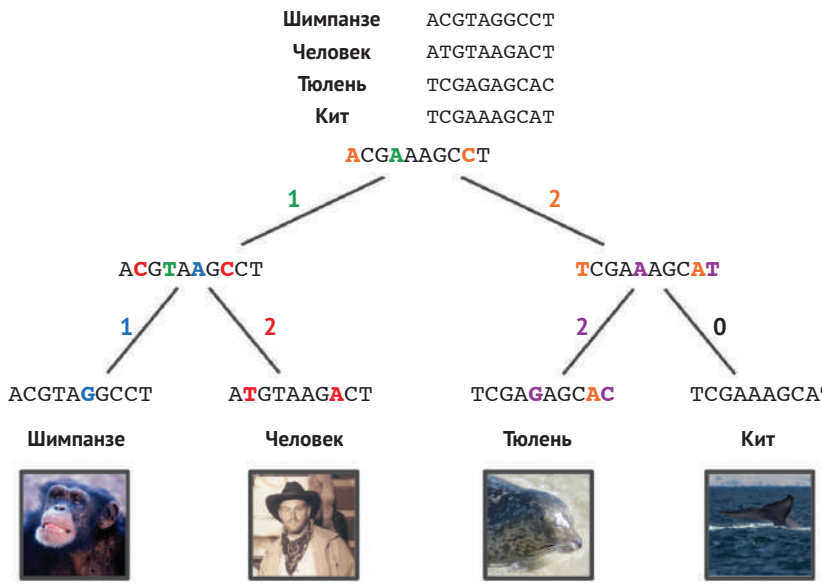


Рис. 7.35 (Вверху) Множественное выравнивание млекопитающего, с которым мы начали работать ранее в этой главе. (Внизу) Эволюционное дерево с показателем экономии 8, чьи листья являются цепочками ДНК в этом множественном выравнивании. Цветные буквы обозначают несовпадения в строках, соединенных ребром

Сначала мы предположим, что нам заранее дана структура корневого бинарного дерева, и в этом случае нам нужно только присвоить строки внутренним узлам, чтобы минимизировать показатель экономии.

Задача минимального показателя экономии: *найдите наиболее экономную маркировку внутренних узлов корневого дерева.*

Input: корневое бинарное дерево, в котором каждый лист помечен строкой длины m.

Output: маркировка всех остальных узлов дерева строками длины m, которая минимизирует показатель экономии дерева.

Для дерева T, у которого каждый узел v помечен строкой длины m, определим помеченные деревья $T_1, \ldots, T_m$, где T_i имеет ту же структуру, что и T, и где данный узел помечен i-м признаком соответствующего узла в T. Поскольку пока-

затель экономии T является суммой показателей экономии деревьев $T_1, \ldots, T_m$, задача минимального показателя экономии может быть решена независимо для каждого столбца выравнивания. Это наблюдение позволяет предположить, что каждый лист помечен одним признаком, а не строкой. Таким образом, вес ребра, соединяющего два узла, должен быть либо 0, либо 1, в зависимости от того, помечены ли эти узлы одним и тем же признаком или разными признаками; для заданных признаков i и j мы определяем $\alpha_{i,j} = 0$, если $i = j$, и $\alpha_{i,j} = 1$, если $i \neq j$.

Мы опишем алгоритм динамического программирования, названный **SmallParsimony**, для решения «однопризнаковой» версии задачи минимального показателя экономии. Напомним, что корневое дерево T можно рассматривать как ориентированное дерево, все ребра которого направлены от корня к листьям. Таким образом, каждый узел v в T определяет поддерево T_v, образованное узлами «ниже» v и состоящее из всех узлов, до которых можно добраться, двигаясь вниз от v (рисунок ниже).

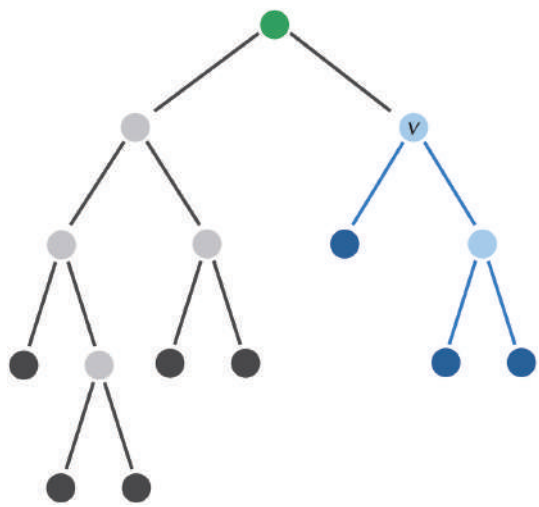


Рис. 7.36 Поддерево T_v (показано синим) узла v в более крупном корневом двоичном дереве T

Пусть k – признак в заданном алфавите, а v – узел в дереве T. Определим $s_k(v)$ как минимальный показатель экономии поддерева T_v по всем возможным меткам узлов T_v таких, что v помечен k. Начальные условия для **SmallParsimony** должны присваивать счета листьям. Если лист v помечен признаком k, то единственный признак, который мы можем присвоить этому листу, – это k. Следовательно, $s_k(v) = 0$, если лист v помечен признаком k, и $s_k(v) = \infty$ в противном случае (рисунок ниже).

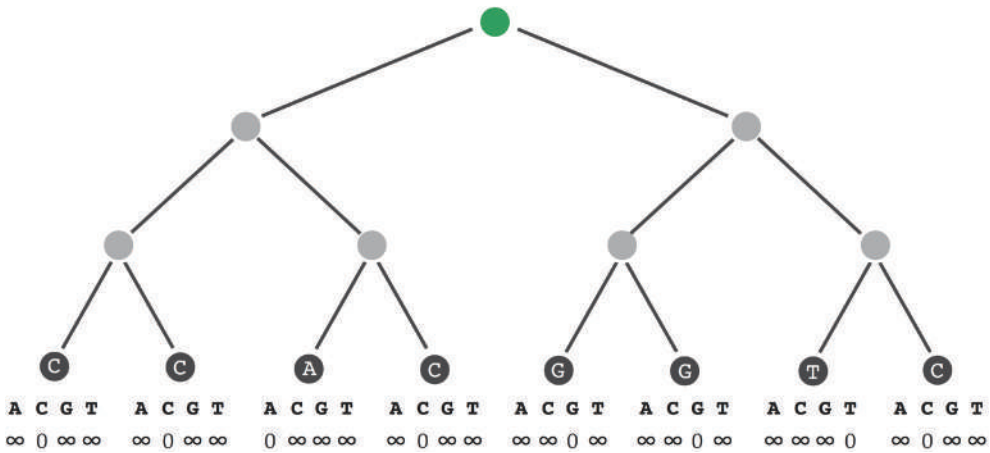


Рис. 7.37 Инициализация значений $s_k(v)$ для всех листьев v, представленных выше, в виде массива для каждого листа. Положим $s_k(v)$ равным нулю, если лист помечен признаком k; в противном случае положим $s_k(v)$ равным бесконечности

Затем v соединяется с двумя «дочерними» узлами (узлами ниже v в T), которые мы условно обозначим как $Daughter(v)$ и $Son(v)$. Счет $s_k(v)$ может быть вычислен как минимум $s_i(Daughter(v)) + \alpha_{i,k}$ по всем возможным признакам i плюс минимум $s_j(Son(v)) + \alpha_{j,k}$ по всем возможным признакам j:

$$s_k(v) = \min_{\text{all symbols } i}\{s_i(Daughter(v)) + \alpha_{i,k}\} + \min_{\text{all symbols } j}\{s_j(Son(v)) + \alpha_{j,k}\}.$$

Это уравнение позволяет нам вычислить все значения $s_k(v)$, прокладывая путь вверх по T от листьев к корню, обозначаемому как корень (рис. 7.38). Поддерево корня – это все дерево T, поэтому минимальный показатель экономии определяется наименьшей счетом $s_k(root)$ по всем признакам k,

$$\min_{\text{all symbols } k} s_k(root).$$

Как только мы вычислим показатель экономии дерева T, нам также понадобится какой-то способ присвоить признаки внутренним узлам T. Минимальное значение $s_k(root)$ на рис. 7.39 равно 3, что достигается при k = C. Присвоение признаков оставшимся внутренним узлам аналогичен подходу с возвратом, который мы использовали для выравнивания последовательностей. Поскольку вы уже являетесь профессионалом в области динамического программирования, мы предоставляем вам эту задачу в качестве упражнения.

Упражнение. Обозначьте нуклеотиды для всех узлов дерева на рис. 7.38, чтобы решить задачу минимального показателя экономии.

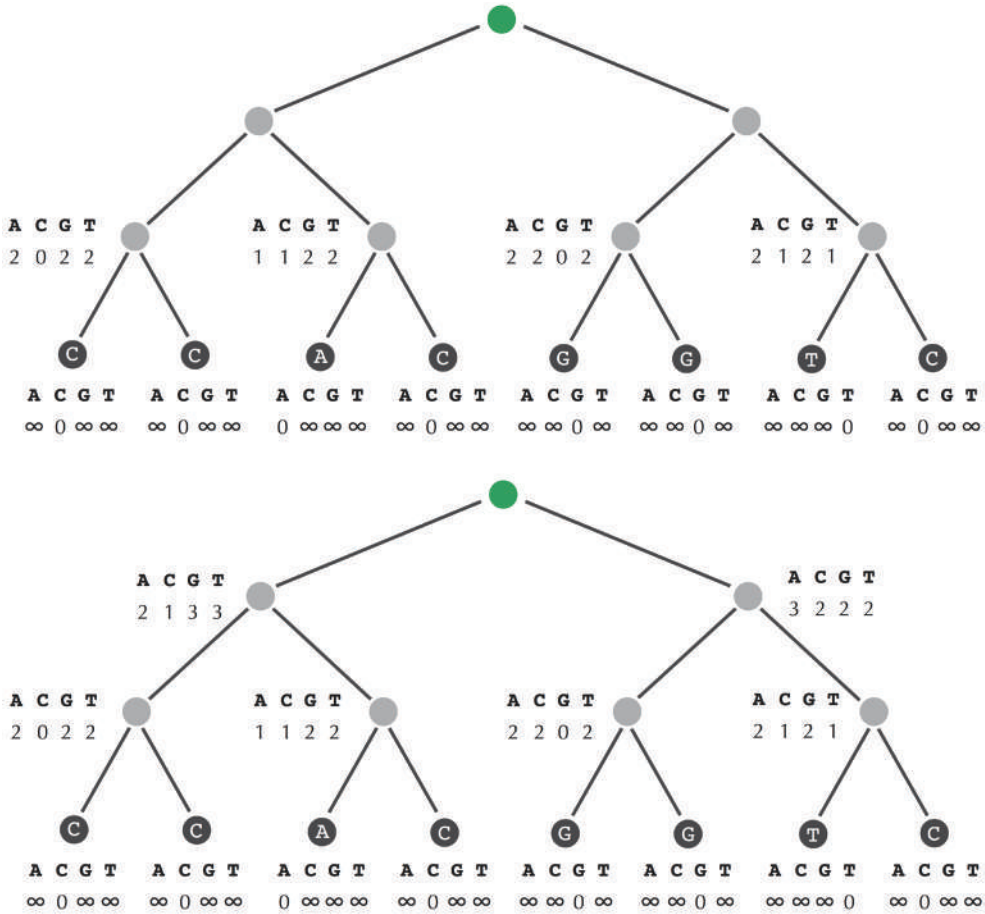


Рис. 7.38 Иллюстрация работы **SmallParsimony** после инициализации. Показатель экономии равен минимальному счету в корне, который для этого дерева равен 3. Это значение соответствует признаку C, поэтому, когда мы начинаем возвращаться, чтобы присвоить символы внутренним узлам, мы приписываем нуклеотид C корню

Ниже представлен псевдокод для **SmallParsimony**. Он выдает показатель экономии для бинарного корневого дерева T, листья которого помечены признаками, хранящимися в массиве *Character* (т. е. *Character*(v) – это метка листа v). На каждой итерации он выбирает узел v и вычисляет $s_k(v)$ для каждого признака k в алфавите. Для каждого узла v **SmallParsimony** поддерживает значение *Tag*(v), которое указывает, был ли обработан узел (т. е. *Tag*(v) = 1, если массив $s_k(v)$ был вычислен, и *Tag*(v) = 0 в противном случае). Мы называем внутренний узел T **созревшим**, если его тег равен 0, но оба его дочерних тега равны 1. **SmallParsimony** работает вверх от листьев, находя созревший узел v, в котором можно вычислить $s_k(v)$ на каждом шаге.

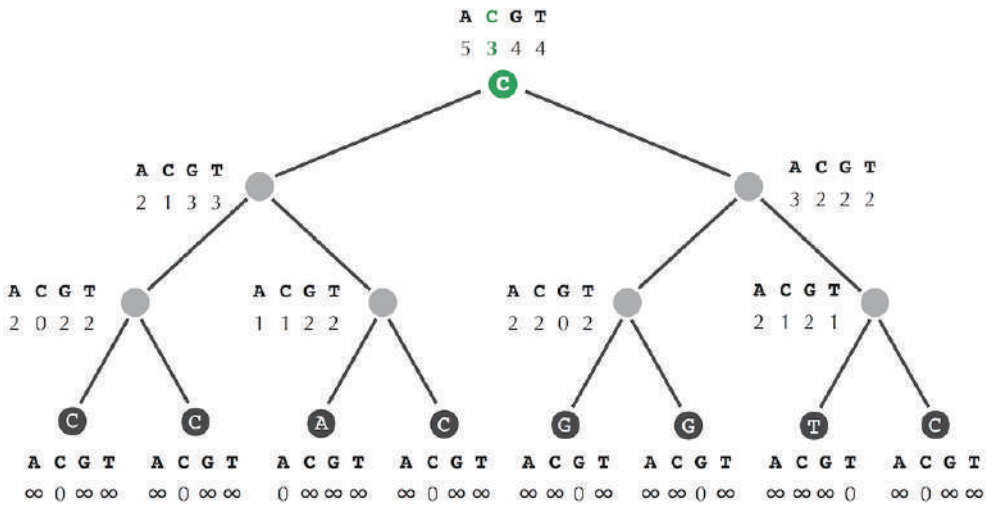


Рис. 7.39 Дерево для упражнения выше

```
SmallParsimony(T, Character)
  for каждого узла υ в дереве T
    Tag(υ) ← 0
    if υ является листом
      Tag(υ) ← 1
      for каждого символа k в алфавите
        if Character(υ) = k
          s_k(υ) ← 0
        else
          s_k(υ) ← ∞
  while в T существуют созревшие узлы
    υ ← созревший узел в T
    Tag(υ) ← 1
    for каждого символа k в алфавите
      s_k(υ) ← minimum_{all symbols i} {s_i(Daughter(υ)) + α_{i,k} } + minimum_{all symbols j}{s_j(Son(υ))
      + α_{i,k}}
  return minimum по всем символам k {s_k(υ)}
```

ОСТАНОВИТЕСЬ и задумайтесь. Какой финальный созревший узел обрабатывается **SmallParsimony** независимо от порядка, в котором мы обрабатываем созревшие узлы?

Когда положение корня в дереве неизвестно, мы можем просто присвоить корень любому ребру, которое нам нравится, применить **SmallParsimony** к ре-

зультирующему корневому дереву, а затем удалить корень. Можно показать, что этот метод позволяет решить следующую задачу.

Задача минимального показателя экономии в некорневом дереве: *найти наиболее экономную маркировку внутренних узлов в некорневом дереве.*

Input: бинарное некорневое дерево, в котором каждый лист помечен строкой длины m.

Output: маркировка всех остальных узлов дерева строками длины m, которая минимизирует показатель экономии дерева.

Задача максимальной экономии

SmallParsimony не поможет нам, если мы заранее не знаем структуры эволюционного дерева. В этом случае мы должны найти дерево, а также приписать строки предков всем внутренним узлам этого дерева, чтобы минимизировать показатель экономии. На рис. 7.40 показано решение задачи «минимального показателя экономии в некорневом дереве» для одного дерева с четырьмя листьями, где листьям приписаны строки из выравнивания млекопитающего, воспроизведенного ниже.

Задача максимальной экономии: *при заданном наборе строк найдите дерево (с листьями, помеченными всеми этими строками), имеющее минимальный показатель экономии.*

Input: набор строк одинаковой длины.

Output: некорневое двоичное дерево T, которое минимизирует показатель экономии среди всех возможных некорневых двоичных деревьев с листьями, помеченными этими строками.

К сожалению, оказывается, что задача максимальной экономии является *NP*-полной отчасти потому, что количество различных деревьев растет очень быстро по сравнению с количеством листьев. В качестве обходного пути мы будем использовать жадную эвристику, которая исследует некоторые, но не все деревья. Во-первых, обратите внимание, что в любом некорневом бинарном дереве удаление внутреннего ребра вместе с двумя внутренними узлами, которые это ребро соединяет, приводит к четырем поддеревьям, которые мы

назовем *W*, *X*, *Y* и *Z* (рис. 7.40). Эти четыре поддерева можно объединить в дерево тремя различными способами, которые мы обозначаем как *WX|YZ*, *WY|XZ* и *WZ|XY*. Эти три дерева называются **ближайшими соседями**; операция **обмена ближайшими соседями** заменяет дерево одним из его ближайших соседей (рис. 7.41).

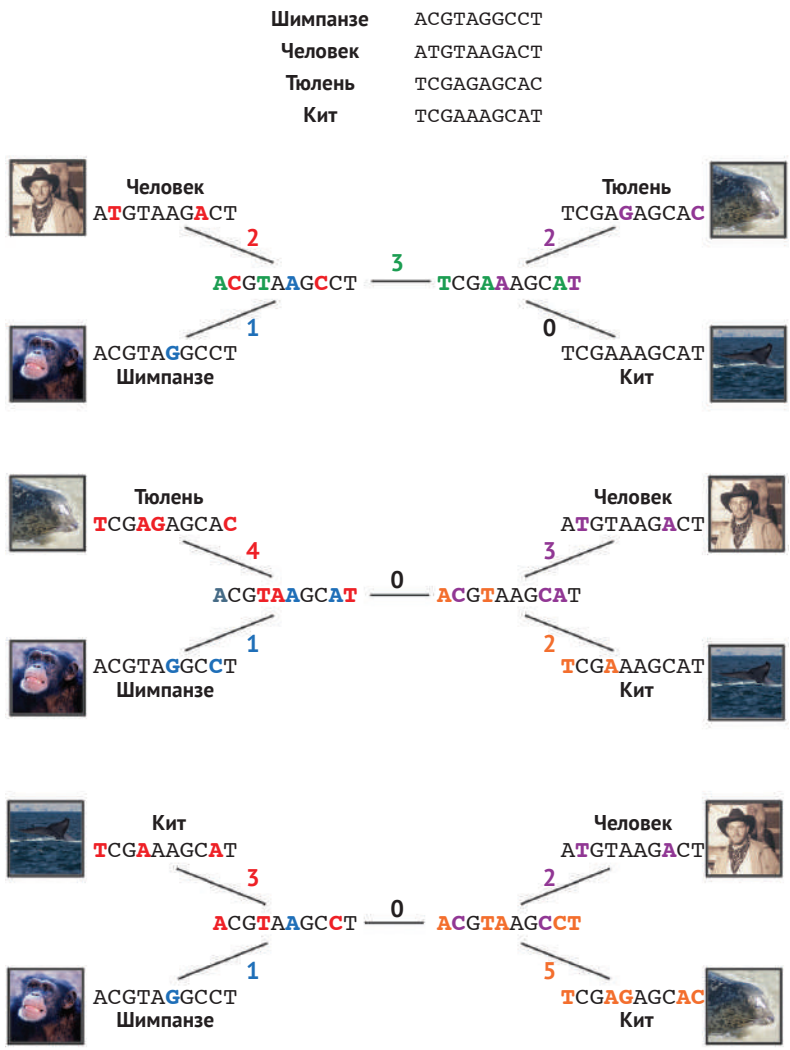


Рис. 7.40 Три структуры бинарного некорневого дерева для четырех видов в выравнивании с внутренними узлами, помеченными решением задачи «Малая экономия в задаче некорневого дерева». Первое дерево решает задачу максимальной экономии, потому что оно имеет меньший показатель экономии (8), чем два других дерева (по 11 каждое)

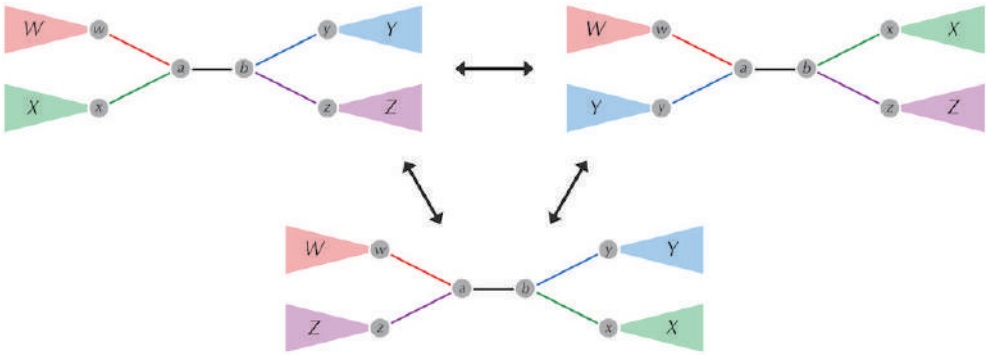


Рис. 7.41 Обмен ближайшими соседями на внутреннем ребре (*a*,*b*), показанный черным цветом, является результатом рекомбинации четырех цветных поддеревьев *W*, *X*, *Y* и *Z*, которые имеют корни в ω, *x*, *y* и *z* соответственно. Операция обмена ближайшими соседями удаляет одно ребро, соединенное с *a*, и другое ребро, соединенное с *b*, а затем заменяет эти ребра двумя новыми ребрами. Три возможные древовидные структуры, возникающие в результате обмена ближайшими соседями на (*a*, *b*), могут быть представлены как *WX*|*YZ* (вверху слева), *WY*|*XZ* (вверху справа) и *WZ*|*XY* (внизу)

Задача определения ближайших соседей дерева: *по ребру в бинарном дереве сгенерировать двух соседей этого дерева.*

Input: внутреннее ребро в бинарном дереве.

Output: два ближайших соседа этого дерева (для данного внутреннего ребра).

Подобно операции двойного разрыва, с которой мы столкнулись при изучении рекомбинаций генома, обмен ближайшими соседями соответствует замене двух ребер в дереве двумя новыми ребрами. Например, обозначим внутреннее реброобмена ближайшими соседями как (*a, b*); обозначим остальные узлы, смежные с *a*, как ω и *x*; а остальные узлы, смежные с *b*, обозначим как *y* и *z*. Дерево справа на рисунке выше получается из дерева слева удалением ребер (*a, x*) и (*b, y*) и заменой их на (*a, y*) и (*b, x*). Дерево внизу получается из дерева слева удалением ребер (*a, x*) и (*b, z*) и заменой их на (*a, z*) и (*b, x*).

Упражнение. Найдите двух ближайших соседей дерева на рисунке ниже для внутреннего ребра, соединяющего родителей гориллы и человека.

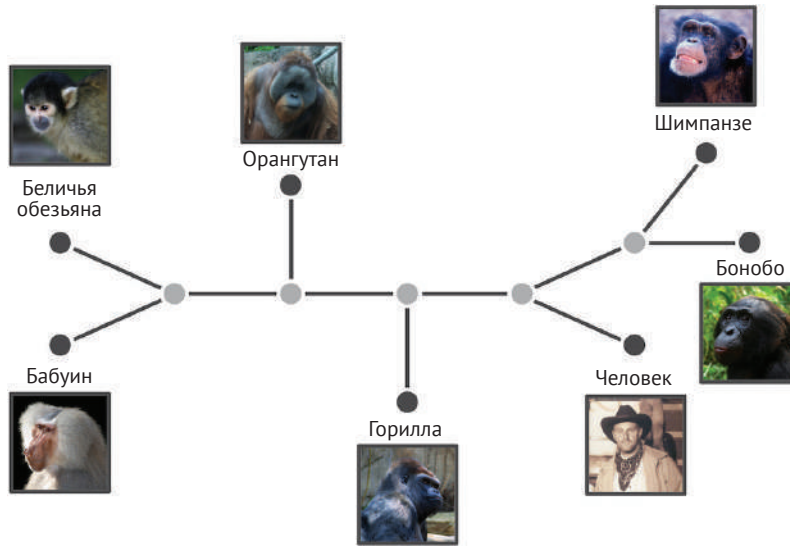


Рис. 7.42 Дерево, объединяющее родственников человека

Упражнение. На рис. 7.43 показаны все возможные некорневые бинарные деревья с пятью листьями. Найдите пару таких деревьев, которые «наиболее удалены друг от друга» в том смысле, что им требуется максимальное количество обменов между ближайшими соседями для преобразования одного дерева в другое.

Эвристика обмена ближайшими соседями для проблемы максимальной экономии начинается с произвольного некорневого двоичного дерева. Она назначает входные строки произвольным листьям этого дерева, назначает строки внутренним узлам дерева, решая задачу минимального показателя экономии в некорневом дереве, а затем перемещается к ближайшему соседу, который обеспечивает наилучшее улучшение счета экономии. На каждой итерации алгоритм исследует все внутренние ребра дерева и генерирует все обмены ближайшими соседями для каждого внутреннего ребра. Для каждого из этих ближайших соседей алгоритм решает задачу минимального показателя экономии, чтобы реконструировать метки внутренних узлов и вычислить показатель экономии. Если найден ближайший сосед с меньшим счетом экономии, то алгоритм выбирает соседа с наименьшим показателем экономии (ничьи разрываются произвольно) и повторяется снова; в противном случае алгоритм завершается. Это достигается с помощью следующего псевдокода.

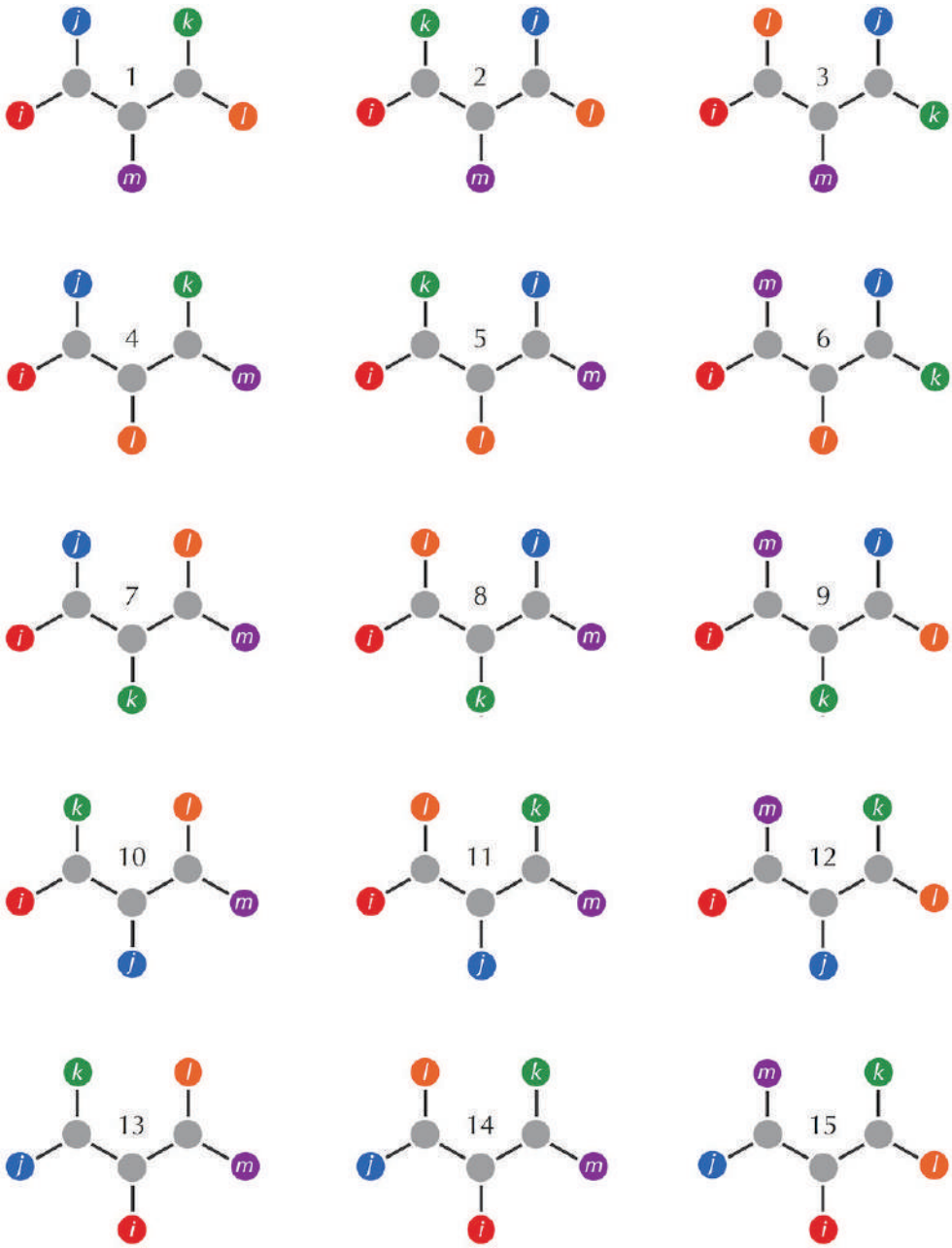


Рис. 7.43 Пятнадцать некорневых бинарных деревьев с пятью помеченными листьями. Дерево 1 может быть преобразовано в деревья 4, 7, 12 и 15 с помощью единственного обмена ближайшими соседями. Обратите внимание, что каждое дерево имеет одинаковую структуру; это не относится к деревьям, содержащим более пяти листьев

```
NearestNeighborInterchange(Strings)
    score ← ∞
    сгенерировать произвольное некорневое бинарное дерево Tree со |Strings|
    листьями
    пометить листья Tree произвольными строками из Strings
    решить задачу минимального показателя экономии в задаче некорневых
    деревьев для Tree
    пометить внутренние узлы Tree в соответствии с наиболее экономной
    маркировкой
    newScore ← показатель экономии дерева Tree
    newTree ← Tree
    while newScore < score
        score ← newScore
        Tree ← newTree
        for для каждого внутреннего ребра e в Tree
            for для каждого ближайшего соседа NeighborTree дерева Tree with
            относительно ребра e
                решить задачу минимального показателя экономии в задаче
                некорневых деревьев для NeighborTree
                neighborScore ← минимальный показатель экономи NeighborTree
                if neighborScore < newScore
                    newScore ← neighborScore
                    newTree ← NeighborTree
    return newTree
```

Упражнение. В следующей главе будет описано, как ген алкогольдегидрогеназы (Adh) помогает дрожжам производить алкоголь. Чтобы изучить эволюцию этого гена, биологи построили множественное выравнивание генов Adh различных видов дрожжей. Используйте следующее выравнивание, чтобы реконструировать эволюционное дерево различных видов дрожжей и ген предка древних дрожжей Adh.

☞ **Загрузить данные 7.2**

В этой главе мы представили ряд алгоритмов построения эволюционных деревьев, но это не означает, что мы можем легко разрешать различные эволюционные противоречия. Например, вопрос, кто есть ближайшей родственник шимпанзе, оставался нерешенной до середины 1990-х годов, а вопрос о том, ближе ли мыши к людям, чем к собакам, до сих пор является предметом споров (см. рис. 7.44, 7.45).

Рис. 7.44 (Вверху) Анализ генов бета-глобина у человека, шимпанзе и гориллы предполагает расхожение человека и шимпанзе. (Внизу) Анализ гена рецептора дофамина D4 предполагает расхожение гориллы и шимпанзе. Изображение предоставлено: Франс де Вааль (шимпанзе), Кабир Баки (горилла)

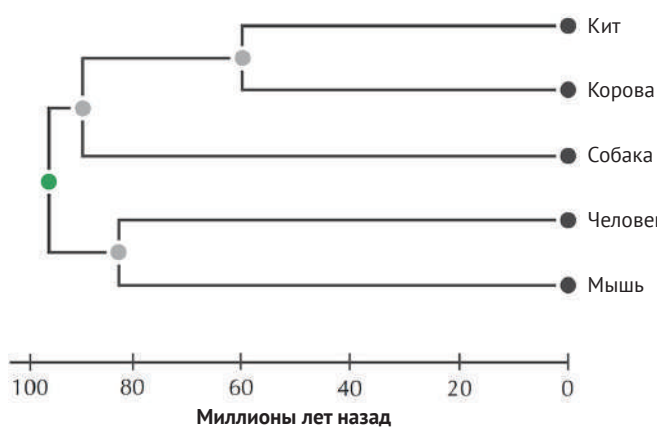


Рис. 7.45 Всего 15 лет назад биологи считали, что собаки эволюционно ближе к людям, чем мыши. Однако недавние исследования свидетельствуют об обратном, как показано в приведенной выше филогении

Эпилог. Эволюционные деревья в борьбе с преступностью

Дженис Трэхан познакомилась с доктором Ричардом Шмидтом в 1982 году, когда начала работать медсестрой в Лафайете, штат Луизиана. И Дженис, и Ричард были в браке и имели детей, но влюбились друг в друга. Дженис вскоре развелась с мужем, но, хотя Ричард обещал, что разведется с женой, так и не сделал этого. Через 12 лет она устала ждать и разорвала отношения. Две недели спустя она проснулась среди ночи и увидела стоящего над ней Ричарда со шприцем в руке.

Хотя Дженис рассталась с Ричардом, она не удивилась, увидев его. Она даже оставила свою дверь незапертой для него, потому что он делал ей инъекции витамина B-12 от ее хронической усталости. Поскольку Дженис раньше делала инъекции B-12, она знала, чего ожидать. Однако на этот раз почувствовала жгучую боль, когда Ричард сжал шприц.

Несколько месяцев спустя у Дженис выявили положительный результат на ВИЧ, и она обвинила Ричарда в том, что он заразил ее через инъекцию. Полицейский детектив в Лафайете никогда не слышал ничего более странного, чем эта история о мести, – шприц, зараженный ВИЧ, никогда не использовался в качестве орудия убийства. Сначала он подозревал, что Дженис сфабриковала эту историю, чтобы запятнать репутацию своего бывшего любовника. Тем не менее он начал расследование, собрав образцы у нескольких ВИЧ-инфицированных в Лафайете.

Изучив больничные записи, детектив обнаружил, что Ричард взял кровь у Дональда Макклелланда, больного ВИЧ, в тот же день, когда он сделал Дженис инъекцию. Теперь задача судебно-медицинской экспертизы заключалась в том, чтобы определить, был ли штамм ВИЧ, взятый у Макклелланда, сходен со штаммом Дженис. После того как ДНК ВИЧ была взята у Дженис, Макклелланд и многих других неродственных ВИЧ-инфицированных пациентов из Лафайета, ученые построили эволюционное дерево этих вирусов ВИЧ и обнаружили, что вирусы, взятые у Дженис и Макклелланд, сформировали поддерево этого дерева (рис. 7.46).

Дело «Штат Луизиана против Ричарда Шмидта» было передано в суд в 1998 году. Известный биолог-эволюционист Дэвид Хиллис представил эволюционное древо как доказательство преступления, продемонстрировав, что последовательность HIV Дженис была получена (с некоторыми небольшими вариациями) из HIV-последовательности Макклелланда. Ричард Шмидт был приговорен к 50 годам тюремного заключения за покушение на убийство.

ОСТАНОВИТЕСЬ и задумайтесь. Если бы вы были адвокатом Ричарда Шмидта, как бы вы доказывали его невиновность?

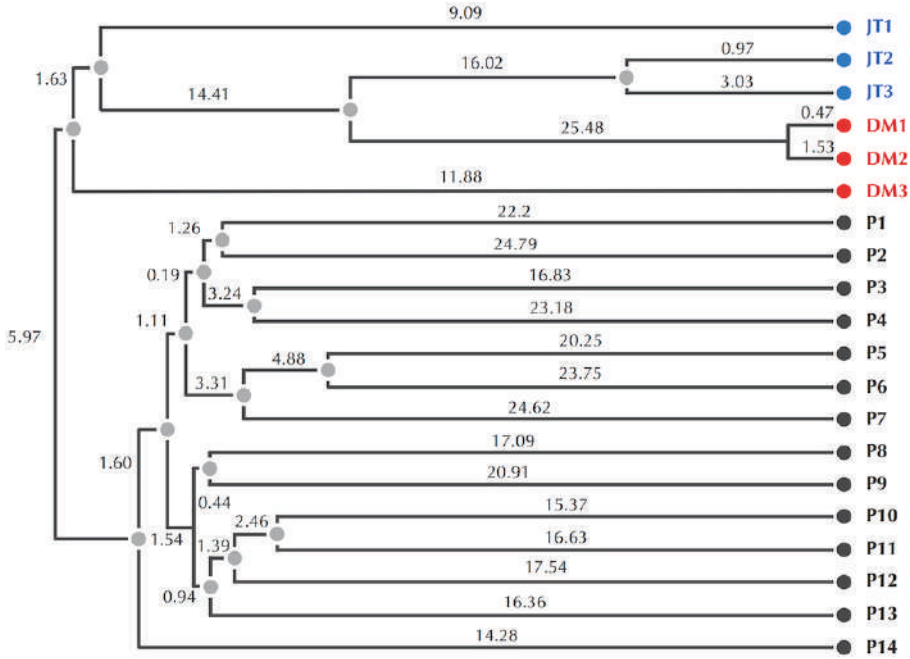


Рис. 7.46 Эволюционное дерево вирусов HIV, взятое у разных пациентов в Лафайете. Образцы Дженис Трэхан (синие листы JT1, JT2 и JT3) и Дональда Макклелланда (красные листы DM1, DM2 и DM3) сгруппированы вместе и довольно сильно отличаются от ДНК-последовательностей других пациентов из Лафайета (обозначены от P1 до P14)

Заключительная задача. Построить эволюционное дерево для другого белка HIV. Поддерживает ли это дерево обвинение доктора Шмидта? Восстановите наследственные последовательности HIV во внутренних узлах полученных деревьев.

☞ **Загрузить данные 7.3**

Сопутствующие материалы

Когда HIV перешел от приматов к человеку?

Ученым стало известно, что HIV вызывал СПИД в 1980-х годах, когда этот вирус был еще редкостью. Они сразу же начали искать более ранние случаи ВИЧ-инфекции в различных медицинских записях и обнаружили ВИЧ в образце крови, взятом у конголезского пациента в 1959 году. Затем генетические исследования показали, что HIV тесно связан с вирусом иммунодефицита обе-

зьян (SIV), который поражает приматов, но оставалось неясным, как и когда SIV проник в человеческую популяцию и превратился в HIV. Самая популярная современная гипотеза состоит в том, что SIV превратился в HIV, когда охотники убивали обезьян, чтобы продать их мясо, и имели контакт с кровью животных. Вирус, присутствовавший в крови обезьян, проникнув в порезы на коже охотников, мутировал, а позже адаптировался к человеку.

Находя области в вирусном геноме, которые мутируют примерно с постоянной скоростью, исследователи могут вывести временную шкалу эволюции HIV. Используя эти «молекулярные часы», биологи оценили моменты времени, когда различные подтипы SIV перескакивали от приматов к человеку: для групп HIV A, B, M и O эти переходы были оценены в 1940, 1945, 1908 и 1920 годах соответственно (время прыжка для группы N на данный момент неизвестно). Секвенировав образцы фекалий диких приматов, биологи даже смогли определить местонахождение популяций шимпанзе и черных мангабеев, чьи SIV являются прямыми предками групп HIV.

Поиск дерева с помощью настройки матрицы расстояний

Любая матрица D размерностью 3×3 является аддитивной. Чтобы понять почему, обратите внимание, что есть только одно дерево с тремя листьями. Обозначим листья этого дерева как 1, 2 и 3, а его внутренний узел – как c. Как показано на рисунке ниже, длины ребер в этом дереве должны удовлетворять следующим трем уравнениям:

$$d_{1,c} + d_{2,c} = D_{1,2} \quad d_{1,c} + d_{3,c} = D_{1,3} \quad d_{2,c} + d_{3,c} = D_{2,3}.$$

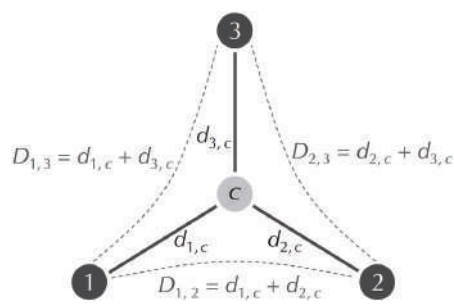


Рис. 7.47 Дерево с тремя листьями 1, 2 и 3 в дополнение к внутреннему узлу c. Расстояния между листьями ($D_{1,2}$, $D_{1,3}$ и $D_{2,3}$) однозначно определяют длины ребер ($d_{1,c}$, $d_{2,c}$ и $d_{3,c}$)

Решение этой системы уравнений дает следующие формулы для длин ребер через значения матрицы D:

$$D_{1,c} = (D_{1,2} + D_{1,3} - D_{2,3})/2$$
$$D_{2,c} = (D_{2,1} + D_{2,3} - D_{1,3})/2$$
$$d_{3,c} = (D_{3,1} + D_{3,2} - D_{1,2})/2.$$

На рисунке ниже показана попытка подогнать матрицу расстояний ко всем возможным бескорневым деревьям с четырьмя листьями. Каждое такое дерево приводит к системе из шести линейных уравнений (с четырьмя или пятью переменными), не имеющей решения. Таким образом, эта матрица расстояний должна быть неаддитивной.

	v_1	v_2	v_3	v_4
v_1	0	3	4	3
v_2	3	0	4	5
v_3	4	4	0	2
v_4	3	5	2	0

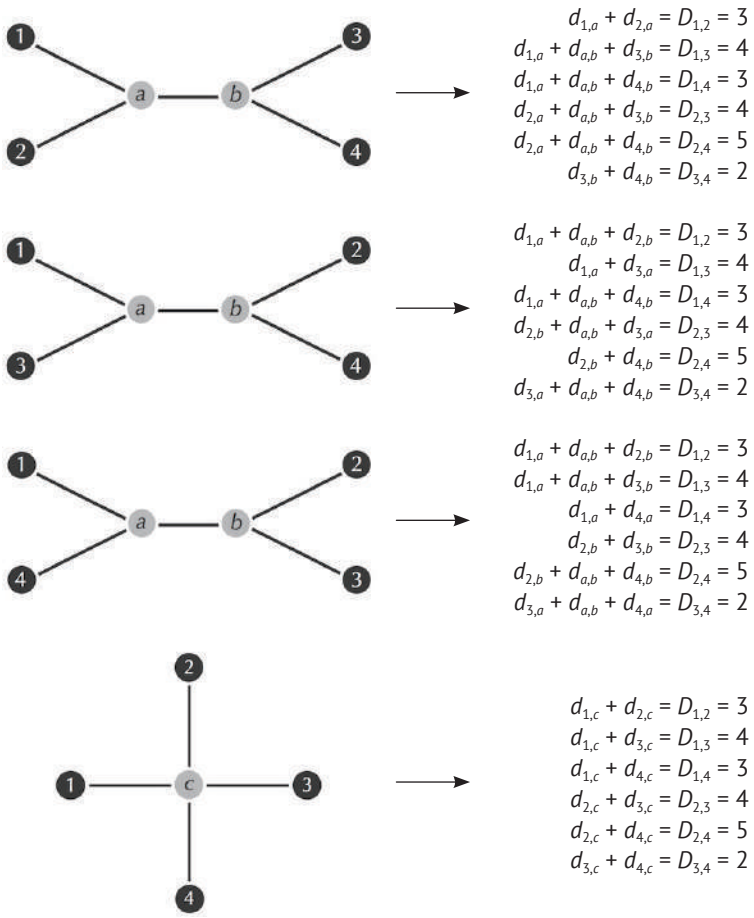


Рис. 7.48 (Вверху) Матрица расстояний D размерностью 4×4. (Слева) Все четыре дерева с четырьмя листьями (обозначены цифрами 1, 2, 3, 4). (Справа) Каждая попытка подогнать D к дереву приводит к системе шести линейных уравнений. Поскольку ни одна из этих систем не имеет решения, D должна быть неаддитивной

Одна из причин, по которой нам не удалось подогнать дерево к матрице расстояний 4×4, состоит в том, что количество уравнений было больше, чем количество переменных ($d_{i,j}$) для каждого дерева (факт, который также будет иметь место для больших значения n). Если количество уравнений в линейной системе уравнений больше или равно количеству переменных, то решение обычно существует, тогда как, если количество уравнений меньше количества переменных, решения обычно нет. Однако есть исключения в обе стороны. Для получения более подробной информации обратитесь к учебнику по линейной алгебре.

Условие четырех точек

Условие четырех точек дает альтернативный способ определить, является ли матрица аддитивной. Рассмотрим дерево, содержащее только четыре листа (рис. 7.49). Для этого дерева заметим, что

$$d_{i,j} + d_{k,l} \leq d_{i,k} + d_{j,l} = d_{i,l} + d_{j,k},$$

потому что первая сумма – это сумма длин всех ребер в дереве *минус* длина внутреннего ребра, а последние две суммы равны сумме длин всех ребер в дереве *плюс* длина внутреннего ребра.

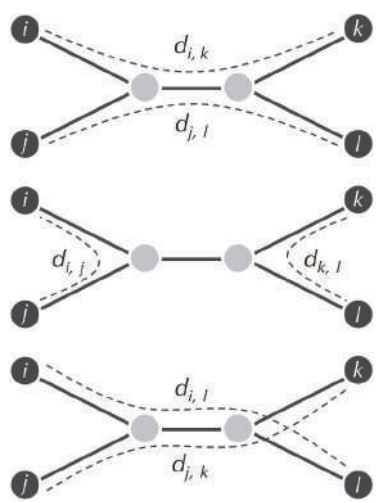


Рис. 7.49 Три пары путей через дерево с четырьмя листьями. Пути вверху слева и вверху справа проходят через одни и те же ребра, поэтому $d_{i,k} + d_{j,l} = d_{i,l} + d_{j,k}$. Кроме того, $d_{i,j} + d_{k,l}$ должно быть меньше или равно этим двум суммам, поскольку оно не пересекает внутреннее ребро дерева, как показано на дереве внизу

Действительно, для любой четверки листьев (*i, j, k, l*) произвольного дерева, если мы вычислим три суммы

$$d_{i,j} + d_{k,l} \quad d_{i,k} + d_{j,l} \quad d_{i,l} + d_{j,k},$$

мы обнаружим, что две суммы равны, а третья сумма меньше или равна двум другим суммам. В терминах матрицы расстояний $n \times n$ мы говорим, что четверка индексов (i, j, k, l) удовлетворяет **условию четырех точек**, если две из следующих сумм равны, а третья меньше или равна двум другим суммам:

$$D_{i,j} + D_{k,l} \quad D_{i,k} + D_{j,l} \quad D_{i,l} + D_{j,k}.$$

Теорема четырех точек. *Матрица расстояний является аддитивной тогда и только тогда, когда условие четырех точек выполняется для каждой четверки индексов (i, j, k, l) этой матрицы.*

Упражнение. Докажите теорему четырех точек.

Теорема четырех точек дает нам альтернативный способ определения того, является ли данная матрица расстояний аддитивной, поскольку мы можем просто проверить, выполняется ли условие четырех точек для каждой четверки индексов матрицы.

Упражнение. Сравните время выполнения этого предложенного метода со временем выполнения варианта **AdditivePhylogeny**, определяющего, является ли данная матрица расстояний аддитивной.

Упражнение. Найдите четверку индексов из матрицы расстояний на рисунке ниже, которая нарушает условие четырех точек.

	v_1	v_2	v_3	v_4
v_1	0	3	4	3
v_2	3	0	4	5
v_3	4	4	0	2
v_4	3	5	2	0

Рис. 7.50 Матрица расстояний

Заразили ли нас атипичной пневмонией летучие мыши?

Во время поиска животного резервуара вируса атипичной пневмонии биологи обнаружили зараженных пальмовых цивет на рынке живых животных в Китае. Мясо этих животных часто добавляют в дорогое кантонское блюдо «суп из дракона, тигра и феникса». Это открытие не сильно изменило судьбу инфи-

цированных цивет: вместо того, чтобы оказаться в супе, их превратили в козлов отпущения атипичной пневмонии и убили. Тем не менее, когда дальнейшие поиски не выявили больше зараженных атипичной пневмонией цивет, биологи начали задаваться вопросом, действительно ли пальмовые циветы были первоначальным источником атипичной пневмонии. В 2005 году они обнаружили SARS-подобный вирус у китайских подковоносов (рисунок ниже). Летучие мыши оказались переносчиками SARS-CoV, но они, вероятно, могут передавать вирус людям только через промежуточных хозяев. Поскольку мясо летучих мышей считается деликатесом и также используется в традиционной китайской медицине, у летучих мышей было много шансов вступить в тесный контакт с циветами на переполненных рынках живых животных.

Рис. 7.51 Подковообразная летучая мышь.
Фото предоставлено Praveenp, пользователем Wikimedia Commons

Когда биологи построили эволюционное древо коронавирусов летучих мышей, цивет и человека (рис. 7.52), они обнаружили, что как вирус цивет, так и человеческий варианты SARS-CoV вложены в филогенез вируса летучих мышей.

Еще до секвенирования SARS-CoV биологи знали о других коронавирусах человека, но считали их безвредными. Однако в 2012 году в Саудовской Аравии появился новый смертоносный коронавирус, похожий на атипичную пневмонию. Этот вирус (вызывающий **ближневосточный респираторный синдром (MERS)**) попал в главные мировые новости, поскольку он быстро распространился на другие страны.

Хотя исследователи изначально полагали, что в MERS виноваты верблюды, – многие жители Саудовской Аравии потребляют непастеризованное верблюжье молоко, – большинство больных не контактировало с верблюдами. Тем не менее, когда исследователи проверили коронавирус, взятый у летучей мыши, найденной всего в нескольких милях от дома одного из первых пациентов с MERS, они обнаружили, что этот вирус почти полностью совпадает с образцом вируса пациента.

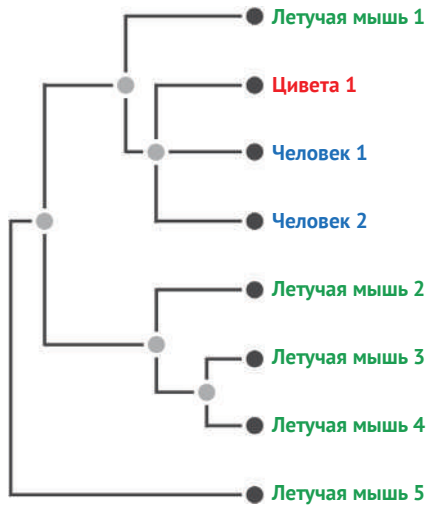


Рис. 7.52 Эволюционное дерево коронавирусов летучих мышей, цивет и человека

Почему алгоритм объединения соседей работает?

В основном тексте мы утверждали, что если матрица расстояний D аддитивна, то существует единственное простое дерево $Tree(D)$, соответствующее этой матрице. Это не совсем так, так как если простое дерево подходит под D и имеет внутреннее ребро нулевого веса, то мы легко можем удалить это ребро, «склеив» между собой узлы, которые оно соединяет (рисунок ниже). Таким образом, мы сделаем еще одно предположение, что $Tree(D)$ не только простое, но и не содержит внутренних ребер нулевой длины.

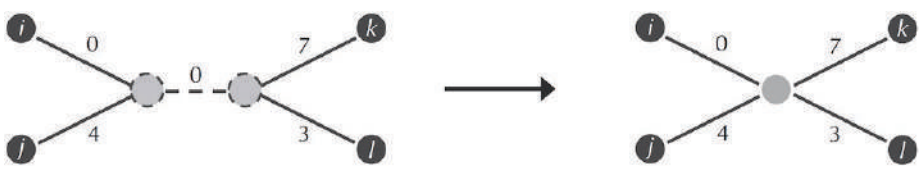


Рис. 7.53 Склеивание узлов на концах внутреннего ребра нулевой длины (показано пунктиром)

Теперь мы можем переписать формулу матрицы объединения соседей следующим образом:

$$D^{*}_{i,j} = (n-2)\cdot D_{i,j} - TotalDistance_D(i) - TotalDistance_D(j) =$$
$$(n-2)\cdot D_{i,j} - \sum_{1\le k\le n} D_{i,k} - \sum_{1\le k\le n} D_{j,k}.$$

Каждый элемент в этой формуле далее разбивается на сумму весов ребер в дереве (D). Например, для первого дерева на рисунке ниже

$$D^*_{1,2} = 2 \cdot d_{1,2} - (d_{1,3} + d_{1,4} + d_{1,2}) - (d_{2,3} + d_{2,4} + d_{1,2}) =$$
$$2 \cdot (d_{1,a} + d_{a,2}) - ((d_{1,a} + d_{a,b} + d_{b,3}) + (d_{1,a} + d_{a,b} + d_{b,4}) + (d_{1,a} + d_{a,2})) -$$
$$((d_{2,a} + d_{a,b} + d_{b,3}) + (d_{2,a} + d_{a,b} + d_{b,4}) + (d_{1,a} + d_{a,2})) =$$
$$-2 \cdot d_{1,a} - 2 \cdot d_{2,a} - 2 \cdot d_{b,3} - 2 \cdot d_{b,4} - 4 \cdot d_{a,b},$$

а также

$$D^*_{1,3} = 2 \cdot d_{1,3} - (d_{1,2} + d_{1,4} + d_{1,3}) - (d_{3,4} + d_{3,2} + d_{3,1}) =$$
$$2 \cdot (d_{1,a} + d_{a,b} + d_{b,3}) - ((d_{1,a} + d_{a,2}) + (d_{1,a} + d_{a,b} + d_{b,4}) + (d_{1,a} + d_{a,b} + d_{b,3})) -$$
$$((d_{3,b} + d_{b,4}) + (d_{3,b} + d_{b,a} + d_{b,2}) + (d_{3,b} + d_{b,a} + d_{a,1})) =$$
$$-2 \cdot d_{1,a} - 2 \cdot d_{2,a} - 2 \cdot d_{b,3} - 2 \cdot d_{b,4} - 2 \cdot d_{a,b}.$$

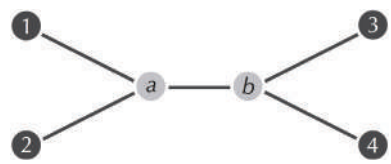


Рис. 7.54 Объединение соседей

Обратите внимание, что выражения для $D^*_{1,2}$ и $D^*_{1,3}$ почти идентичны, различаются только коэффициенты $d_{a,b}$ (выделены синим цветом выше). Поскольку $D^*_{1,2} - D_{1,3} = -2 \cdot d_{a,b} < 0$, алгоритм объединения с соседями предпочтет меньшее значение $D^*_{1,2}$, чем $D^*_{1,3}$.

Для данного ребра e в $Tree(D)$ его **кратность** в $D^*_{i,j}$, обозначаемая как $Multiplicity_{i,j}(e)$, является коэффициентом d_e в выражении для $D^*_{i,j}$. Следующий результат показывает, что все ветви дерева (D) имеют одинаковую кратность.

Лемма. Для аддитивной матрицы расстояний D и любой пары листьев i и j в $Tree(D)$ $Multiplicity_{i,j}(e)$ равна -2 для любой ветви e в $Tree(D)$.

Доказательство. Если ветвь e не является ветвью листа i или листа j, то она подсчитывается ноль раз в $(n - 2) \cdot D_{i,j}$, один раз в $TotalDistance_D(i)$ и один раз в $TotalDistance_D(j)$, что делает ее кратность равной -2. С другой стороны, если e является ветвью i или j (скажем, i), то она считает n дважды в $(n - 2) \cdot D_{i,j}$, $n - 1$ раз в $TotalDistance_D(i)$ и один раз в $TotalDistance_D(j)$. Следовательно, его кратность равна $n - 2 - (n - 1) - 1 = -2$. ■

Эта лемма подразумевает, что, независимо от того, какую пару листьев i и j мы выберем, все ветви $Tree(D)$ будут иметь одинаковый вклад в вычисление $D^*_{i,j}$. В результате только кратности внутренних ребер $Tree(D)$ дифференцируют значения матрицы D^*. Можем ли мы определить эти кратности?

Упражнение. Докажите, что для любых i и j и для любого внутреннего ребра e в $Tree(D)$ $Multiplicity_{i,j}(e) \leq -2$.

Мало того, что кратность каждого внутреннего ребра не превышает –2, у нас также есть условие, определяющее, когда кратность внутреннего ребра будет равна –2. Чтобы вывести это условие, сначала заметим, что удаление внутреннего ребра e разъединяет любое дерево на два поддерева. Если e лежит на (уникальном) пути, соединяющем листья i и j в $Tree(D)$, обозначаемом $Path(i, j)$, то i и j принадлежат разным поддеревьям, обозначаемым T_i и T_j соответственно; в противном случае i и j принадлежат одному и тому же поддереву (рисунок ниже). В последнем случае мы обозначаем количество листьев в поддереве, не включающем листья i и j, как $Leaves_{i,j}(e)$.

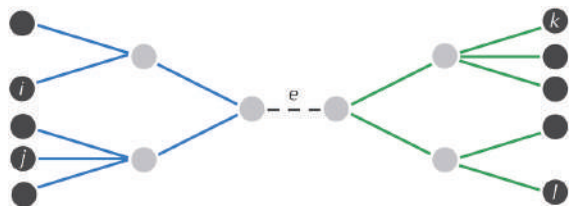


Рис. 7.55 Если внутреннее ребро e лежит на уникальном пути, соединяющем два листа в исходном дереве (например, j и k), то после удаления e эти листья разделяются на разные поддеревья (показаны синим и зеленым). Если e не лежит на единственном пути, соединяющем два листа (например, k и l), то после удаления e эти листья останутся принадлежащими к одному и тому же поддереву

Теорема о кратности ребер. Для аддитивной матрицы D кратность внутреннего ребра e равна –2, если e лежит на $Path(i, j)$ в $Tree(D)$, и равна $-2 \cdot Leaves_{i,j}(e)$ в противном случае.

Доказательство. Если внутреннее ребро e лежит на $Path(i, j)$ в $Tree(D)$, то e имеет коэффициент $n-2$ в члене $(n-2) \cdot D_{i,j}$ в $D^*_{i,j}$. Чтобы вычислить $Multiplicity_{i,j}(e)$, рассмотрим поддеревья T_i и T_j, образованные удалением e. Для каждого листа k в T_i $Path(j, k)$ проходит через e, таким образом внося 1 в коэффициент e в $TotalDistance_D(j)$, но $Path(i, k)$ не проходит через e, таким образом внося 0 в коэффициент e в $TotalDistance_D(i)$.

Аналогично для каждого листа k в T_j $Path(i, k)$ проходит через e, таким образом добавляя 1 к коэффициенту e в $TotalDistance_D(i)$, но $Path(j,k)$ не проходит через e, таким образом добавляя 0 к коэффициенту e в $TotalDistance_D(i)$. В результате каждый лист k внесет 1 в коэффициент e либо в $TotalDistance_D(i)$, либо в $TotalDistance_D(j)$. Таким образом, коэффициент e в $TotalDistance_D(i)$ + $TotalDistance_D(j)$ равен n, что означает, что

$$Multiplicity_{i,j}(e) = (n-2) - n = -2.$$

С другой стороны, если e не лежит на $Path(i, j)$, то e имеет коэффициент 0 в $(n-2)\cdot D_{i,j}$. А если k – лист в поддереве, не содержащий i и j, то, чтобы добраться до k, нужно пройти через e. Таким образом, коэффициент e в каждом из $TotalDistance_D(i)$ и $TotalDistance_D(j)$ равен $Leaves_{i,j}$. Следовательно,

$$Multiplicity(e) = 0 - 2\cdot Leaves_{i,j}(e) = -2\cdot Leaves_{i,j}\ (e).\ \blacksquare$$

Мы можем интерпретировать теорему о кратности ребер как утверждение, что внутренние ребра на пути $Path(i, j)$ имеют большие кратности (–2), а другие внутренние ребра имеют малые кратности (меньше –2). Таким образом, если мы пытаемся минимизировать $D^*_{i,j}$ (среди всех возможных вариантов i и j), то мы должны искать пару листьев (i, j), имеющих мало внутренних ребер на $Path(i, j)$. Соседи не имеют внутренних ребер, соединяющих их, что делает их привлекательными кандидатами. Следующее упражнение поможет нам частично доказать, что соседи действительно минимизируют $D^*_{i,j}$.

Упражнение. Покажите, что если листья i и j являются соседями дерева (D), а лист k не является соседом i, то $D^*_{i,j} < D^*_{i,k}$.

Теорема объединения соседей. При наличии аддитивной матрицы расстояний D минимальный элемент $D^*_{i,j}$ матрицы объединения соседей D^* соответствует соседним листьям i и j в $Tree(D)$.

Доказательство. Предположим, что $D^*_{i,j}$ – минимальный элемент D^*, но i и j не являются соседями в $Tree(D)$. Мы стремимся прийти к противоречию, найдя пару соседей k и l таких, что $D^*_{k,l} < D^*_{i,j}$. Согласно предыдущему упражнению ни i, ни j не могут иметь соседа, если $D^*_{i,j}$ – минимальный элемент D^*. Таким образом, i и j – единственные листья, связанные с $Parent(i)$ и $Parent(j)$ соответственно. Поскольку $Tree(D)$ простое, $Parent(i)$ и $Parent(j)$ имеют степень не менее 3, а это означает, что каждый из этих узлов соединен как минимум с двумя другими узлами $Tree(D)$, один из которых лежит на $Path(i, j)$. Другой узел является частью собственного поддерева; мы называем эти поддеревья T_1 и T_2 (рис. 7.56).

Без ограничения общности будем считать, что количество листьев в T_1 не превосходит количество листьев в T_2. Поскольку i и j не принадлежат T_1 или T_2, T_1 должно содержать менее $n/2$ листьев, а остальная часть $Tree(D)$ должна содержать более $n/2$ листьев (напомним, что n – это общее количество листьев в $Tree(D)$). Поскольку у i нет соседей, у T_1 должно быть не менее двух листьев, а это означает, что у T_1 есть пара соседей, которые мы обозначим как (k, l). Покажем, что $D^*_{k,l} < D^*_{i,j}$.

Рассмотрим внутреннее ребро e дерева $Tree(D)$. Сначала мы покажем, что $D^*_{k,l} \le D^*_{i,j}$, показав, что кратность e в $D^*_{k,l}$ не превышает кратности e в $D^*_{i,j}$. Есть три возможности.

- Если e лежит на $Path(i, j)$, то по теореме о кратности ребер $Multiplicity_{i,j}(e) = 2$, и отсюда следует результат.

- Если e лежит на $Path(i, k)$, то удаление e разбивает $Tree(D)$ на два поддерева, одно из которых содержит k и l (с количеством листьев, равным $Leaves_{i,j} < n/2$), а другое содержит i и j (с числом листьев, равным $Leaves_{k,l}(e) > n/2$). Таким образом, по теореме о кратности ребер $Multiplicity_{k,l}(e) = 2 \cdot Leaves_{k,l}(e) < Multiplicity_{i,j}(e) = 2 \cdot Leaves_{i,j}$.
- Если e не лежит ни на $Path(i, j)$, ни на $Path(i, k)$, то поддерево, не содержащее i и j, при удалении e совпадает с поддеревом, не содержащим k и l. Таким образом, $Leaves_{i,j}(e) = Leaves_{k,l}(e)$, что означает, что $Multiplicity_{i,j}(e) = Multiplicity_{k,j}(e)$ (теорема о кратности ребер).

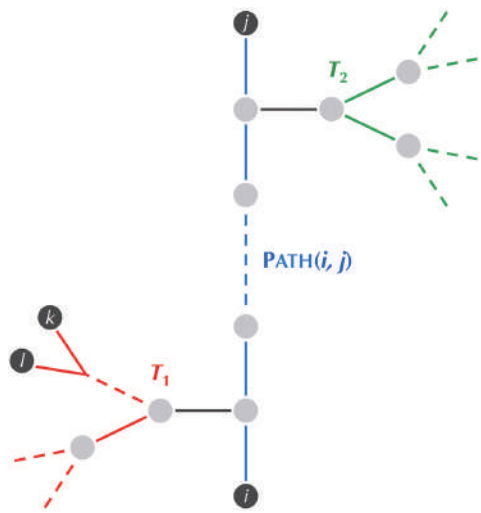


Рис. 7.56 Листья i и j не являются соседями. Уникальный путь, соединяющий их в $Tree(D)$, $Path(i, j)$, показан синим цветом. Поскольку $Tree(D)$ простое, $Parent(i)$ и $Parent(j)$ должны быть соединены как минимум с двумя другими внутренними узлами, образуя таким образом поддеревья T_1 и T_2 (показаны красным и зеленым)

Чтобы доказать, что $D^*_{k,l}$ на самом деле меньше, чем $D^*_{i,j}$, заметим, что $Path(i, k)$ должен содержать внутреннее ребро e, поскольку i и k не являются соседями. В приведенном выше среднем случае мы знаем, что $Multiplicity_{k,l}(e) < Multiplicity_{i,j}(e)$. Наше предположение, что никакие внутренние ребра $Tree(D)$ не имеют нулевой длины, означает, что d_e должно быть положительным, что и требовалось доказать. ■

Вычисление длин ветвей в алгоритме объединения соседей

В основном тексте мы пытались присвоить длины ветвей листьям дерева, построенного из произвольной матрицы расстояний. Положим длину ветви i равной $1/2(D_{i,j} + \Delta_{i,j})$, а длину ветви j равной $1/2(D_{i,j} - \Delta_{i,j})$, где

$$\Delta = \frac{TotalDistance_D(i) - TotalDistance_D(j)}{n-2}.$$

Откуда берутся эти формулы? Предположим на мгновение, что D – аддитивная матрица, и выберем некоторый лист k, не являющийся равным i или j. Если m является родителем i и j, то у нас уже есть формула длины ветви

$$LimbLength(i) = (1/2) \cdot (D_{i,j} + D_{i,k} - D_{j,k}).$$

В результате может показаться, что эту формулу следует использовать в алгоритме объединения соседей для произвольной матрицы расстояний D. Однако если матрица D неаддитивна, то выражение $(D_{i,j} + D_{i,k} - D_{j,k})/2$ будет варьироваться в зависимости от того, как мы выбираем k. Итак, нам нужна формула, которая по-прежнему вычисляет $LimbLength(i)$, когда матрица D является аддитивной, но также дает нам одно значение, когда D неаддитивна. С этой целью мы можем вычислить среднее значение приведенной выше формулы по всем выборкам матрицы из $n - 2$ листьев k:

$$\frac{1}{n-2} \sum_{\text{all leaves } k \text{ differing from } i \text{ and } j} \frac{D_{i,j} + D_{i,k} - D_{j,k}}{2}.$$

Если матрица D является аддитивной, то приведенная выше сумма содержит $n - 2$ члена, все из которых равны $LimbLength(i)$. Кроме того, если D неаддитивна, то эта формула дает нам оценку длины ветви. Заметим также, что приведенная выше сумма имеет $n - 2$ вхождения $D_{i,j}$. Если их выделить, то получим

$$\begin{aligned}
LimbLength(i) &= \frac{D_{i,j}}{2} + \frac{1}{n-2} \cdot \sum_{\text{all leaves } k \text{ differing from } i \text{ and } j} \frac{D_{i,k} - D_{j,k}}{2} \\
&= \frac{D_{i,j}}{2} + \frac{1}{n-2} \cdot \left(\sum_{\text{all leaves } k \text{ differing from } i \text{ and } j} \frac{D_{i,k}}{2} - \sum_{\text{all leaves } k \text{ differing from } i \text{ and } j} \frac{D_{j,k}}{2} \right) \\
&= \frac{1}{2} \cdot \left(D_{i,j} + \frac{1}{n-2} \cdot \left(\sum_{\text{all leaves } k \text{ differing from } i \text{ and } j} D_{i,k} - \sum_{\text{all leaves } k \text{ differing from } i \text{ and } j} D_{j,k} \right) \right) \\
&= \frac{1}{2} \cdot \left(D_{i,j} + \frac{\left(TotalDistance_D(i) - TotalDistance_D(j) \right)}{n-2} \right) \\
&= \frac{1}{2} \cdot (D_{i,j} + \Delta_{i,j}),
\end{aligned}$$

что и является формулой, приведенной в основном тексте, которую мы использовали для вычисления $LimbLength(i)$.

Большая панда: медведь или енот?

В течение многих лет биологи не могли прийти к единому мнению, следует ли классифицировать гигантскую панду как медведя или как енота. Хотя ги-

гантские панды внешне похожи на медведей, у них есть черты, необычные для медведей и типичные для енотов: зимой они не впадают в спячку, а их мужские гениталии крошечные и направлены назад. В результате Эдвин Колберт писал в 1938 году:

> *Таким образом, квест продолжается уже много лет: сторонники медведя, сторонники енота и группа тех, кто между ними, выдвигают свои аргументы с самой ясной логикой, в то время как гигантская панда безмятежно живет в горах Сычуаня, не задумываясь о зоологических противоречиях, которые она вызывает самим фактом своего существования.*

В то время как анализ анатомических и поведенческих признаков (таких как зимняя спячка) привел только к нерешенным спорам, анализ генетических признаков, проведенный Стивеном О'Брайеном в 1985 году, показал, что гигантские панды действительно более тесно связаны с медведями, чем с енотами (рисунок ниже).

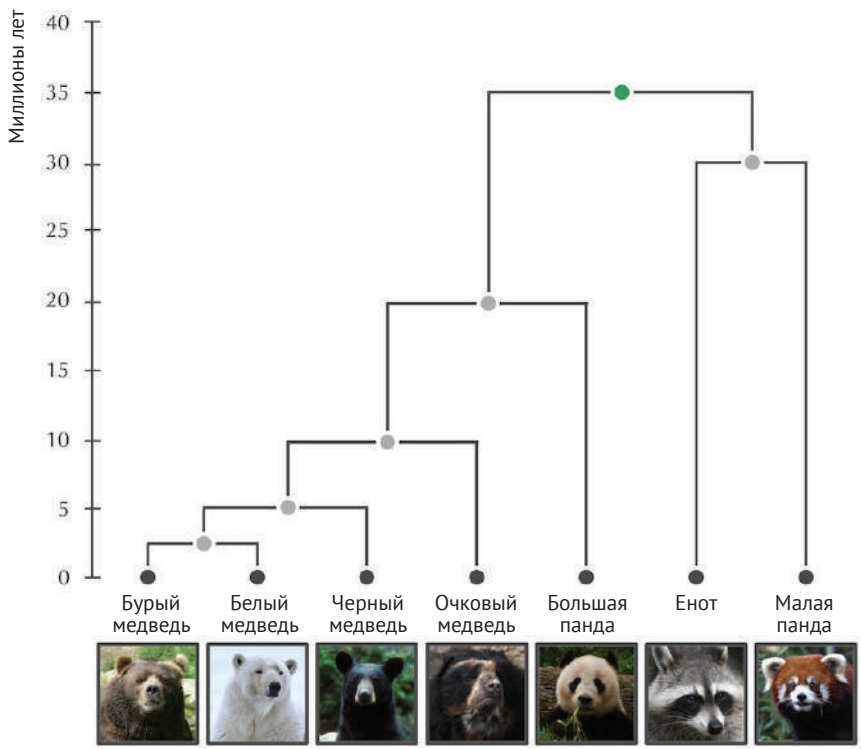


Рис. 7.57 Эволюционное дерево медведей и енотов. Изображения предоставлены: Алан Д. Уилсон (белый медведь), Марк Дюмон (медведь в очках), Питер Минен (красная панда), пользователь Викисклада Darkone (енот)

Откуда пришли люди?

В 1987 году Ребекка Канн, Марк Стоункинг и Аллан Уилсон построили эволюционное дерево **митохондриальной ДНК (мтДНК)** 133 человек, представляющих африканские, азиатские, австралийские, кавказские и новогвинейские этнические группы. Это дерево привело к гипотезе **«Из Африки»**, которая утверждает, что у людей есть общий предок, живший в Африке. Это исследование превратило вопрос о происхождении человека в алгоритмическую задачу.

Эволюционное дерево мтДНК показало ствол, разделяющийся на две основные ветви (рис. 7.58). Одна ветвь, содержащая пять нижних особей на рисунке ниже, состояла только из африканцев, тогда как другая ветвь включала некоторых современных африканцев, а также всех людей, принадлежащих к другим этническим группам.

ОСТАНОВИТЕСЬ и задумайтесь. Взглянув на дерево на рисунке ниже, какой вывод вы сделаете, откуда произошли все люди?

Если люди заселили Африку раньше Азии, то геномы африканцев начали расходиться друг от друга раньше, чем геномы азиатов. Таким образом, мы ожидаем обнаружить, что африканские геномы, у которых было больше времени, чтобы разойтись друг от друга, имеют больше мутаций (по сравнению друг с другом и другими геномами), чем азиатские геномы. Это рассуждение дает нам подсказку, как проверить, распространился ли человеческий род из Африки.

Африканские геномы действительно более разнообразны, чем геномы с других континентов, что привело Уилсона и его коллег к выводу, что африканская родословная является древнейшей и что современные люди ведут свои корни в Африку. Таким образом, популяция африканцев, первых современных людей, образует одно поддерево, тогда как другое поддерево представляет собой подгруппу, покинувшую Африку и позже распространившуюся по всему миру. Используя митохондриальное дерево, Уилсон и его коллеги дополнительно подсчитали, что люди появились в Африке 130 000 лет назад, а расовые различия возникли только 50 000 лет назад. На рисунке ниже показаны предполагаемые модели миграции людей, полученные по геномным данным.

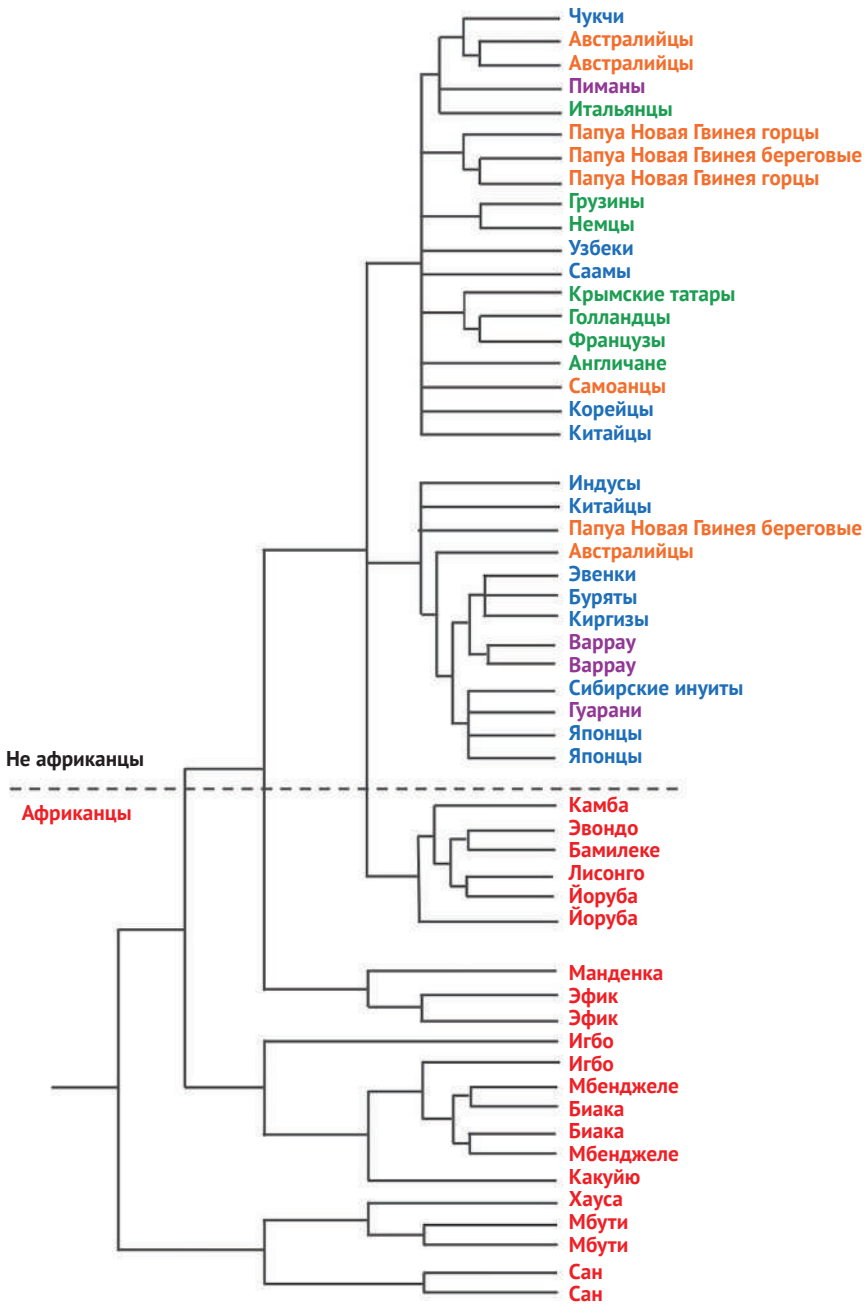


Рис. 7.58 Эволюционное дерево, построенное для различных митохондриальных геномов человека из Африки (красный), Азии (синий), Северной и Южной Америки (фиолетовый), Европы (зеленый) и Океании (оранжевый). Пунктирная линия отделяет африканские геномы от неафриканских

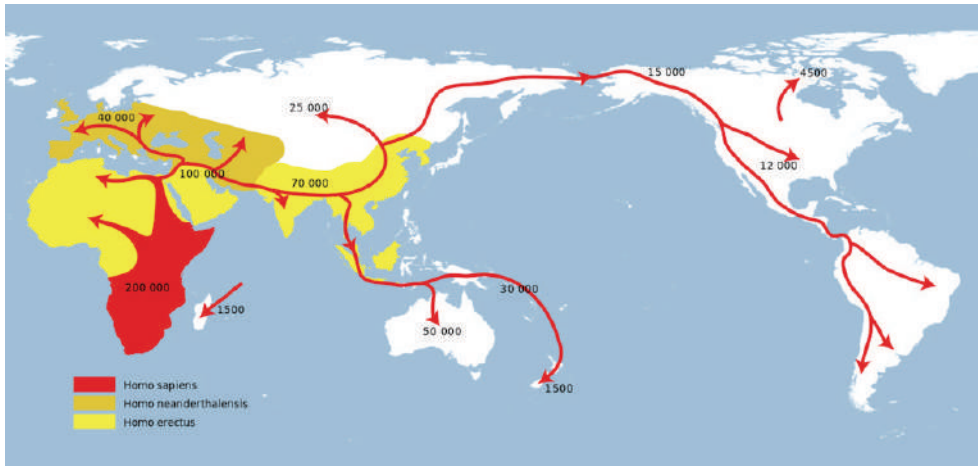


Рис. 7.59 Предполагаемые маршруты миграции людей, полученные по геномным данным, помеченные количеством лет назад, когда произошли эти миграции

Библиографические примечания

Zuckerkandl and Pauling, 1965[1], опубликовали «Молекулы как документы эволюционной истории». Эволюционное дерево, разрешившее загадку гигантской панды, было построено O'Brien et al., 1985[2]. Гипотеза «Из Африки» была предложена Cann, Stoneking, and Wilson, 1987[3]. Исследования происхождения HIV и резервуара приматов вируса HIV были инициированы Gao et al., 1999[4]. Первые молекулярные доказательства передачи HIV в уголовном деле были представлены Metzker et al., 2002[5]. Whiting, Bradler, and Maxwell, 2003[6], опубликовали исследование потерь и восстановлений крыльев у палочников.

Метод UPGMA реконструкции эволюционного дерева был разработан Sokal and Michener, 1958[7]. Алгоритм объединения соседей был разработан Saitou and Nei, 1987[8]. Studier and Keppler, 1988[9], доказали, что этот алгоритм решает задачу филогении по расстоянию для аддитивных деревьев. Алгоритм динамического программирования, решающий задачу минимального показателя эконо-

1 https://www.ncbi.nlm.nih.gov/pubmed/5876245.
2 https://www.nature.com/articles/317140a0.
3 https://www.nature.com/articles/325031a0.
4 https://www.ncbi.nlm.nih.gov/pubmed/9989410.
5 https://www.pnas.org/content/99/22/14292.
6 https://www.ncbi.nlm.nih.gov/pubmed/12529642.
7 https://archive.org/details/cbarchive_33927_astatisticalmethodforevaluatin1902/page/n2.
8 https://www.ncbi.nlm.nih.gov/pubmed/3447015.
9 https://www.ncbi.nlm.nih.gov/pubmed/3221794.

мии, был разработан Sankoff, 1975[1]. Четырехточечное условие было сформулировано Zaretskii, 1965[2]. Метод обмена ближайшими соседями к исследованию деревьев был предложен Robinson, 1971[3]. Felsenstein, 2004[4], предлагает превосходный охват различных алгоритмов реконструкции эволюционного дерева.

[1] https://epubs.siam.org/doi/10.1137/0128004.
[2] http://www.mathnet.ru/links/5c5e375d283525ade1c2e1b170bad8e7/rm6134.pdf.
[3] https://www.sciencedirect.com/science/article/pii/0095895671900207.
[4] https://academic.oup.com/sysbio/article/53/4/669/1649371.

Глава 8

Как дрожжи научились делать вино?

Алгоритмы кластеризации

Эволюционная история виноделия

Как давно мы зависим от алкоголя?

Одним из первых живых организмов, одомашненных человеком, были дрожжи. В 2011 году при раскопках старого кладбища в армянской пещере ученые обнаружили 6000-летнюю винодельню с винным прессом, сосудами для брожения и даже чашками для питья. Эта винодельня была крупным технологическим новшеством, которое требовало понимания того, как контролировать ***сахаромицеты***, род дрожжей, используемых в производстве алкоголя и хлеба.

Тем не менее наш интерес к алкоголю мог возникнуть гораздо раньше, чем 6000 лет назад. В 2008 году ученые обнаружили, что перохвостые древесные землеройки (рисунок ниже), похожие на древних предков всех приматов, являются алкоголиками. Их любимый напиток? «Пальмовое вино», произведенное из цветов бертамовой пальмы и естественно ферментированное дрожжами *Saccharomyces*, живущими на цветках. Это открытие предполагает, что наша собственная склонность к алкоголю может иметь генетическую основу, которая древнее армянской винной пещере на миллионы лет!

Рис. 8.1 Перохвостая древесная землеройка

В пересчете на вес количество пальмового вина, которое потребляют древесные землеройки, было бы смертельным для большинства млекопитающих. К счастью, землеройки разработали эффективные способы метаболизма алкоголя, поэтому они избегают опьянения, которое увеличивало бы риск гибели от хищников. Из-за толерантности к алкоголю древесной землеройки ученые считают, что алкоголь, вероятно, дает землеройкам некоторые эволюционные преимущества, такие как защита от сердечного приступа. Также возможно, что наши более поздние предки-приматы также были пьяницами, – в конце концов, шимпанзе пьют натуральный фруктовый нектар, – так что, возможно, обычай употребления алкоголя мы унаследовали от своих предшественников.

Диауксический сдвиг

Вид дрожжей, рассматриваемый в этой главе, – это *Saccharomyces cerevisiae*, которые могут делать вино, потому что они превращают **глюкозу**, содержащуюся во фруктах, в **этанол**. Поэтому мы начнем с простого вопроса: если *S. cerevisiae* часто живет на виноградных лозах, то почему давленый виноград должен храниться в плотно закрытых бочках для производства вина?

Как только запасы глюкозы иссякают, *S. cerevisiae* должны что-то делать, чтобы выжить. Поэтому дрожжи инвертирует свой метаболизм, и этанол, который они только что произвели, становится их новым источником пищи. Это переключение метаболизма, называемое **диауксическим сдвигом**, может происходить только в присутствии кислорода. Без кислорода *S. cerevisiae* впадает в спячку до тех пор, пока не получит доступа к глюкозе или кислороду. Другими словами, если виноделы не запечатают свои бочки, то дрожжи в бочках начнут перерабатывать только что произведенный этанол, что испортит вино.

Диауксический сдвиг представляет собой сложный процесс, влияющий на экспрессию многих генов. Соответственно, он должен быть результатом крупного эволюционного события, которое дало предку *Saccharomyces* огромное преимущество перед конкурентами: *Saccharomyces* могут не только убивать своих конкурентов, производя этанол, который токсичен для большинства бактерий и других дрожжей, но затем они могут использовать накопленный этанол в качестве источника энергии. Но как и когда *Saccharomyces* изобрели диауксический сдвиг? И какие гены в нем задействованы?

Идентификация генов, ответственных за диауксический сдвиг

Две эволюционные гипотезы с разными судьбами

Помните Сусуму Оно и его модель случайных разрывов? Оно также выдвинул гипотезу о том, что существуют редкие эволюционные события, называемые **полногеномными дупликациями**, или **WGD**, которые дублируют геном цели-

ком. Он предложил эту **модель WGD** в 1970 году, когда не было абсолютно никаких доказательств в ее поддержку. Тем не менее он считал, что WGD требуется во время критической эволюционной инновации для того, чтобы вид реализовал какую-то новую революционно функцию, такую как диауксический сдвиг.

Например, представьте себе время миллионы лет назад, когда появились первые плодовые растения, но ни один организм не мог метаболизировать глюкозу, вырабатываемую этими плодами, и использовать полученный этанол. Первый вид, сделавший это, обладал бы огромным эволюционным преимуществом, но метаболизировать глюкозу – не говоря уже об этаноле – задача не из простых. Вместо того чтобы создавать новый ген тут или там, диауксический сдвиг потребовал бы создания новых метаболических путей, в которых многие гены работали бы вместе. Сусумо Оно утверждал, что WGD может обеспечить платформу для такой революционной инновации, если каждый дублированный ген будет иметь две копии. Одна копия могла бы свободно развиваться без ущерба для существующей функции гена, которую будет выполнять другая копия.

У модели случайных разрывов и модели WGD были очень разные судьбы. С момента своего предложения модель случайных разрывов была принята биологами и стала догмой до ее опровержения в 2003 году. Напротив, модель WGD изначально была встречена со скептицизмом (поскольку только 13 % генов *S. cerevisiae* дублируются) и не набрала популярности в течение 25 лет.

В 1997 году Вулф и Шилдс представили первые вычислительные аргументы в пользу WGD у *S. cerevisiae*. Они утверждали, что тот факт, что только 13 % генов *S. cerevisiae* дублируются, неудивителен, потому что, даже если сотни генов вносят свой вклад в эволюционную инновацию WGD, большинство генов для этой инновации не нужны. Таким образом, ненужные дубликаты генов будут бомбардироваться мутациями, пока не превратятся в псевдогены и в конце концов не исчезнут из генома через миллионы лет. Чтобы узнать больше об аргументах, связанных с WGD у *S. cerevisiae*, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Полногеномная дупликация или серия дупликаций одного гена?**.

Какие гены дрожжей вызывают диауксический сдвиг

Одним из многих этапов, которые выполняют дрожжи во время ферментации, является превращение **ацетальдегида** в этанол. Если впоследствии кислород становится доступным, накопленный этанол снова превращается в ацетальдегид. Превращение ацетальдегида в этанол и этанола в ацетальдегид катализируется ферментом, называемым **алкогольдегидрогеназой (Adh)**. У *S. cerevisiae* активность Adh управляется двумя генами, Adh_1 и Adh_2, которые возникли в результате дупликации одного гена-предка. Фермент, кодируемый Adh_1, обладает повышенной способностью производить этанол, тогда как фермент, кодируемый Adh_2, обладает повышенной способностью перерабатывать этанол.

В 2005 году Майкл Томсон использовал множественное выравнивание генов *Adh* различных видов дрожжей для реконструкции древнего гена *Saccharomyces*. Этот ген имел предпочтение к превращению ацетальдегида в этанол, что напоминало поведение Adh_1. Таким образом, Томсон пришел к выводу, что до WGD у *Saccharomyces* алкогольдегидрогеназа в основном участвовала в обра-

зовании, а не в потреблении этанола. После WGD Adh_1 выполнял свою первоначальную функцию, в то время как Adh_2 был свободен, и мог способствовать возникновению диауксическому сдвигу.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы нашли остальные гены *S. cerevisae*, которые работают вместе для осуществления диауксического сдвига?

Представьте, что вы можете отслеживать все n генов дрожжей в m контрольных точках по обе стороны от диауксического сдвига, в результате чего получается **матрица экспрессии генов** E, размерностью $n \times m$, где $E_{i,j}$ – число, представляющее уровень экспрессии гена i в контрольной точке j. Сама i-я строка E называется **вектором экспрессии** гена i. Просто взглянув на векторы экспрессии генов дрожжей, вы увидите различные модели поведения генов в отношении диауксического сдвига. Вы увидите гены, экспрессия которых почти не меняется; гены, экспрессия которых быстро увеличивается до диауксического сдвига и снижается после него; гены, экспрессия которых резко возрастает после диауксического сдвига, и т. д.

Хотя в этой главе основное внимание уделяется экспрессии генов в связи с диауксическим сдвигом, матрицы экспрессии широко используются в биологическом анализе. Например, если вектор экспрессии вновь секвенированного гена аналогичен вектору экспрессии гена с известной функцией, биолог может заподозрить, что эти гены выполняют родственные функции. Кроме того, гены со сходными векторами экспрессии могут означать, что гены совместно регулируются, что означает, что их экспрессия контролируется одним и тем же фактором транскрипции. Это предполагает стратегию «вины по ассоциации» для определения функций генов, начиная с нескольких генов с известными функциями и потенциально распространяя функции этих генов на другие гены с аналогичными векторами экспрессии. Наконец, анализ экспрессии генов важен в биомедицинских исследованиях, таких как анализ тканей до и после введения лекарства или сравнение раковых и нераковых клеток. Например, анализ экспрессии привел к **MammaPrint**, диагностическому тесту, который определяет вероятность рецидива рака молочной железы на основе анализа экспрессии 70 генов человека, связанных с активацией и подавлением опухоли.

Тем не менее во всех этих случаях остается вопрос: *какие методы используют биологи для анализа данных об экспрессии генов?*

Введение в кластеризацию

Анализ экспрессии генов

В 1997 году Джозеф ДеРизи провел первый массовый эксперимент по экспрессии генов, отбирая образцы культуры *S. cerevisiae* каждые два часа в течение

шести часов до и после диауксического сдвига. Поскольку у *S. cerevisiae* примерно 6400 генов и было взято семь проб, в результате этого эксперимента была получена матрица экспрессии генов 6400×7.

☞ **Загрузить данные 8.1**

ОСТАНОВИТЕСЬ и задумайтесь. Какую технологию вы бы использовали для создания этой матрицы?

Мы уже познакомились с тремя технологиями, которые можно использовать для создания этой матрицы (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Измерение экспрессии генов**). Однако к 1997 году ни одна из этих технологий еще не созрела. По этой причине ДеРизи пришлось использовать ДНК-микрочипы, отличающиеся от ДНК-микрочипов, которые мы обсуждали в предыдущей главе при секвенировании геномов (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Микрочипы**). Микрочипы сегодня используются редко, но алгоритмические методы, которые ДеРизи использовал для анализа микрочипов, одинаково хорошо работают для современных технологий экспрессии генов.

ОСТАНОВИТЕСЬ и задумайтесь. На рисунке ниже показаны векторы экспрессии трех генов дрожжей. Как вы думаете, какие из этих генов участвуют в диауксическом сдвиге?

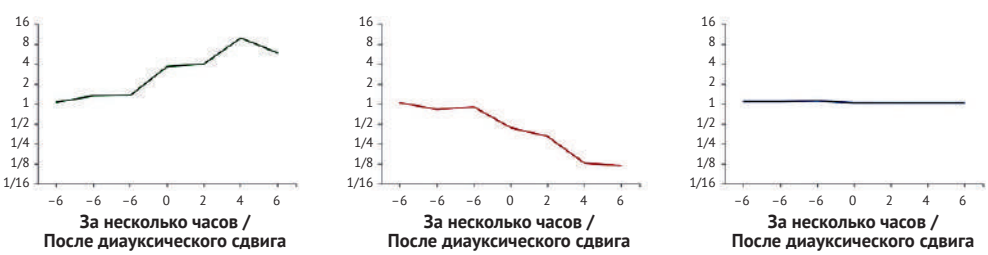


Рис. 8.2 Векторы экспрессии (1,07, 1,35, 1,37, 3,70, 4,00, 10,00, 5,88), (1,06, 0,83, 0,90, 0,44, 0,33, 0,13, 0,12) и (1,11, 1,11, 1,12, 1,06, 1,05, 1,06, 1,05) трех генов дрожжей (YLR258W, YPL012W и YPR055W соответственно), представленных в виде графов. Каждый вектор выражения ($e_1, \ldots, e_m$) представлен как набор отрезков, соединяющих точки (j, e_j) с (j + 1, e_{j+1}) для каждого j между 1 и m – 1 = 6. В эксперименте ДеРизи уровень экспрессии в начальной контрольной точке соответствует базовому уровню экспрессии; обратите внимание, что он близок к 1 для трех генов. Значения выше 1 в векторах экспрессии соответствуют повышенной экспрессии, а значения ниже 1 соответствуют пониженной

Обратите внимание, что паттерн вектора экспрессии гена YPR055W остается плоским во время диауксического сдвига. Таким образом, мы заключаем, что

этот ген, вероятно, не участвует в диауксическом сдвиге. С другой стороны, экспрессия гена YLR258W во время диауксического сдвига значительно изменяется, что позволяет предположить, что этот ген участвует в диауксическом сдвиге. Действительно, проверка базы данных генома *Saccharomyces* показывает, что YLR258W является **гликогенсинтазой**. Этот фермент контролирует выработку **гликогена**, полисахарида глюкозы, который является основным резервуаром для хранения глюкозы в дрожжевых клетках.

ОСТАНОВИТЕСЬ и задумайтесь. Как вы думаете, участвует ли ген YPL012W в диауксическом сдвиге?

На практике биологи часто логарифмируют значения выражений (рисунок ниже). После этой трансформации положительные значения вектора экспрессии гена соответствуют повышенной экспрессии, а отрицательные значения соответствуют пониженной экспрессии.

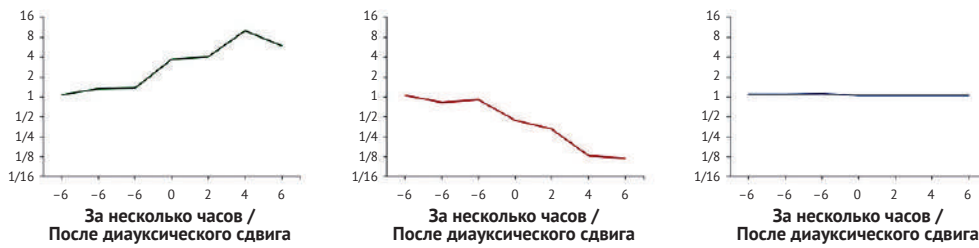


Рис. 8.3 Векторы экспрессии трех генов из обсуждения выше с уровнями экспрессии, замененными их логарифмами по основанию 2: (0,11, 0,43, 0,45, 1,89, 2,00, 3,32, 2,56), (0,09, −0,28, −0,15, − 1,18, −1,59, −2,96, −3,08) и (0,15, 0,15, 0,17, 0,09, 0,07, 0,09, 0,07)

На рис. 8.4 показана матрица экспрессии десяти генов дрожжей после логарифмирования.

Хотя диауксический сдвиг является важным событием в жизни *S. cerevisiae*, он не имеет отношения к большинству функций дрожжей. Поэтому мы подозреваем, что большинство генов *S. cerevisiae* показывает нейтральное поведение на графике во время диауксического сдвига, и мы хотели бы исключить эти гены из дальнейшего рассмотрения, тем самым уменьшив размер матрицы экспрессии.

Уровни экспрессии большинства генов дрожжей почти не изменяются до и после диауксического сдвига (синие гены в матрице дрожжевых генов 10×7, воспроизведенные ниже). Эти гены также обладают тем свойством, что все значения их векторов экспрессии очень близки к нулю. В нашем анализе мы будем исключать гены с векторами экспрессии, все значения которых находятся между $-\delta$ и δ для некоторого параметра δ (в нашем примере мы выбрали

δ = 2,3). Это уменьшает исходный набор данных из 6400 генов дрожжей до набора данных, содержащего всего 230 генов, экспрессия которых значительно меняется в зависимости от диауксического сдвига.

☞ **Загрузить данные 8.2 (сокращенный набор данных диауксического сдвига)**

Ген	Вектор экспрессии						
YLR361C	0.14	0.03	−0.06	0.07	−0.01	−0.06	−0.01
YMR290C	0.12	−0.23	−0.24	−1.16	−1.40	−2.67	−3.00
YNR065C	−0.10	−0.14	−0.03	−0.06	−0.07	−0.14	−0.04
YGR043C	−0.43	−0.73	−0.06	−0.11	−0.16	3.47	2.64
YLR258W	0.11	0.43	0.45	1.89	2.00	3.32	2.56
YPL012W	0.09	−0.28	−0.15	−1.18	−1.59	−2.96	−3.08
YNL141W	−0.16	−0.04	−0.07	−1.26	−1.20	−2.82	−3.13
YJL028W	−0.28	−0.23	−0.19	−0.19	−0.32	−0.18	−0.18
YKL026C	−0.19	−0.15	0.03	0.27	0.54	3.64	2.74
YPR055W	0.15	0.15	0.17	0.09	0.07	0.09	0.07

Рис. 8.4 Субматрица 10×7 матрицы экспрессии генов ДеРизи 6400×7 для десяти генов дрожжей (после логарифмирования по основанию 2 каждого значения экспрессии). Гены с рисунка, представленного ранее, окрашены соответствующим образом

Вы можете видеть на рисунке, показанном выше, что ген YLR258W имеет другой характер изменений, чем ген YGR043C, что указывает на то, что разделение 230 генов дрожжей всего на два кластера (т. е. с повышающимся и понижающимся уровнем экспрессии) может быть слишком упрощенным. Наша цель состоит в том, чтобы сгруппировать эти гены на основе схожих моделей поведения.

YLR361C	**0.14**	0.03	−0.06	0.07	−0.01	−0.06	−0.01
YMR290C	0.12	−0.23	−0.24	−1.16	−1.40	−2.67	**−3.00**
YNR065C	−0.10	**−0.14**	−0.03	−0.06	−0.07	−0.14	−0.04
YGR043C	−0.43	−0.73	−0.06	−0.11	−0.16	**3.47**	2.64
YLR258W	0.11	0.43	0.45	1.89	2.00	**3.32**	2.56
YPL012W	0.09	−0.28	−0.15	−1.18	−1.59	−2.96	**−3.08**
YNL141W	−0.16	−0.04	−0.07	−1.26	−1.20	−2.82	**−3.13**
YJL028W	−0.28	−0.23	−0.19	−0.19	**−0.32**	−0.18	−0.18
YKL026C	−0.19	−0.15	0.03	0.27	0.54	**3.64**	2.74
YPR055W	0.15	0.15	**0.17**	0.09	0.07	0.09	0.07

Рис. 8.5 Жирным шрифтом выделен элемент с наибольшим абсолютным значением в каждом векторе экспрессии

Кластеризация генов дрожжей

Наша цель состоит в том, чтобы **разделить** набор всех генов дрожжей на k непересекающихся **кластеров** таких, чтобы гены в одном кластере имели схожие векторы экспрессии. На практике количество кластеров заранее неизвестно, поэтому биологи обычно применяют алгоритмы кластеризации к данным об экспрессии генов для различных значений k, выбирая значение k, которое имеет биологический смысл. Для простоты будем считать, что k фиксировано. На рисунке ниже показано разделение генов из рисунка, представленного ранее, на три кластера, указывающих на повышенную, пониженную и нейтральную экспрессию во время диауксического сдвига.

YLR361C	**0.14**	0.03	−0.06	0.07	−0.01	−0.06	−0.01
YMR290C	0.12	−0.23	−0.24	−1.16	−1.40	−2.67	**−3.00**
YNR065C	−0.10	**−0.14**	−0.03	−0.06	−0.07	−0.14	−0.04
YGR043C	−0.43	−0.73	−0.06	−0.11	−0.16	**3.47**	2.64
YLR258W	0.11	0.43	0.45	1.89	2.00	**3.32**	2.56
YPL012W	0.09	−0.28	−0.15	−1.18	−1.59	−2.96	**−3.08**
YNL141W	−0.16	−0.04	−0.07	−1.26	−1.20	−2.82	**−3.13**
YJL028W	−0.28	−0.23	−0.19	−0.19	**−0.32**	−0.18	−0.18
YKL026C	−0.19	−0.15	0.03	0.27	0.54	**3.64**	2.74
YPR055W	0.15	0.15	**0.17**	0.09	0.07	0.09	0.07

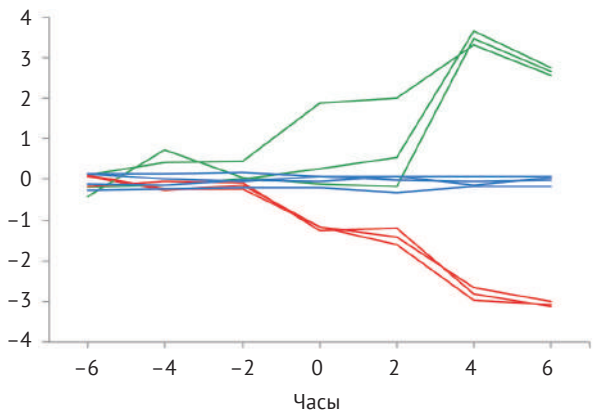


Рис. 8.6 (Вверху) Строки матрицы экспрессии генов из рисунка, представленного ранее, разделены на три кластера. Зеленые гены проявляют повышенную экспрессию, красные демонстрируют пониженную экспрессию, а синие гены – нейтральную экспрессию и вряд ли связаны с диауксическим сдвигом. (Внизу) Строки матрицы, представленные в виде графиков

Принцип правильной кластеризации

Чтобы идентифицировать группы генов с похожими паттернами экспрессии, мы будем думать о векторе экспрессии длины m как о точке в m-мерном пространстве; поэтому гены с похожими векторами экспрессии будут образовывать кластеры из сгруппированных точек. В идеале кластеры должны удовлетворять следующему принципу здравого смысла, который показан на рисунке ниже для $m = 2$.

Принцип правильной кластеризации. Каждая пара точек из одного кластера должна быть ближе друг к другу, чем любая пара точек из разных кластеров.

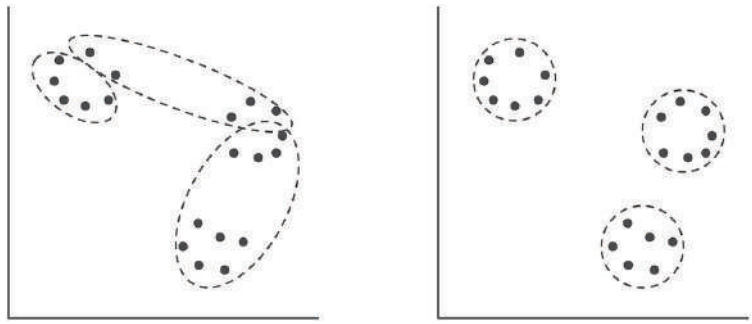


Рис. 8.7 (Слева) Разбиение двадцати точек на три кластера, которые не удовлетворяют принципу правильной кластеризации. (Справа) Другое разделение этих точек, удовлетворяющее принципу правильной кластеризации

Таким образом, мы встроили анализ экспрессии генов в алгоритмическую задачу разделения набора n точек в m-мерном пространстве на k кластеров, которым мы и займемся в этой главе.

Задача правильной кластеризации: *разбейте набор точек на кластеры.*

Input: набор из n точек в m-мерном пространстве и целое число k.

Output: разбиение набора n точек на k кластеров, удовлетворяющих принципу правильной кластеризации.

Упражнение. Сформируйте десять точек в двухмерном пространстве, взяв четвертый и седьмой столбцы матрицы дрожжей 10×7, воспроизведенной ниже. Как эти точки можно разделить на три кластера?

YLR361C	**0.14**	0.03	−0.06	0.07	−0.01	−0.06	−0.01
YMR290C	0.12	−0.23	−0.24	−1.16	−1.40	−2.67	**−3.00**
YNR065C	−0.10	**−0.14**	−0.03	−0.06	−0.07	−0.14	−0.04
YGR043C	−0.43	−0.73	−0.06	−0.11	−0.16	**3.47**	2.64
YLR258W	0.11	0.43	0.45	1.89	2.00	**3.32**	2.56
YPL012W	0.09	−0.28	−0.15	−1.18	−1.59	−2.96	**−3.08**
YNL141W	−0.16	−0.04	−0.07	−1.26	−1.20	−2.82	**−3.13**
YJL028W	−0.28	−0.23	−0.19	−0.19	**−0.32**	−0.18	−0.18
YKL026C	−0.19	−0.15	0.03	0.27	0.54	**3.64**	2.74
YPR055W	0.15	0.15	**0.17**	0.09	0.07	0.09	0.07

Рис. 8.8 Матрица дрожжей

Упражнение. Подсчитайте количество разбиений n точек на два непустых кластера.

Глаз естественным образом разделяет точки на рисунке ниже на два кластера. К сожалению, эти кластеры не удовлетворяют принципу правильной кластеризации; на самом деле такого разбиения этих точек на два кластера не существует! В результате нам нужно будет использовать другой метод, чтобы разработать четко определенную вычислительную задачу для кластеризации.

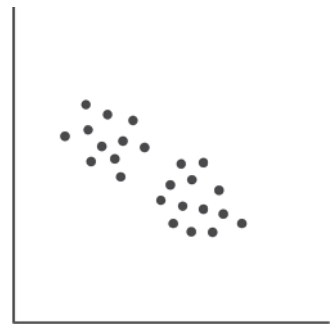


Рис. 8.9 Глаз естественным образом разделяет точки на два кластера

Упражнение. Разработайте полиномиальный алгоритм, чтобы проверить, существует ли решение проблемы правильной кластеризации.

ОСТАНОВИТЕСЬ и задумайтесь. На рисунке ниже показаны восемь точек данных в двухмерном пространстве. Как бы вы разделили эти точки на три кластера? Как мы можем преобразовать задачу правильной кластеризации в четко определенную вычислительную задачу?

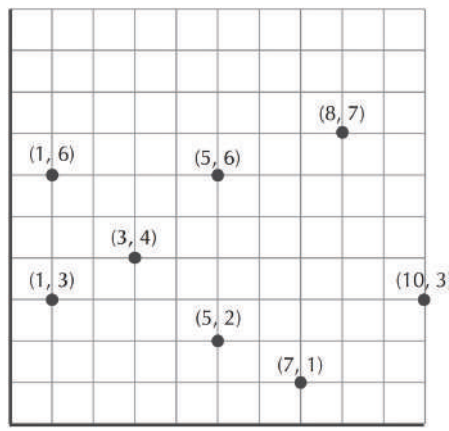


Рис. 8.10 Набор точек в двумерном пространстве

Кластеризация как задача оптимизации

Вместо того чтобы думать о кластеризации как о разделении данных точек данных на k кластеров, мы попытаемся выбрать набор центров *Centers* из k точек, которые будут служить **центрами** этих кластеров. Мы хотели бы выбрать центры так, чтобы они минимизировали некоторую функцию расстояния между центрами и данными по всем возможным выборам центров. Но как определить эту функцию расстояния?

Сначала определим **евклидово расстояние** между точками $\upsilon = (\upsilon_1, \ldots, \upsilon_m)$ и $\omega = (\omega_1, \ldots, \omega_m)$ в m-мерном пространстве, обозначаемое $d(\upsilon, \omega)$, как длину отрезка, соединяющего эти точки,

$$d(v,w) = \sqrt{\sum_{i=1}^{m}(v_i - w_i)^2}.$$

Затем, имея точку *DataPoint* в многомерном пространстве и набор из k точек *Centers*, мы определяем расстояние от *DataPoint* до *Centers*, обозначаемое $d(DataPoint, Centers)$, как евклидово расстояние от *DataPoint* до его ближайшего центра,

$$d(DataPoint, Centers) = \min_{\text{all points } x \text{ from } Centers} d(DataPoint, x).$$

Длина сегментов на рисунке ниже соответствует *d*(*DataPoint, Centers*) для каждой точки *DataPoint*.

Теперь мы определим расстояние между всеми точками данных *Data* и центрами *Centers*. Это расстояние, обозначаемое как *MaxDistance*(*Data, Centers*), является максимальным значением *d*(*DataPoint, Centers*) среди всех точек данных *DataPoint*,

$$MaxDistance(Data, Centers) = \max_{\text{all points } DataPoint \text{ from } Data} d(DataPoints, Centers).$$

На рисунке ниже это расстояние соответствует длине красного сегмента.

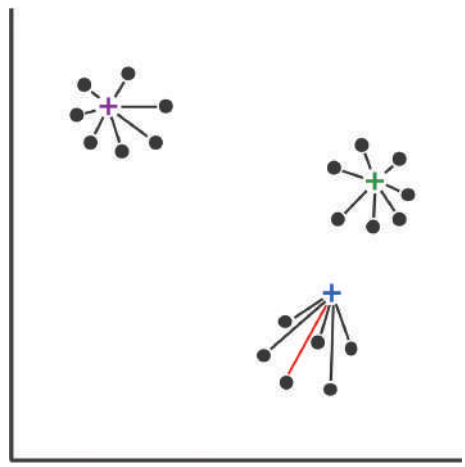


Рис. 8.11 Набор точек *Data* (показаны черными точками) вместе с тремя центрами, образующими набор *Centers* (показаны цветными крестиками). Для каждой точки *DataPoint* в *Data d*(*DataPoint, Centers*) равно длине сегмента, соединяющего ее с ближайшим центром. *MaxDistance*(*Data, Centers*) равно длине самого длинного такого сегмента, который показан красным цветом

Упражнение. Вычислите *MaxDistance*(*Data, Centers*) для данных, показанных на рис. 8.12, и для центров *Centers* (2, 4), (6, 7) и (7, 3).

Теперь мы можем сформулировать четко определенную задачу кластеризации.

Задача кластеризации *k*-центров: *при заданном наборе точек данных найдите k центров, минимизирующих максимальное расстояние между этими точками данных и центрами.*

Input: набор точек *Data* и целое число *k*.

Output: набор *Centers* из *k* центров, которые минимизируют расстояние *MaxDistance*(*DataPoints, Centers*) по всем возможным выборам *k* центров.

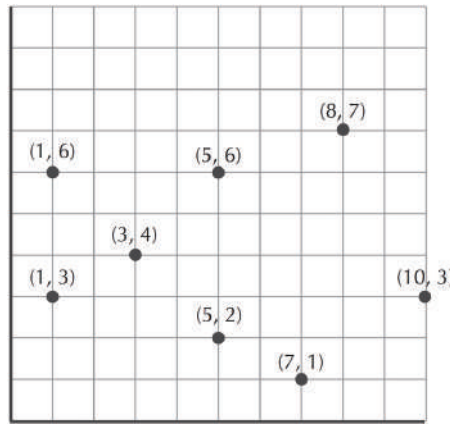


Рис. 8.12 Данные для упражнения

Упражнение. Как бы вы выбрали центр в случае только одного кластера (т. е. когда k = 1)?

Самый дальний первый обход

Самый дальний первый обход

Хотя задачу кластеризации *k*-центров легко сформулировать, она является *NP*-сложной. **Эвристика самого дальнего первого обхода**, псевдокод которой показан ниже, выбирает центры из точек в *Data* (а не из всех возможных точек в *m*-мерном пространстве). Она начинается с выбора произвольной точки в *Data* в качестве первого центра, далее итеративно добавляется новый центр в качестве точки в *Data*, которая находится дальше всего от выбранных до сих пор центров, с произвольно разорванными связями (рисунок ниже).

```
FarthestFirstTraversal(Data, k)
  Centers ← набор, состоящий из одной случайно выбранной точки в Data
  while |Centers| < k
    DataPoint ← точка в Data, максимизирующая d(DataPoint, Centers)
    добавить DataPoint к Centers
  return Centers
```

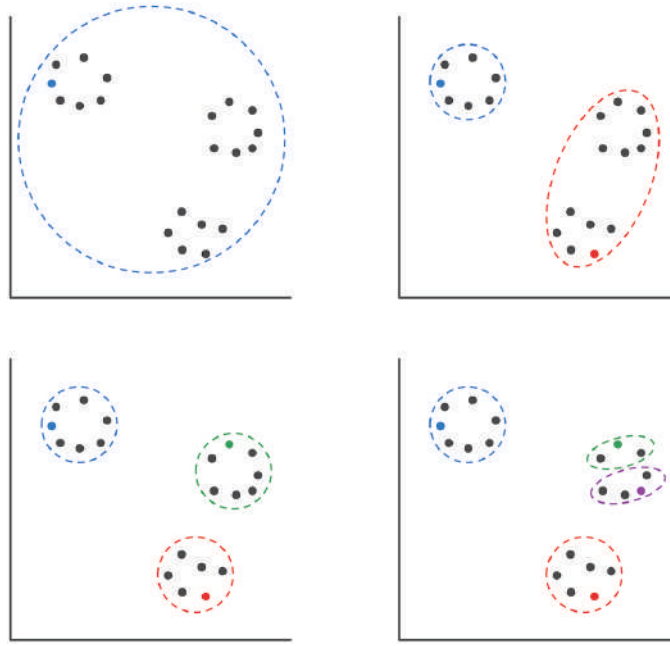


Рис. 8.13 Применение **FarthestFirstTraversal**. (Вверху слева) Произвольная точка из набора данных (показана синим цветом) выбрана в качестве первого центра. Все точки принадлежат одному кластеру. (Вверху справа) Красная точка выбрана в качестве второго центра, так как она находится дальше всего от синей точки. (Внизу слева) После вычисления минимального расстояния каждой точки данных до каждого из первых двух центров мы обнаруживаем, что точка с наибольшим таким расстоянием является зеленой точкой, которая становится третьим центром. (Внизу справа) Четвертый центр показан фиолетовым цветом

Упражнение. Пусть *Centers* будет набором центров, выдаваемым **FarthestFirstTraversal**, а $Centers_{opt}$ будет набором центров, соответствующим оптимальному решению задачи кластеризации *k-Center*. Докажите, что

$$MaxDistance(Data, Centers) \leq 2 \cdot MaxDistance(Data, Centers_{opt}).$$

Можете ли вы найти набор точек данных, центры которых, выдаваемые **FarthestFirstTraversal**, не оптимальны?

FarthestFirstTraversal выполняется быстро, и, согласно предыдущему упражнению, его решение приближается к оптимальному решению задачи кластеризации *k*-центров; однако этот алгоритм редко используется для анализа экспрессии генов. В нем мы выбираем *Centers*, чтобы эти точки минимизировали *MaxDistance*(*Data, Centers*), максимальное расстояние между лю-

бой точкой в *Data* и ее ближайшим центром. Но биологов обычно интересует анализ типичных, а не максимальных отклонений, поскольку последние могут соответствовать выбросам, представляющим ошибки эксперимента (рисунок ниже).

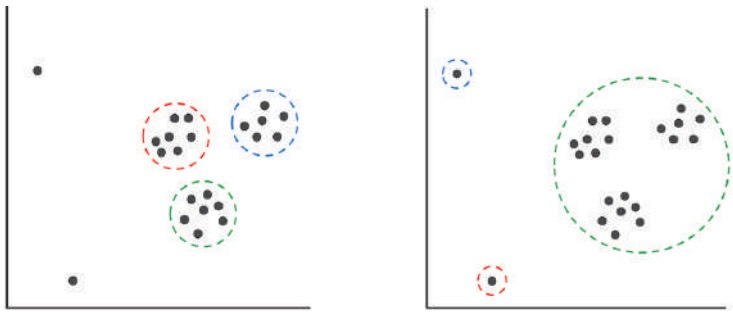


Рис. 8.14 (Слева) Набор точек данных с тремя отчетливо видимыми кластерами и двумя выбросами. (Справа) Поскольку **FarthestFirstTraversal** полагается на *MaxDistance* для вычисления новых центров, если мы попытаемся сгруппировать точки данных в три кластера, то независимо от того, какая точка выбрана в качестве первого центра, два выброса слева будут выбраны как центры 1-элементных кластеров, при этом все остальные точки данных окажутся в одном кластере

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы разработать альтернативную оценочную функцию, более подходящую с биологической точки зрения, чем *MaxDistance*?

Кластеризация *k*-средних

Искажение квадрата ошибки

Чтобы устранить ограничения *MaxDistance*, мы представим новую оценочную функцию. Имея набор *Data* из *n* точек данных и набор *Centers* из *k* центров, **искажение квадрата ошибки** *Data* и *Centers*, обозначаемое *Distortion*(*Data,Centers*), определяется как среднеквадратичное расстояние от каждой точки данных до ее ближайшего центра,

$$Distortion(Data,Centers) = (1/n) \sum_{all\ points\ DataPoint\ in\ Data} d(DataPoint,\ Centers)^2.$$

Обратите внимание, что, в то время как *MaxDistance*(*Data, Centers*) учитывает только длину одного красного сегмента на рисунке ниже, искажение квадрата ошибки учитывает длину всех сегментов на этом рисунке.

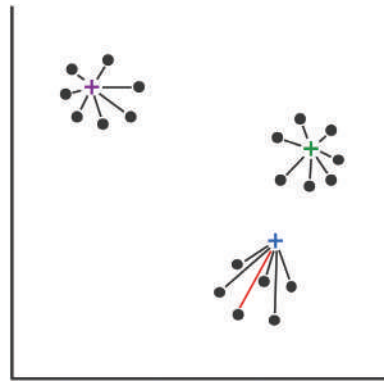


Рис. 8.15 Искажение квадрата ошибки

Упражнение. Вычислите значения *MaxDistance*(*Data, Centers*) и *Distortion*(*Data, Centers*) для восьми точек данных на рисунке ниже и трех центров (3, 4,5), (6, 1,5) и (9, 5). Как изменятся эти значения, если центрами вместо них будут (5/3, 13/3), (6,5, 6,5) и (22/3, 2)?

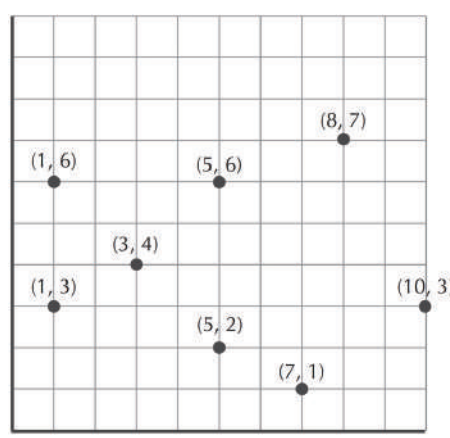


Рис. 8.16 Точки данных для упражнения выше

Задача вычисления искажения квадрата ошибки: *вычислить искажение квадрата ошибки набора точек данных по отношению к набору центров.*

Input: набор точек *Data* и набор центров *Centers*.

Output: искажение квадрата ошибки *Distortion*(*Data, Centers*).

Искажение квадрата ошибки приводит нас к следующей модификации задачи кластеризации *k*-центров.

Задача кластеризации *k*-средних: *по заданному набору точек данных найдите k центральных точек, минимизирующих искажение квадрата ошибки.*

Input: набор точек *Data* и целое число *k*.

Output: набор *Centers* из *k* центров, которые минимизируют *Distortion*(*Data, Centers*) по всем возможным выборкам *k* центров.

Хотя задачи кластеризации *k*-центров и кластеризации *k*-средних могут выглядеть одинаково, они могут давать разные результаты, как показано на рисунке ниже.

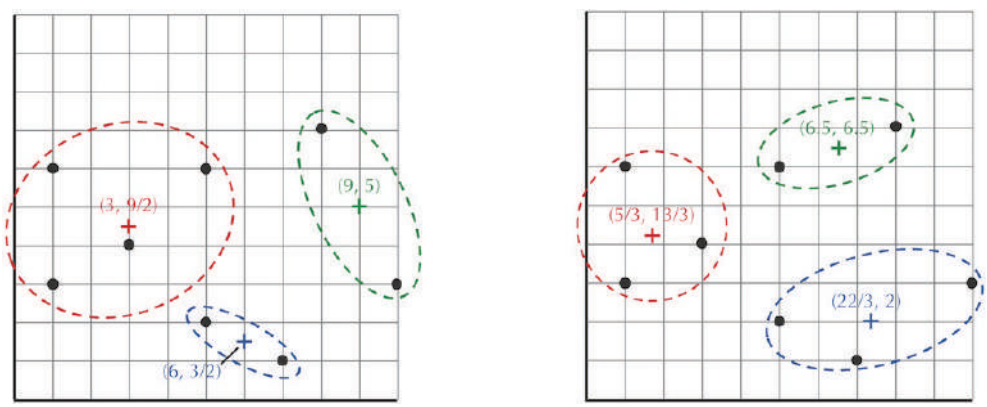


Рис. 8.17 (Слева) Три цветных центра решают задачу кластеризации *k*-центров для восьми черных точек данных из рисунка, представленного ранее вместе с кластерами, которые они образуют. (Справа) Три цветных центра решают задачу кластеризации *k*-средних для этих точек вместе с кластерами, которые они образуют

Ключевое различие между задачами кластеризации *k*-центров и кластеризации *k*-средних заключается в том, что в последней размещение центра гораздо меньше зависит от выбросов (рис. 8.18).

Кластеризация k-средних и центр тяжести

Оказывается, что задача кластеризации ***k*-средних** (или ***k*-means**) является *NP*-трудной, когда $k > 1$. Однако, когда $k = 1$, задача кластеризации *k*-средних сводится к поиску одной центральной точки *x*, которая минимизирует искажение квадрата ошибки. Хотя мы признаем, что разбиение набора точек данных

на один кластер тривиально, остается неясным, как найти единый центр, минимизирующий искажение квадрата ошибки. Мы хотели бы решить эту более простую задачу, потому что это поможет нам разработать эвристику для случая, когда $k > 1$.

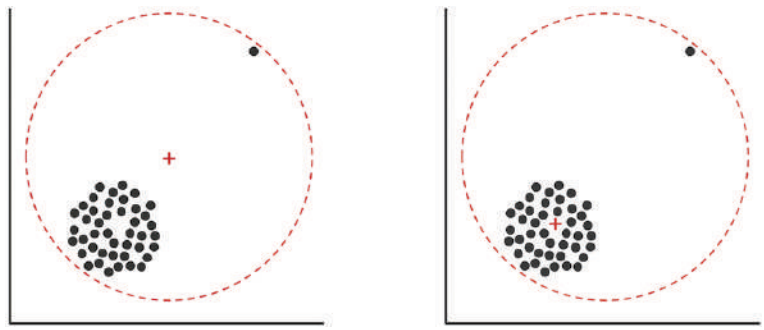


Рис. 8.18 Расположение центра различается в разных постановках задачи кластеризации. (Слева) В задаче кластеризации *k*-центров центр кластера выбирается таким образом, чтобы максимальное расстояние между центром и любой точкой кластера было минимальным. В результате на положение центра могут сильно влиять выбросы. (Справа) В задаче кластеризации *k*-средних влияние выброса на размещение центра намного меньше. Это предпочтительнее при анализе наборов биологических данных, в которых выбросы часто соответствуют ошибочным данным

Определим **центр тяжести** *Data* как точку, *i*-я координата которой является средним значением *i*-х координат всех точек из *Data*. Например, центр тяжести точек (3, 8), (8, 0) и (7, 4) равен

$$\left(\frac{3+8+7}{3}, \frac{8+0+4}{3}\right) = (6, 4).$$

Теорема о центре тяжести. Центр тяжести набора точек *Data* – это единственная точка, решающая задачу кластеризации *k*-средних для $k = 1$.

Доказательство этой теоремы см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Доказательство теоремы о центре тяжести**.

Упражнение. Мы уже говорили, что задача кластеризации *k*-средних является *NP*-трудной для $k > 1$. Однако в случае кластеризации в одномерном пространстве (т. е. когда все точки данных попадают на прямую) задача кластеризации *k*-средних может быть решена за полиномиальное время для любого значения *k*. Разработайте алгоритм для решения проблемы кластеризации *k*-средних в этом случае.

Упражнение. Докажите, что приведенные ниже центры решают задачу кластеризации *k*-средних для черных точек данных, когда $k = 3$.

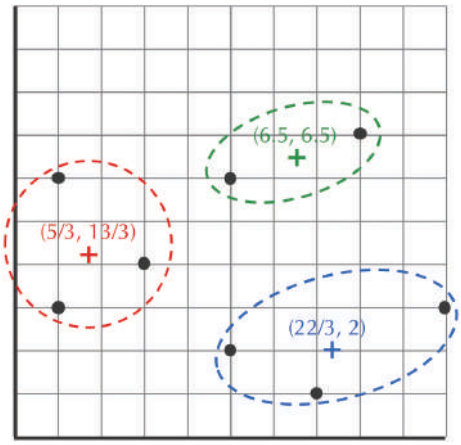


Рис. 8.19 Центры для задачи кластеризации *k*-средних, когда $k = 3$

Алгоритм Ллойда

От центров к кластерам и обратно

Алгоритм Ллойда – одна из самых популярных эвристик кластеризации для задачи кластеризации *k*-средних. Сначала он выбирает *k* произвольных различных точек *Centers* from *Data* в качестве центров, а затем итеративно выполняет следующие два шага (рис. 8.20).

- **Центры в кластеры:** после того, как центры выбраны, назначьте каждую точку данных кластеру, соответствующему его ближайшему центру; связи разрываются произвольно.
- **Кластеры в центры:** после того, как точки данных были назначены кластерам, назначьте центр тяжести каждого кластера в качестве нового центра кластера.

На этом рисунке кажется, что между итерациями центры перемещаются все меньше и меньше. Мы говорим, что алгоритм Ллойда сошелся, если центры (и, следовательно, их кластеры) между итерациями перестают перемещаться.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы найти набор точек данных, для которых алгоритм Ллойда не сходится?

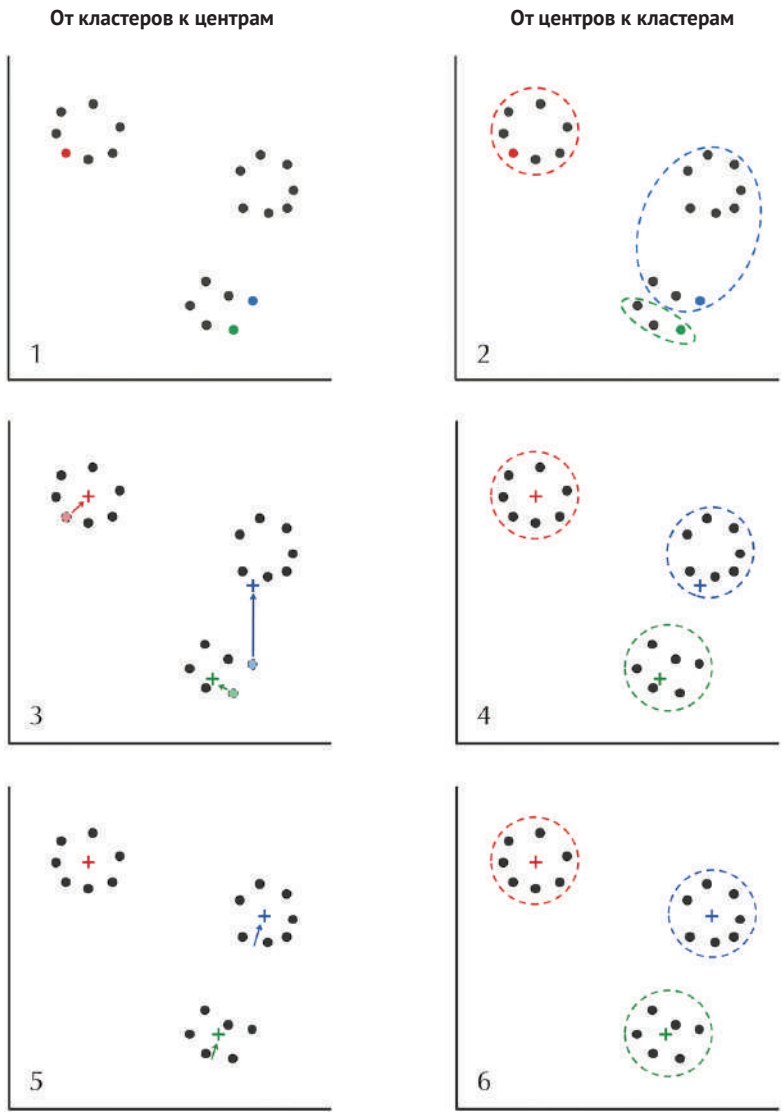


Рис. 8.20 Алгоритм Ллойда в действии для k = 3. На верхней левой диаграмме мы выбираем три произвольные точки данных в качестве центров, показанных точками разного цвета. На последующих диаграммах мы повторяем шаг «От центров к кластерам», за которым следует шаг «От кластеров к центрам». На нижней правой диаграмме алгоритм Ллойда сошелся

Если **алгоритм Ллойда** не сошелся, искажение квадрата ошибки должно уменьшаться на любом шаге в соответствии со следующими рассуждениями:

- если на шаге «От центров к кластерам» точка данных назначается новому центру, то эта точка должна быть ближе к новому центру, чем ее предыду-

щий центр. Таким образом, искажение квадрата ошибки должно уменьшаться;
- на шаге «Кластеры к центру», если центр обновляется как центр тяжести кластера, то по теореме о центре тяжести новый центр является единственной точкой, минимизирующей искажение квадрата ошибки для точек в его кластере. Таким образом, искажение квадрата ошибки должно уменьшаться.

ОСТАНОВИТЕСЬ и задумайтесь. Означает ли это рассуждение, что алгоритм Ллойда всегда сходится?

Утверждение, что алгоритм Ллойда должен сходиться только потому, что искажение квадрата ошибки уменьшается на каждом шаге, не соответствует истине. Например, может быть так, что последующее уменьшение искажения квадрата ошибки становится все меньше и меньше, что приводит к бесконечному процессу (например, если искажение ошибки уменьшается на 1/2, затем на 1/4, затем на 1/8 и т. д.). Следующее упражнение показывает, что такой сценарий не может произойти.

Упражнение. Докажите, что количество итераций алгоритма Ллойда не превышает количества разбиений набора данных на k кластеров.

Тот факт, что алгоритм Ллойда сходится, не означает, что он сходится к оптимальному решению проблемы кластеризации k-средних. Чтобы понять почему, попробуйте выполнить следующее упражнение.

Упражнение. Запустите алгоритм Ллойда для набора из четырех одномерных точек данных {0, 1, 1,9, 3} с двумя начальными центрами {1, 3}. Приводит ли алгоритм Ллойда к решению проблемы кластеризации k-средних?

Упражнение. Выберите свой любимый параметр k и запустите алгоритм Ллойда 1000 раз на наборе данных из 230 генов с диауксическим сдвигом, каждый раз инициализируя новый набор из k случайно выбранных центров. Постройте гистограмму искажений квадрата ошибки полученных 1000 результатов. Сколько раз вам пришлось запускать алгоритм Ллойда, прежде чем вы нашли прогон с наивысшим результатом среди ваших 1000 прогонов?

☞ Загрузить данные 8.3 (сокращенный набор данных диауксического сдвига)

Инициализация алгоритма Ллойда

На рисунке ниже показано, что все может пойти совсем не так, если мы не будем обращать внимание на этап инициализации алгоритма Ллойда. На верхнем рисунке мы не выбрали ни одного центра из сгустка 1, два центра из сгустка 3 и по одному центру из каждого из сгустков 2, 4 и 5. Как показано на нижнем рисунке, после первой итерации алгоритма Ллойда все точки в сгустках 1 и 2 будут назначены красному центру, который будет перемещаться примерно посередине между группами 1 и 2. Два центра в сгустке 3 разделят точки в нем на два кластера. А центры в сгустках 4 и 5 будут двигаться к середине этих сгустков. Затем алгоритм Ллойда быстро сходится, что приводит к неправильной кластеризации.

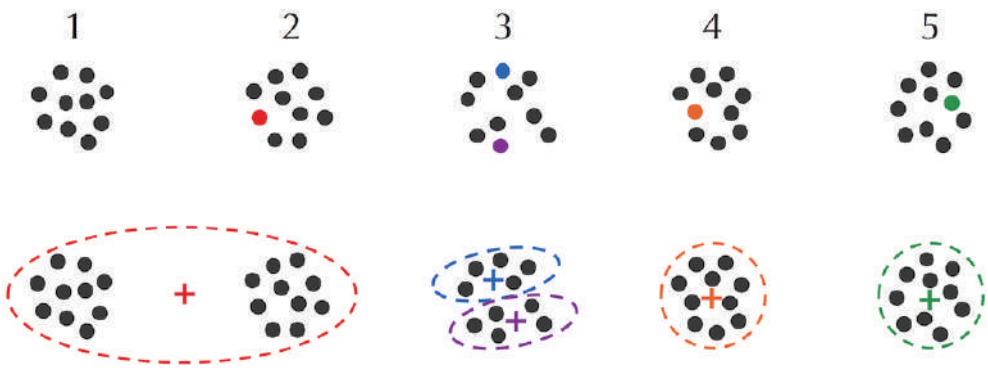


Рис. 8.21 (Вверху) Пять сгустков из десяти точек в двумерном пространстве. Алгоритм Ллойда инициализируется таким образом, что сгусток 1 не содержит центров, сгусток 3 содержит два центра (синий и фиолетовый), а каждый из остальных трех сгустков содержит по одному центру (красный, оранжевый и зеленый). (Внизу) Точки сгруппированы в соответствии с центром, которому они назначены. Алгоритм Ллойда объединил точки в кластерах 1 и 2 в один кластер и разделил кластер 3 на два кластера

Упражнение. Оцените вероятность того, что по крайней мере один из пяти кластеров на рисунке ниже не будет иметь центров, если пять центров будут выбраны случайным образом из данных (как в алгоритме Ллойда). Примечание: для счета вероятности предположим, что мы можем выбрать один и тот же центр более одного раза; т. е. мы можем выбирать центры «с заменой».

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы изменили этап инициализации алгоритма Ллойда, чтобы улучшить находимые им кластеры?

Инициализатор k-means++

До сих пор мы не обращали особого внимания на то, как выбираются начальные центры в алгоритме Ллойда, который выбирает их случайным образом. Подобно **FartherstFirstTraversal**, **k-Means++Initializer** (Улучшенная версия алгоритма кластеризации *k*-средних. – *Прим. ред.*) выбирает *k* центры по одному, но вместо выбора точки, наиболее удаленной от выбранных до сих пор, он выбирает каждую точку случайным образом, таким, что вероятность выбора удаленных точек выше, чем вероятность выбора близлежащих точек. В частности, вероятность выбора центра *DataPoint* из *Data* пропорциональна квадрату расстояния *DataPoint* от уже выбранных центров, т. е. $d(DataPoint, Centers)^2$.

В качестве простого примера предположим, что у нас есть только три точки данных и что квадраты расстояний от этих точек до существующих центров центров равны 1, 4 и 5. Тогда вероятность того, что **k-Means++Initializer** выберет каждую из этих точек в качестве следующего центра, равна 1/10, 4/10 и 5/10 соответственно.

```
k-Means++Initializer(Data, k)
    Centers ← набор, состоящей из одной случайно выбранной точки из Data
    while |Centers| < k
        случайным образом выбрать DataPoint из Data с вероятностью,
        пропорциональной d(DataPoint, Centers)^2
        добавить DataPoint к Centers
    return Centers
```

ОСТАНОВИТЕСЬ и задумайтесь. Хотя **k-Means++Initializer** тоже может попасть в ловушку, показанную ранее, это случается редко. Почему?

Упражнение. Усильте свою реализацию алгоритма Ллойда с помощью **k-Means++Initializer** и примените его к набору данных из 230 генов диауксического сдвига для различных значений *k*.

☞ **Загрузите данные 8.4 (сокращенный диауксический набор данных)**

Кластеризация генов, вовлеченных в диауксический сдвиг

Поскольку выбор наиболее биологически релевантного значения *k* может оказаться сложной задачей, мы (несколько произвольно) выберем кластеризацию 230 генов дрожжей в шесть кластеров (рис. 8.22).

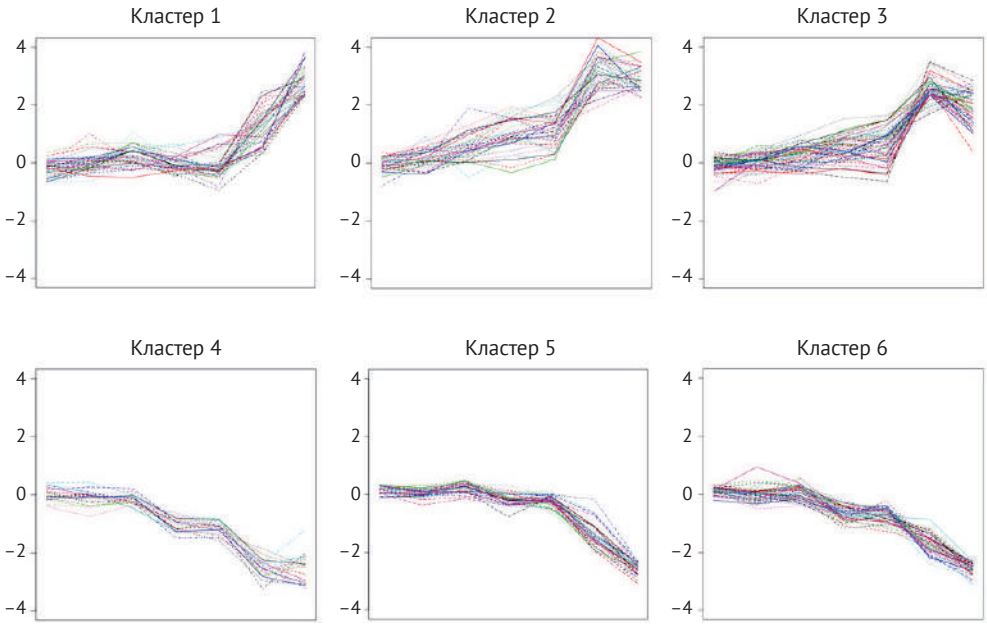


Рис. 8.22 Применение алгоритма Ллойда (с $k = 6$) к сокращенному набору данных о дрожжах, содержащему 230 генов, дает шесть кластеров, содержащих 37, 36, 58, 19, 36 и 44 гена. Векторы экспрессии для всех генов в каждом из этих шести кластеров визуализируются как отдельные диаграммы

Чтобы выявить закономерности векторов экспрессии в каждом кластере, мы усредняем эти векторы (рис. 8.23).

Диаграммы дрожжей показывают шесть типов поведения генов, участвующих в диауксическом сдвиге, и поднимают вопросы для дальнейших биологических исследований, выходящих за рамки этой главы. Например, какие регуляторные механизмы заставляют гены в первом кластере увеличивать свою экспрессию? Какие механизмы заставляют гены четвертого кластера снижать свою экспрессию? И как эти изменения способствуют диауксическому сдвигу?

Упражнение. Кластеризация всего набора данных из 6400 генов дрожжей в шесть кластеров подразумевает кластеризацию сокращенного набора, содержащего 230 генов. Как эта подразумеваемая кластеризация отличается от кластеризации, показанной на предыдущих шагах?

☞ **Загрузите данные 8.5 (полный диауксический набор данных)**

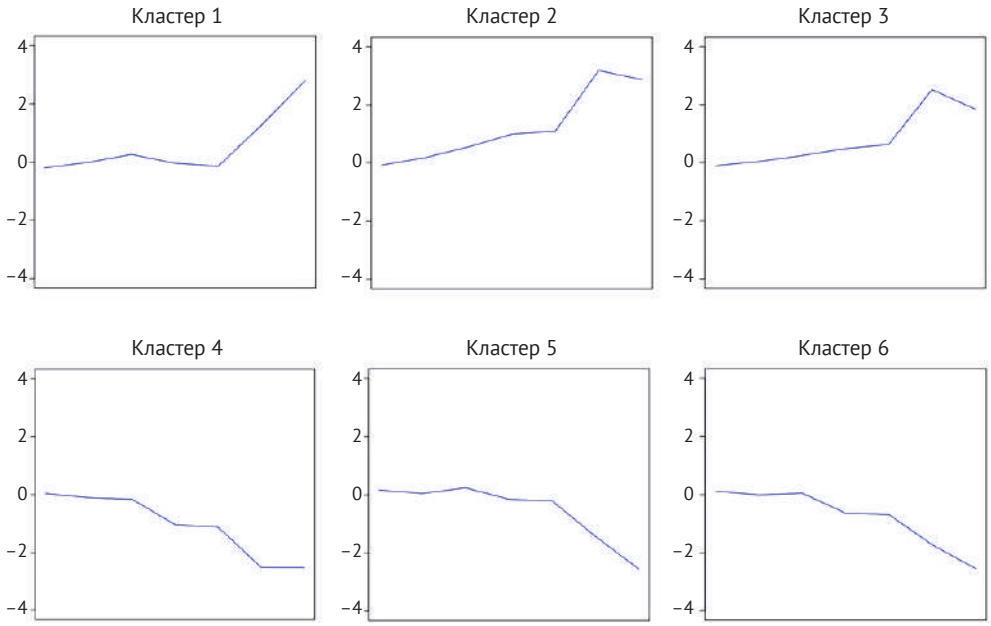


Рис. 8.23 Усреднение векторов экспрессии на каждой диаграмме показывает шесть различных типов регуляторного поведения

Ограничения кластеризации *k*-средних

Ограничения кластеризации k-средних

Увидев алгоритм Ллойда в действии, может показаться, что кластеризация – это просто. Если вы так думаете, подумайте над следующим вопросом.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы сгруппировали точки на рисунке ниже?

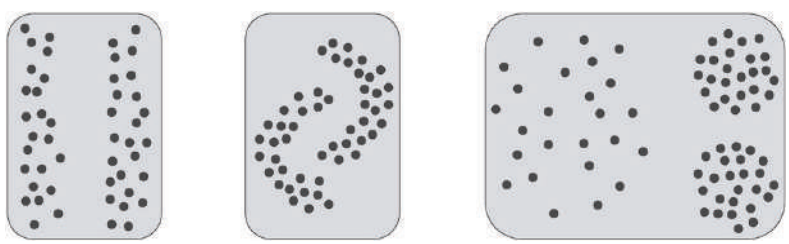


Рис. 8.24 Сложные задачи кластеризации для $k = 2$ (слева и посередине) и $k = 3$ (справа)

В случае сложных задач кластеризации алгоритм Ллойда иногда не может идентифицировать то, что может показаться очевидным кластером (рис. 8.25).

ОСТАНОВИТЕСЬ и задумайтесь. Алгоритм Ллойда назначает каждой точке данных ее ближайший центр, при этом связи разрываются произвольно. Каковы негативные последствия этого назначения?

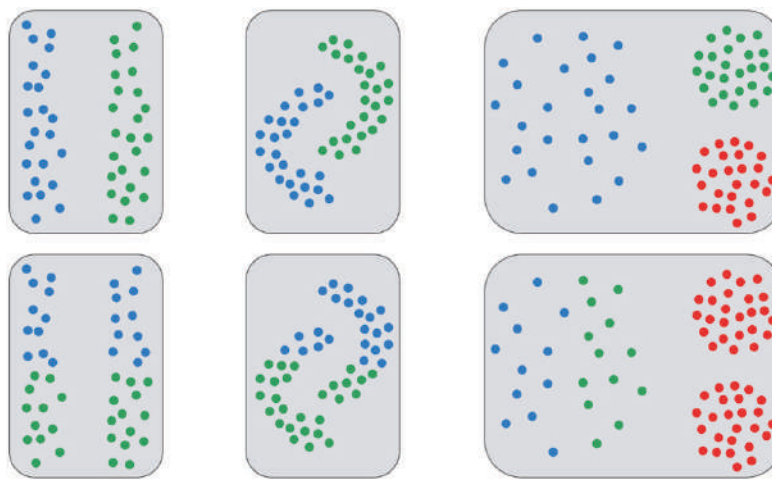


Рис. 8.25 Человеческий глаз (вверху) и алгоритм Ллойда (внизу) часто расходятся в случае вытянутых кластеров (слева), кластеров нешаровидной формы (в центре) и кластеров с сильно разной плотностью точек данных (справа)

Одним из недостатков нашей формулировки задачи кластеризации k-средних является то, что она вынуждает нас «жестко» приписывать каждую точку только одному кластеру. Эта стратегия не имеет особого смысла для **средних точек** или точек, приблизительно равноудаленных от двух центров (рисунок ниже).

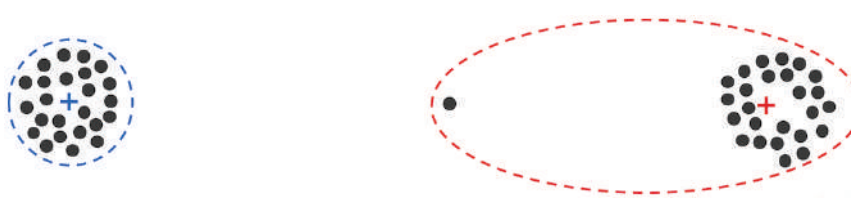


Рис. 8.26 Алгоритм Ллойда жестко приписывает среднюю точку между двумя кластерами одному кластеру, а не дает ему приблизительно равное членство в каждом кластере

Наша цель – перейти от жесткого присвоения точки данных одному кластеру (рисунок ниже (слева)) к «мягкому» назначению (рисунок ниже (справа)). (Также используется термин «нечеткая кластеризация». – *Прим. ред.*)

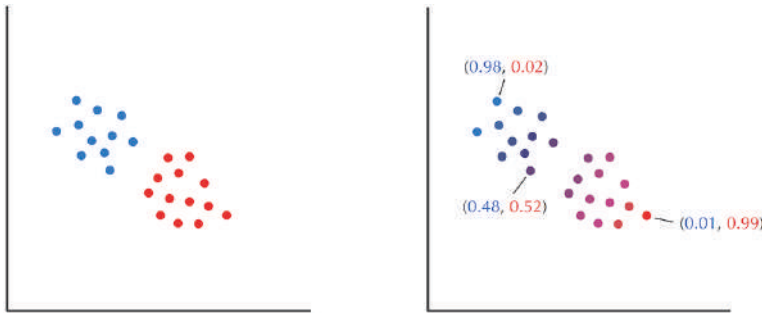


Рис. 8.27 (Слева) Точки, разделенные на два кластера алгоритмом Ллойда; точки окрашены в красный или синий цвет в зависимости от их принадлежности к кластеру. (Справа) Мы можем визуализировать мягкую кластеризацию одних и тех же данных в два кластера, назначая каждой точке пару чисел, представляющих процент «синего» и «красного» для точек на основе ответственности каждого кластера за эту точку. Цвета смешиваются, образуя цвет из красно-синего спектра

От подбрасывания монеты к кластеризации *k*-средних

Подбрасывание монет с неизвестной симметрией

Чтобы разработать алгоритм мягкой кластеризации, мы приведем, казалось бы, несвязанную с этой задачей аналогию. Пьеса Тома Стоппарда «Розенкранц и Гильденстерн мертвы» начинается с того, что два главных героя снова и снова подбрасывают монету, обнаруживая, что это приводит к выпадению орла 157 раз подряд. Розенкранц и Гильденстерн задаются вопросом, оторвались ли они от законов вероятности, но, если бы вы были свидетелем такого события, вы, вероятно, догадались бы, что монета была несимметричной. Предположим, что ваш друг подбрасывает несимметричную монету n раз, и вы хотите оценить вероятность θ того, что при одном подбрасывании этой монеты выпадет орел. В дальнейшем будем называть θ «смещением» (bias).

ОСТАНОВИТЕСЬ и задумайтесь. Для каждого значения θ от 0 до 1 вы можете вычислить вероятность данной последовательности бросков. Как бы вы оценили значение смещения θ, максимизирующее эту вероятность, после наблюдения за серией подбрасываний, когда монета приземляется орлом i из n раз?

Кажется, что наилучшей оценкой θ должно быть количество выпадений орла, деленное на общее количество подбрасываний монеты. Но как мы можем это доказать? Для последовательности n подбрасываний монеты, содержащей i выпадений орла, вероятность того, что монета со смещением θ породила эту последовательность, равна $f(\theta) = \theta^i \cdot (1 - \theta)^{n-i}$. Поскольку наиболее вероятным смещением монет является значение θ, которое максимизирует эту вероятность, мы приравняем производную $f(\theta)$ к нулю:

$$\begin{aligned} f'(\theta) &= i \cdot \theta^{i-1} \cdot (1 - \theta)^{n-i} - \theta^i \cdot (n - i) \cdot (1 - \theta)^{n-i-1} \\ &= [i \cdot (1 - \theta) - \theta \cdot (n - i)] \cdot \theta^{i-1} \cdot (1 - \theta)^{n-i-1} \\ &= (i - \theta \cdot n) \cdot \theta^{i-1} \cdot (1 - \theta)^{n-i-1} = 0. \end{aligned}$$

За исключением θ = 0 и θ = 1, решением этого уравнения является $\theta = i/n$, что означает, что наблюдаемое соотношение выпадений орла обеспечивает наилучшую оценку θ.

Чтобы сделать задачу о подбрасывании монеты немного более интересной, предположим, что ваш друг тайно использует подмену монет A и B, которые выглядят одинаково, но имеют неизвестные смещения θ_A и θ_B. После наблюдения за последовательностью подбрасываний монеты ваша цель состоит в том, чтобы оценить θ_A и θ_B, которые мы вместе обозначаем как *Parameters*.

Мы упростим задачу, предположив, что каждые n подбрасываний ваш друг тайно решает либо оставить себе ту же монету, либо поменять монеты. На рис. 8.28 показаны пять последовательностей из $n = 10$ подмен. Мы представим долю выпаденирй орлов в каждой из этих последовательностей в виде вектора,

$$Data = (Data_1, Data_2, Data_3, Data_4, Data_5) = (0.4, 0.9, 0.8, 0.3, 0.7).$$

Если бы вы знали, что ваш друг использовал монету A в первой и четвертой последовательностях подбрасываний, то вы бы оценили θ_A, вычислив долю орла в этих последовательностях,

$$\theta_A = \frac{Data_1 + Data_4}{2} = \frac{0.4 + 0.3}{2} = 0.35.$$

Затем вы должны оценить θ_B как долю орлов в оставшихся трех последовательностях,

$$\theta_B = \frac{Data_2 + Data_3 + Data_5}{3} = \frac{0.9 + 0.8 + 0.7}{3} = 0.8.$$

Мы представим этот выбор монет как бинарный вектор *HiddenVector* = (1, 0, 0, 1, 0), где 1 в k-й позиции означает, что монета A использовалась для генерации k-й последовательности бросков, а 0 означает, что использовалась монета B. Это обозначение позволяет нам переписать уравнения для *Parameters* в терминах *Data* и *HiddenVector*:

$$\theta_A = \frac{\sum_i HiddenVector_i \cdot Data_i}{\sum_i HiddenVector_i} = \frac{1\cdot 0.4 + 0\cdot 0.9 + 0\cdot 0.8 + 1\cdot 0.3 + 0\cdot 0.7}{1+0+0+1+0} = 0.35;$$

$$\theta_B = \frac{\sum_i (1 - HiddenVector_i)\cdot Data_i}{\sum_i (1 - HiddenVector_i)} = \frac{0\cdot 0.4 + 1\cdot 0.9 + 1\cdot 0.8 + 0\cdot 0.3 + 1\cdot 0.7}{0+1+1+0+1} = 0.80,$$

где *i* пробегает по всем точкам данных.

										Data
H	T	T	T	H	T	T	H	T	H	0.4
H	H	H	H	T	H	H	H	H	H	0.9
H	T	H	H	H	H	H	T	H	H	0.8
H	T	T	T	T	T	H	H	T	T	0.3
T	H	H	H	T	H	H	H	T	H	0.7

Рис. 8.28 Пять последовательностей из десяти подбрасываний монеты приводят к *Data* = (0,4, 0,9, 0,8, 0,3, 0,7). «H» обозначает орел, а «T» – решку

Выражение $\sum_i HiddenVector_i \cdot Data_i$ представляет собой **скалярное произведение** векторов *HiddenVector* и *Data*, записанное как $HiddenVector \cdot Data$. Определим **вектор всех единиц**, $\vec{1}$, как вектор, состоящий из всех единиц и имеющий длину, равную *HiddenVector*. Это позволяет нам записать $\sum_i HiddenVector_i$ как скалярное произведение $HiddenVector \cdot \vec{1}$ и $\sum_i (1 - HiddenVector_i)$ как скалярное произведение $(\vec{1} - HiddenVector) \cdot \vec{1}$. В результате приведенные выше уравнения становятся

$$\theta_A = \frac{HiddenVector \cdot Data}{HiddenVector \cdot \vec{1}};$$

$$\theta_B = \frac{(\vec{1} - HiddenVector) \cdot Data}{(\vec{1} - HiddenVector) \cdot \vec{1}}.$$

ОСТАНОВИТЕСЬ и задумайтесь. Мы только что видели, что с учетом данных и скрытого вектора мы можем найти *Parameters* = (θ_A, θ_B). Если вам даны *Data* и *Parameters*, можете ли вы найти наиболее вероятный выбор *HiddenVector*?

Если мы знаем *Parameters*, то решение о наиболее вероятном выборе *Parameters* соответствует определению того, монета *A* или монета *B* с большей вероятностью произвели *n* наблюдаемых подбрасываний в каждой из пяти последовательностей подбрасывания монеты. Например, предположим, что мы знаем, что *Parameters* = (θ_A, θ_B) = (0,6, 0,82). Если монета *A* использовалась для

генерации пятой последовательности бросков (с семью орлами и тремя решками), то вероятность того, что монета *A* дала этот результат, равна

$$\theta_A^7(1-\theta_A)^3 = 0.6^7 \cdot 0.4^3 \approx 0.00179.$$

Если для генерации пятой последовательности использовалась монета *B*, то вероятность того, что она сгенерировала этот результат, равна

$$\theta_B^7(1-\theta_B)^3 = 0.82^7 \cdot 0.18^3 \approx 0.00145.$$

Поскольку 0,00179 > 0,00145, мы установили бы $HiddenVector_5$ равным 1.

Упражнение. Определите остальные записи в *HiddenVector* для *Parameters* = (0,6, 0,82) для пяти последовательностей бросаний монеты, воспроизведенных на рис. 8.28.

В более общем смысле пусть $Pr(Data_i|\theta)$ обозначает **условную вероятность** генерации исхода $Data_i$ для монеты со смещением θ,

$$Pr(Data_i|\theta) = \theta^{n \cdot Data_i}(1-\theta)^{n \cdot (1-Data_i)}.$$

Если $Pr(Data_i|\theta_A) > Pr(Data_i|\theta_B)$, то монета *A*, скорее всего, сгенерировала *i*-ю последовательность бросков и установит $HiddenVector_i$ равным 1. Если $Pr(Data_i|\theta_A) < Pr(Data_i|\theta_B)$, то монета *B* более вероятна, и мы устанавливаем $HiddenVector_i$ равным 0. Ничьи разбиваются произвольно.

Таким образом, если *HiddenVector* известен, а *Parameters* неизвестны, то мы можем реконструировать наиболее вероятные *Parameters* = (θ_A, θ_B):

$$(Data, HiddenVector, ?) \rightarrow Parameters.$$

Аналогичным образом, если *Parameters* известен, а *HiddenVector* нет, мы можем реконструировать наиболее вероятный *HiddenVector*:

$$(Data, Parameters, ?) \rightarrow HiddenVector.$$

Однако наша первоначальная задача заключалась в том, что и *HiddenVector*, и *Parameters* неизвестны:

$$(Data, ?, ?) \rightarrow ???.$$

В чем же состоит вычислительная задача?

Вы могли заметить, что мы не сформулировали вычислительную задачу, которую пытаемся решить. Итак, определим условную вероятность генерации

последовательности подбрасываний монет $Data_i$ с заданными *HiddenVector* и *Parameters* как

$$Pr(Data_i|HiddenVector, Parameters) = \begin{cases} Pr(Data_i|\theta_A), & \text{если } HiddenVector_i = 1 \\ Pr(Data_i|\theta_B), & \text{если } HiddenVector_i = 0 \end{cases}.$$

Кроме того, определим условную вероятность генерации *Data* с учетом *HiddenVector* и *Parameters* как

$$Pr(Data|HiddenVector, Parameters) = \prod_{i=1}^{n} Pr(Data_i|HiddenVector, Parameters).$$

Для заданных *Data* вычислительная задача, которую мы пытаемся решить, состоит в том, чтобы найти *HiddenVector* и *Parameters*, максимизирующие *Pr*(*Data*|*HiddenVector*, *Parameters*).

От подбрасывания монеты к алгоритму Ллойда

Идентификация *HiddenVector* и *Parameters* из *Data* может показаться безнадежной, но мы уже узнали, что начинать со случайного предположения не обязательно плохая идея. Поэтому мы начнем с произвольного выбора параметров = (θ_A, θ_B) и немедленно реконструируем наиболее вероятный скрытый вектор:

$$(Data, ?, Parameters) \rightarrow HiddenVector.$$

Как только мы узнаем *HiddenVector*, то поставим под сомнение правильность нашего первоначального выбора *Parameters* и пересчитаем *Parameters*':

$$(Data, HiddenVector, ?) \rightarrow Parameters.$$

Как показано на рис. 8.29 для начального выбора *Parameters* = (0,6, 0,82), мы повторяем эти два шага и надеемся, что *Parameters* и *HiddenVector* приближаются к значениям, которые максимизируют Pr(*Data*|*HiddenVector*, *Parameters*),

$$\begin{aligned} (Data, ?, Parameters) &\rightarrow (Data, HiddenVector, Parameters) \\ &\rightarrow (Data, HiddenVector, ?) \\ &\rightarrow (Data, HiddenVector, Parameters') \\ &\rightarrow (Data, ?, Parameters') \\ &\rightarrow (Data, HiddenVector', Parameters') \\ &\rightarrow \quad \ldots \end{aligned}$$

Упражнение. Докажите, что этот процесс завершается, т. е. что *HiddenVector* и *Parameters* в конце концов перестают изменяться от итерации к итерации.

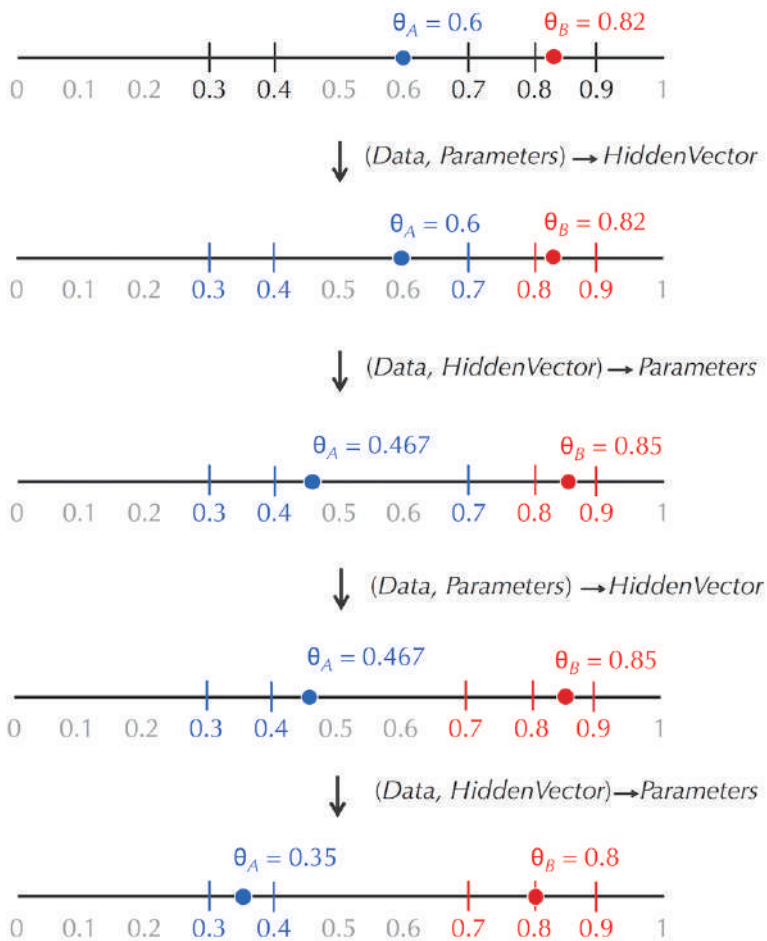


Рис. 8.29 Начиная с *Parameters* = (0,6, 0,82), получается *HiddenVector* = (1, 0, 0, 1, 1) для *Data* = (0,4, 0,9, 0,8, 0,3, 0,7). Затем мы обновляем *Parameters* как (0,467, 0,85), что, в свою очередь, приводит к *HiddenVector* = (1, 0, 0, 1, 0). Этот новый вектор приводит к назначению *Parameters* как (0,35, 0,8), после чего процесс завершается, поскольку *HiddenVector* на следующем шаге не изменится

ОСТАНОВИТЕСЬ и задумайтесь. Если *HiddenVector* состоит из одних нулей, то бросков с монетой *A* нет, и формула для вычисления θ_A недействительна. Что бы вы сделали, чтобы устранить это осложнение?

Вернемся к кластеризации

Рисунок с подбрасыванием монеты (рис. 8.29), показал, что, хотя эта наша аналогия кажется совершенно другой задачей, на самом деле это просто замаскированная задача одномерной кластеризации!

ОСТАНОВИТЕСЬ и задумайтесь. Что такое *Data*, *HiddenVector* и *Parameters* в алгоритме Ллойда?

Имея n точек данных в m-мерном пространстве $Data = (Data_1, \ldots, Data_n)$, мы представляем их приписывание k кластерам в виде n-мерного вектора

$$HiddenVector = (HiddenVector_1, \ldots, HiddenVector_n),$$

где каждый $HiddenVector_i$ может принимать целые значения от 1 до k. Тогда мы представим k центров как k точек в m-мерном пространстве, $Parameters = (\theta_1, \ldots, \theta_k)$. В кластеризации k-средних, как и в аналогии с подбрасыванием монеты, нам даны *Data*, но скрытый вектор и параметры неизвестны. Алгоритм Ллойда начинается со случайно выбранных параметров, и теперь мы можем переписать два его основных шага следующим образом:

- от центров к кластерам: (*Data*, ?, *Parameters*) → *HiddenVector*;
- от кластеров к центрам: (*Data*, *HiddenVector*, ?) → *Parameters*.

Единственная разница между алгоритмом подбрасывания монеты и алгоритмом Ллойда для кластеризации k-средних заключается в том, как они выполняют шаг «от центров к кластерам». В первом случае мы вычисляем $HiddenVector_i$, сравнивая $Pr(Data_i|\theta_A)$ с $Pr(Data_i|\theta_B)$, тогда как во втором мы приписываем точку кластеру, содержащему ближайший к этой точке центр.

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующие вопросы, касающиеся подбрасывания монеты и кластеризации.

- Справедливо ли всегда выбирать монету A, если $Pr(Data_i|\theta_A)$ лишь немного больше, чем $Pr(Data_i|\theta_B)$?
- Справедливо ли всегда приписывать точку центру, если этот центр лишь немного ближе к точке, чем другой центр?

Принятие мягких решений при подбрасывании монет

Максимизация ожиданий: E-шаг

Теперь мы будем использовать нашу аналогию с подбрасыванием монеты, чтобы представить мягкую версию кластеризации *k*-средних. Имея *Parameters* = (θ_A, … , θ_B), мы можем принимать трудные решения для *HiddenVector*, сравнивая $Pr(Data_i|\theta_A)$ с $Pr(Data_i|\theta_B)$. Но это не означает, что мы уверены, какая монета использовалась. Если бы $Pr(Data_i|\theta_B)$ была примерно равна $Pr(Data_i|\theta_A)$, то наша уверенность в том, что использовалась монета *B*, была бы примерно 50 %. С другой стороны, если бы $Pr(Data_i|\theta_B)$ была намного больше, чем $Pr(Data_i|\theta_A)$, то мы были бы почти уверены, что использовалась именно монета *B*. В более общем смысле мы можем говорить о нашей уверенности в том, какая монета использовалась, как об ответственности этой монеты за данную последовательность бросков. (В сумме ответственности должны равняться 1.)

С точки зрения кластеризации *k*-средних, если точка данных является средней точкой между двумя центрами, то каждый из этих центров должен нести примерно одинаковую ответственность за привлечение ее в свои кластеры. Как и в случае с подбрасыванием монет, ответственность всех центров за данную точку данных должна в сумме равняться 1.

ОСТАНОВИТЕСЬ и задумайтесь. Имея *Parameters* = (θ_A, θ_B) = (0,6, 0,82) и последовательность бросков монеты «THHHTHHHTH», как бы вы вычислили ответственность монет *A* и *B*?

Чтобы ответить на предыдущий вопрос, мы увидели, что $Pr(Data_i|\theta_A) = 0{,}67 \cdot 0{,}43 \approx 0{,}00179$ и что $Pr(Data_i|\theta_B) = 0{,}827 \cdot 0{,}183 \approx 0{,}00145$. Раньше мы жестко заключали, что монета *A* более вероятна. Теперь, поскольку монета *A* с большей вероятностью выдаст семь орлов в последовательности из десяти подбрасываний монеты, мы должны возложить на монету *A* большую ответственность, чем на монету *B*. Один из возможных способов распределения этих ответственностей задается формулами:

$$\frac{Pr(0,7|\theta_A)}{Pr(0,7|\theta_A)+Pr(0,7|\theta_B)} = \frac{0.00179}{0.00179+0.00145} \approx 0.55;$$

$$\frac{Pr(0,7|\theta_B)}{Pr(0,7|\theta_A)+Pr(0,7|\theta_B)} = \frac{0.00145}{0.00179+0.00145} \approx 0.45.$$

В результате вместо вектора *HiddenVector* у нас появилась **матрица профиля ответственности** *HiddenMatrix* размерностью 2×5, которую можно построить из данных и параметров (рис. 8.30):

$$(Data, ?, Parameters) \rightarrow HiddenMatrix.$$

Мы называем этот переход **E-шагом**.

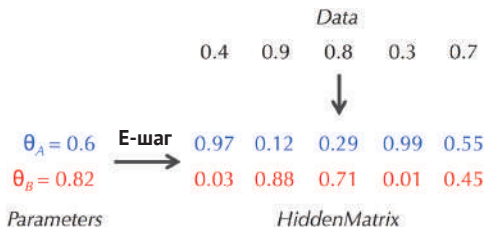


Рис. 8.30 Вычисление *HiddenMatrix* по *Data* = (0,4, 0,9, 0,8, 0,3, 0,7) и *Parameters* = (0,6, 0,82)

Для произвольного набора точек данных $Data = (Data_1, \ldots, Data_n)$ и набора k монет со смещениями $Parameters = (\theta 1, \ldots, \theta_k)$ матрица kxn *HiddenMatrix* определяется как

$$HiddenMatrix_{i,j} = \frac{Pr(Data_j|\theta_i)}{\sum_{1 \le t \le k} Pr(Data_j|\theta_t)}.$$

Максимизация ожиданий: M-шаг

При выполнении сложных назначений мы вычисляли *Parameters* из *Data* и *HiddenVector* следующим образом:

$$\theta_A = \frac{\sum_i HiddenVector_i \cdot Data_i}{\sum_i HiddenVector_i};$$

$$\theta_B = \frac{\sum_i (1 - HiddenVector_i) \cdot Data_i}{\sum_i (1 - HiddenVector_i)}.$$

Чтобы сделать мягкие назначения, обратите внимание, что жесткое назначение результатов двум монетам может быть представлено бинарной матрицей ответственности ниже. Вхождение 1 в i-ю позицию первой строки означает, что мы заключаем, что i-ю последовательность бросков произвела монета A,

а появление 1 во второй строке означает, что i-ю последовательность бросков произвела монета B:

$$HiddenMatrix = \begin{matrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{matrix}.$$

Таким образом, первая строка $HiddenMatrix$, обозначенная как $HiddenMatrix_A$, – это просто $HiddenVector$, а вторая строка $HiddenMatrix$, обозначенная как $HiddenMatrix_B$, – это просто $\vec{1}$ – $HiddenVector$. Поэтому мы можем переписать предыдущие формулы для θ_A и θ_B в терминах $HiddenMatrix$:

$$\theta_A = \frac{HiddenMatrix_A \cdot Data}{HiddenMatrix_A \cdot \vec{1}};$$

$$\theta_B = \frac{HiddenMatrix_B \cdot Data}{HiddenMatrix_B \cdot \vec{1}}.$$

Теперь для матрицы ответственности на рисунке ниже мы можем пересчитать параметры следующим образом:

$$\theta_A = \frac{0.97 \cdot 0.4 + 0.12 \cdot 0.9 + 0.29 \cdot 0.8 + 0.99 \cdot 0.3 + 0.55 \cdot 0.7}{0.97 + 0.12 + 0.29 + 0.99 + 0.55} = \frac{1.41}{2.92} \approx 0.483;$$

$$\theta_B = \frac{0.03 \cdot 0.4 + 0.88 \cdot 0.9 + 0.71 \cdot 0.8 + 0.01 \cdot 0.3 + 0.45 \cdot 0.7}{0.03 + 0.88 + 0.71 + 0.01 + 0.45} = \frac{1.69}{2.08} \approx 0.813.$$

ОСТАНОВИТЕСЬ и задумайтесь. Обратите внимание, что выбор мягких параметров θ_A = 0,483 и θ_B = 0,813 немного ближе друг к другу, чем выбор жестких параметров θ_A = 0,467 и θ_B = 0,85. Как вы считаете, почему это так?

В общем, переход

$$(Data, HiddenMatrix, ?) \to Parameters$$

называется **М-шагом** (см. ниже).

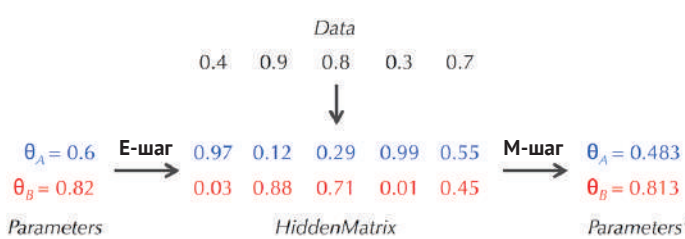


Рис. 8.31 E-шаг и M-шаг

Алгоритм максимизации ожидания

Алгоритм максимизации ожидания начинается со случайного выбора параметров. Затем он чередуется с E-шагом, на котором мы вычисляем матрицу ответственности *HiddenMatrix* для данных с заданными параметрами:

$$(Data, ?, Parameters) \rightarrow HiddenMatrix,$$

и M-шагом, на котором мы пересчитываем *Parameters*, используя *HiddenMatrix*:

$$(Data, HiddenMatrix, ?) \rightarrow Parameters.$$

Упражнение. Выполните еще несколько шагов алгоритма максимизации ожидания для данных, показанных на рис. 8.31. Когда мы должны остановить алгоритм?

Мягкая кластеризация *k*-средних

Применение алгоритма максимизации ожидания к кластеризации

Теперь мы готовы использовать алгоритм максимизации ожидания, чтобы преобразовать алгоритм Ллойда в **алгоритм мягкой кластеризации *k*-средних** (нечеткой кластеризации *k*-средних). Этот алгоритм начинается со случайно выбранных центров и повторяет следующие два шага:

- **от центров к мягким кластерам (E-шаг)**: после того, как центры выбраны, назначьте каждой точке данных значение ответственности для каждого кластера, где более высокие значения соответствуют более сильному членству в кластере;
- **от мягких кластеров к центрам (M-шаг)**: после того, как точки данных были приписаны мягким кластерам, вычислите новые центры.

От центров к мягким кластерам

Начнем с шага «От центров к мягким кластерам» для k центров $Centers = (x_1, \ldots, x_k)$ и n точек $Data = (Data_1, \ldots, Data_n)$. Когда мы ввели E-шаг, мы применили формулу

$$HiddenMatrix_{i,j} = Pr(Data_j|x_i) / \sum_{1 \le t \le k} Pr(Data_j|x_t).$$

Однако эта формула работает только для задачи подбрасывания монеты, когда мы знаем вероятность $Pr(Data_{j|xi})$. Поскольку непонятно, как определить

эту вероятность в случае произвольного набора точек *Data*, опишем другую формулу для вычисления *HiddenMatrix*.

Мы уже использовали термин «центр тяжести», вычисляя центры; если мы представим центры как звезды, а точки данных как планеты, то чем ближе точка к центру, тем сильнее должно быть «притяжение» этого центра к точке. Имея k центров $Centers = (x_1, \ldots, x_k)$ и n точек $Data = (Data_1, \ldots, Data_n)$, нам, следовательно, необходимо построить матрицу ответственности *HiddenMatrix* размерностью $k \times n$, для которой $HiddenMatrix_{i,j}$ является притяжением центра i в точке данных j. Это притяжение можно вычислить в соответствии с ньютоновским законом обратных квадратов всемирного тяготения:

$$HiddenMatrix_{i,j} = \frac{1/d(Data_j, x_i)^2}{\sum_{\text{all center } x_t} 1/d(Data_j, x_t)^2}.$$

К несчастью для поклонников Ньютона, следующая **статистическая сумма** из статистической физики часто на практике работает лучше:

$$HiddenMatrix_{i,j} = \frac{e^{-\beta \cdot d(Data_j, x_i)}}{\sum_{\text{all center } x_t} e^{-\beta \cdot d(Data_j, x_t)}}.$$

В этой формуле e – основание натурального логарифма ($e \approx 2{,}72$), а β – параметр, отражающий степень гибкости нашего мягкого назначения и называемый **параметром жесткости**.

На рис. 8.32 показаны различные методы вычисления *HiddenMatrix*, когда данные представляют точки в одномерном пространстве; он должен дать вам представление о том, почему β называется параметром «жесткости».

ОСТАНОВИТЕСЬ и задумайтесь. Как меняется назначение точек мягким кластерам при $\beta \to -\infty$?

Упражнение. Вычислите *HiddenMatrix*, используя закон обратных квадратов Ньютона для трех центров и восьми точек данных, показанных на рис. 8.33.

0.992	0.988	0.500	0.012	0.008
0.008	0.012	0.500	0.988	0.992
0.500	0.500	0.500	0.500	0.500
0.500	0.500	0.500	0.500	0.500
0.924	0.881	0.500	0.119	0.076
0.076	0.119	0.500	0.881	0.924
0.993	0.982	0.500	0.018	0.007
0.007	0.018	0.500	0.982	0.993
1.000	1.000	0.500	0.000	0.000
0.000	0.000	0.500	1.000	1.000

Рис. 8.32 (Вверху) Пять одномерных точек *Data* = (−3, −2, 0, +2, +3) с двумя центрами *Centers* (показаны синим и красным) *Centers* = (−2,5, +2,5). (Внизу) Пять версий *HiddenMatrix*, построенных для *Data* и *Centers* с использованием ньютоновского закона обратных квадратов (первая матрица) и статистическая сумма с жесткостью β = 0 (вторая матрица), β = 0,5 (третья матрица), β = 1 (четвертая матрица) и β = 1000 (пятая матрица)

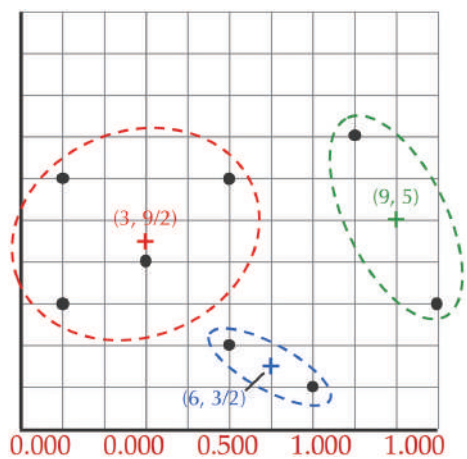


Рис. 8.33 Три центра и восемь точек данных

От мягких кластеров к центрам

Когда мы реализовали М-шаг для подбрасывания монеты, мы получили следующие формулы для θ_A и θ_B:

$$\theta_A = \frac{HiddenMatrix_A \cdot Data}{HiddenMatrix_A \cdot \vec{1}};$$

$$\theta_B = \frac{HiddenMatrix_B \cdot Data}{HiddenMatrix_B \cdot \vec{1}}.$$

В мягкой кластеризации *k*-средних если мы обозначим *i*-ю строку *HiddenMatrix* как $HiddenMatrix_i$, тогда мы можем обновить центр x_i, используя аналог приведенных выше формул. В частности, определим *j*-ю координату центра x_i, обозначаемую $x_{i,j}$, как

$$x_{i,j} = \frac{HiddenMatrix_i \cdot Data^j}{HiddenMatrix_i \cdot \vec{1}}.$$

Здесь $Data^j$ – это *n*-мерный вектор, содержащий *j*-е координаты *n* точек в *Data*.

Обновленный центр x_i называется **взвешенным центром тяжести** точек *Data*.

Пять версий *HiddenMatrix* для нашего текущего примера воспроизведены ниже:

0.992	0.988	0.500	0.012	0.008
0.008	0.012	0.500	0.988	0.992
0.500	0.500	0.500	0.500	0.500
0.500	0.500	0.500	0.500	0.500
0.924	0.881	0.500	0.119	0.076
0.076	0.119	0.500	0.881	0.924
0.993	0.982	0.500	0.018	0.007
0.007	0.018	0.500	0.982	0.993
1.000	1.000	0.500	0.000	0.000
0.000	0.000	0.500	1.000	1.000

Рис. 8.34 Пять версий *HiddenMatrix* для нашего текущего примера

Вычисление взвешенных центров тяжести для четвертой *HiddenMatrix* дает следующие обновленные центры:

$$x_1 = \frac{0.993 \cdot (-3) + 0.982 \cdot (-2) + 0.500 \cdot (0) + 0.018 \cdot (2) + 0.007 \cdot (3)}{0.993 + 0.982 + 0.500 + 0.018 + 0.007} = -1.995;$$

$$x_2 = \frac{0.007 \cdot (-3) + 0.018 \cdot (-2) + 0.500 \cdot (0) + 0.982 \cdot (2) + 0.993 \cdot (3)}{0.007 + 0.018 + 0.500 + 0.982 + 0.993} = 1.995.$$

ОСТАНОВИТЕСЬ и задумайтесь. Останавливается ли алгоритм кластеризации мягких *k*-средних? Если нет, как бы вы изменили его, чтобы он не работал вечно?

Упражнение. Несколько шагов назад вы вычислили *HiddenMatrix* для восьми точек и трех центров (рис. 8.33). Теперь вычислите центры для этих точек данных еще раз.

Упражнение. Примените кластеризацию мягких *k*-средних к сокращенным данным экспрессии гена дрожжевого диауксического сдвига и сравните результаты с результатами алгоритма Ллойда.

☞ **Загрузить данные 8.6 (сокращенный набор данных диауксического сдвига)**

Иерархическая кластеризация

Введение в кластеризацию по расстояниям

В предыдущей главе мы обсудили два метода реконструкции эволюционного дерева с разными сильными и слабыми сторонами: алгоритмы на основе расстояния (включая алгоритм объединения соседей) и алгоритмы на основе выравнивания (включая алгоритм для задачи минимального показателя экономии). Точно так же биологи не всегда анализируют матрицу экспрессии генов размерностью $n \times m$ в лоб. Вместо этого они иногда сначала преобразуют эту матрицу в **матрицей расстояний** D размерностью $n \times n$, где $D_{i,j}$ указывает расстояние между векторами экспрессии для генов i и j (рисунок ниже). В этом разделе мы увидим, как использовать матрицу расстояний для разделения генов на кластеры (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Преобразование матрицы экспрессии в матрицу расстояний/сходств**).

	1 час	2 часа	3 часа
g_1	10.0	8.0	10.0
g_2	10.0	0.0	9.0
g_3	4.0	8.5	3.0
g_4	9.5	0.5	8.5
g_5	4.5	8.5	2.5
g_6	10.5	9.0	12.0
g_7	5.0	8.5	11.0
g_8	3.7	8.7	2.0
g_9	9.7	2.0	9.0
g_{10}	10.2	1.0	9.2

	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}
g_1	0.0	8.1	9.2	7.7	9.3	2.3	5.1	10.2	6.1	7.0
g_2	8.1	0.0	12.0	0.9	12.0	9.5	10.1	12.8	2.0	1.0
g_3	9.2	12.0	0.0	11.2	0.7	11.1	8.1	1.1	10.5	11.5
g_4	7.7	0.9	11.2	0.0	11.2	9.2	9.5	12.0	1.6	1.1
g_5	9.3	12.0	0.7	11.2	0.0	11.2	8.5	1.0	10.6	11.6
g_6	2.3	9.5	11.1	9.2	11.2	0.0	5.6	12.1	7.7	8.5
g_7	5.1	10.1	8.1	9.5	8.5	5.6	0.0	9.1	8.3	9.3
g_8	10.2	12.8	1.1	12.0	1.0	12.1	9.1	0.0	11.4	12.4
g_9	6.1	2.0	10.5	1.6	10.6	7.7	8.3	11.4	0.0	1.1
g_{10}	7.0	1.0	11.5	1.1	11.6	8.5	9.3	12.4	1.1	0.0

Рис. 8.35 (Вверху) Матрица экспрессии генов из десяти генов, измеренная в трех временных точках. (Внизу) Соответствующая матрица расстояний на основе евклидова расстояния

В предыдущих разделах мы предполагали, что работаем с фиксированным числом кластеров k. Но на практике кластеры часто имеют подкластеры, у которых есть подподкластеры и т. д. Чтобы зафиксировать эту кластерную иерерхию, алгоритм иерархической кластеризации использует матрицу расстояний D размерностью $n \times n$ для организации n точек данных в дерево (рис. 8.36).

Как показано на рис. 8.37, горизонтальная линия, пересекающая дерево в i местах, делит n генов на i кластеров.

Упражнение. На рис. 8.37 показаны два способа кластеризации данных из рис. 8.36. Найдите оставшиеся восемь способов кластеризации этих данных, используя одно и то же дерево.

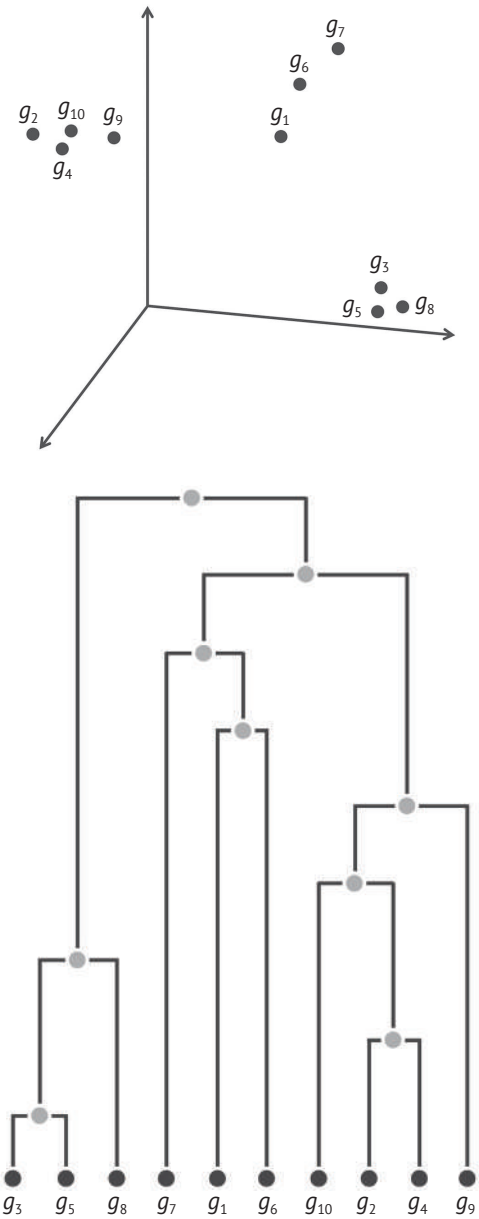


Рис. 8.36 (Вверху) Векторы экспрессии генов из текста выше в виде точек в трехмерном пространстве. (Внизу) Дерево, полученное из матрицы расстояний алгоритмом иерархической кластеризации. Листья на этом дереве соответствуют генам; внутренние узлы соответствуют кластерам генов

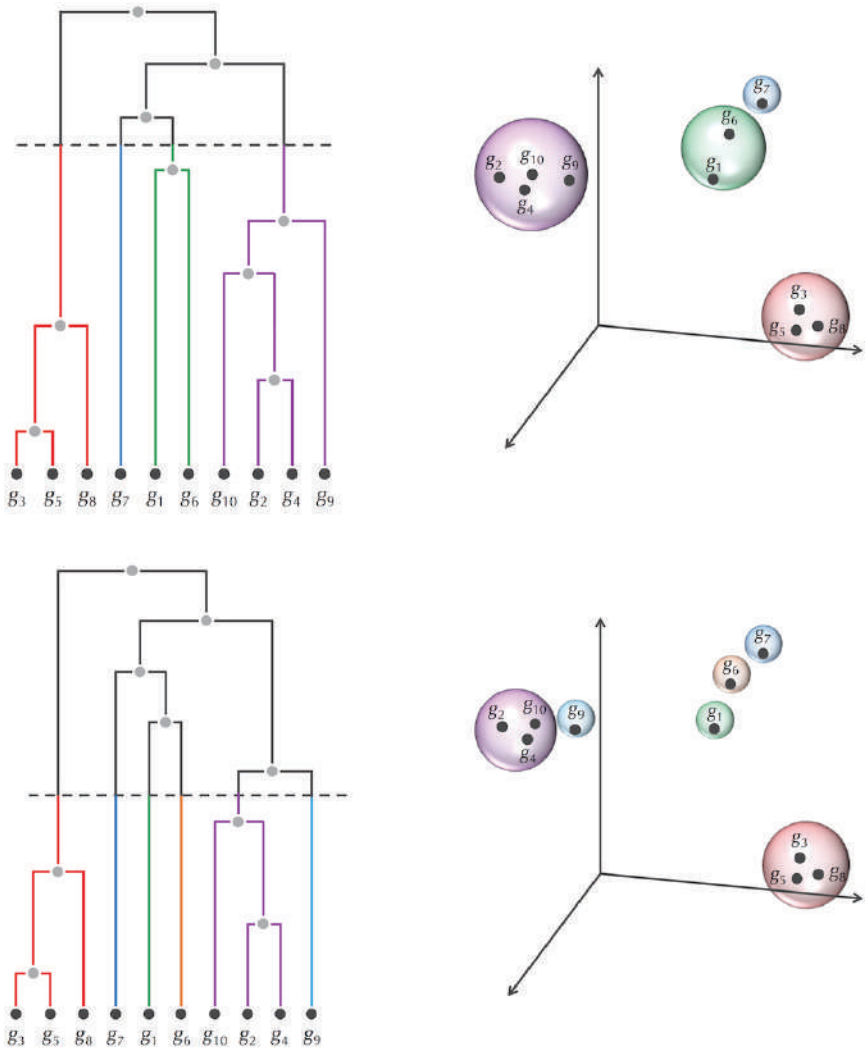


Рис. 8.37 Дерево с *n* листьями предполагает *n* различных способов разделения данных на кластеры. (Вверху) Горизонтальная линия, проходящая через дерево (слева), пересекает данные в четырех местах и разбивает данные на четыре кластера (справа). (Внизу) То же дерево с другой горизонтальной линией (слева) разбивает данные на шесть кластеров (справа)

Определение кластеров по структуре дерева

Алгоритм **HierarchicalClustering**, псевдокод которого показан ниже, постепенно генерирует *n* различных разделов базовых данных в кластеры, представленные деревом, в котором каждый узел помечен кластером генов. Первый раздел имеет *n* одноэлементных кластеров, представленных листьями дерева,

где каждый элемент образует свой собственный кластер. Второй раздел объединяет два «ближайших» кластера в кластер, состоящий из двух элементов. В общем случае i-й раздел объединяет два ближайших кластера из $(i - 1)$-го раздела и имеет $n - i + 1$ кластеров. Мы надеемся, что этот алгоритм выглядит знакомым, – это замаскированный **UPGMA** (из главы о построении эволюционного дерева).

```
HierarchicalClustering(D, n)
    Clusters ← n одноэлементных кластеров, помеченных 1, ..., n
    построить граф T с n изолированными узлами, помеченными отдельными
    элементами 1, ..., n
    while при наличии более одного кластера
        найти два ближайших кластера C_i и C_j
        объединить C_i и C_j в новый кластер C_new вместе с элементами |C_i| + |C_j|
        добавить новый узел, помеченный кластером C_new, в T
        присоединить направленными ребрами узел C_new к C_i и C_j
        удалить строки и столбцы D соответствующие C_i и C_j
        удалить C_i и C_j из Clusters
        добавить строку/столбец в D для C_new, вычислив D(C_new, C) для каждого C
        в Clusters
        добавить C_new в Clusters
    назначьте корень в T как узел без входящих ребер
    return T
```

Обратите внимание, что мы еще не определили, как **HierarchicalClustering** вычисляет расстояние $D(C_{new}, C)$ между вновь сформированным кластером C_{new} и каждым старым кластером C. На практике алгоритмы кластеризации различаются по способу вычисления этих расстояний, и результаты могут сильно различаться. Один широко используемый метод (рисунок ниже) определяет расстояние между кластерами C_1 и C_2 как наименьшее расстояние между любой парой элементов из этих кластеров:

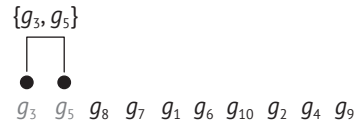
$$D_{\min}(C_1, C_2) = \min_{\text{all points } i \text{ in cluster } C_1, \text{ all points } j \text{ in cluster } C_2} D_{i,j}.$$

Функция расстояния, с которой мы столкнулись в **UPGMA**, использует среднее расстояние между элементами в двух кластерах.

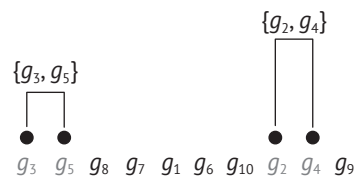
$$D_{\text{avg}}(C_1, C_2) = \frac{\sum_{\text{all points } i \text{ in cluster } C_1} \sum_{\text{all points } j \text{ in cluster } C_2} D_{i,j}}{|C_1| \cdot |C_2|}.$$

Упражнение. Примените **HierarchicalClustering** к матрице расстояний на рисунке, данном ниже, используя D_{avg} вместо $D_{\min}$.

	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}
g_1	0.0	8.1	9.2	7.7	9.3	2.3	5.1	10.2	6.1	7.0
g_2	8.1	0.0	12.0	0.9	12.0	9.5	10.1	12.8	2.0	1.0
g_3	9.2	12.0	0.0	11.2	**0.7**	11.1	8.1	1.1	10.5	11.5
g_4	7.7	0.9	11.2	0.0	11.2	9.2	9.5	12.0	1.6	1.1
g_5	9.3	12.0	0.7	11.2	0.0	11.2	8.5	1.0	10.6	11.6
g_6	2.3	9.5	11.1	9.2	11.2	0.0	5.6	12.1	7.7	8.5
g_7	5.1	10.1	8.1	9.5	8.5	5.6	0.0	9.1	8.3	9.3
g_8	10.2	12.8	1.1	12.0	1.0	12.1	9.1	0.0	11.4	12.4
g_9	6.1	2.0	10.5	1.6	10.6	7.7	8.3	11.4	0.0	1.1
g_{10}	7.0	1.0	11.5	1.1	11.6	8.5	9.3	12.4	1.1	0.0



	g_1	g_2	g_3, g_5	g_4	g_6	g_7	g_8	g_9	g_{10}
g_1	0.0	8.1	9.2	7.7	2.3	5.1	10.2	6.1	7.0
g_2	8.1	0.0	12.0	**0.9**	9.5	10.1	12.8	2.0	1.0
g_3, g_5	9.2	12.0	0.0	11.2	11.1	8.1	1.0	10.5	11.5
g_4	7.7	0.9	11.2	0.0	9.2	9.5	12.0	1.6	1.1
g_6	2.3	9.5	11.1	9.2	0.0	5.6	12.1	7.7	8.5
g_7	5.1	10.1	8.1	9.5	5.6	0.0	9.1	8.3	9.3
g_8	10.2	12.8	1.0	12.0	12.1	9.1	0.0	11.4	12.4
g_9	6.1	2.0	10.5	1.6	7.7	8.3	11.4	0.0	1.1
g_{10}	7.0	1.0	11.5	1.1	8.5	9.3	12.4	1.1	0.0



	g_1	g_2, g_4	g_3, g_5	g_6	g_7	g_8	g_9	g_{10}
g_1	0.0	7.7	9.2	2.3	5.1	10.2	6.1	7.0
g_2, g_4	7.7	0.0	11.2	9.2	9.5	12.0	1.6	1.0
g_3, g_5	9.2	11.2	0.0	11.1	8.1	**1.1**	10.5	11.5
g_6	2.3	9.2	11.1	0.0	5.6	12.1	7.7	8.5
g_7	5.1	9.5	8.1	5.6	0.0	9.1	8.3	9.3
g_8	10.2	12.0	1.0	12.1	9.1	0.0	11.4	12.4
g_9	6.1	1.6	10.5	7.7	8.3	11.4	0.0	1.1
g_{10}	7.0	1.0	11.5	8.5	9.3	12.4	1.1	0.0

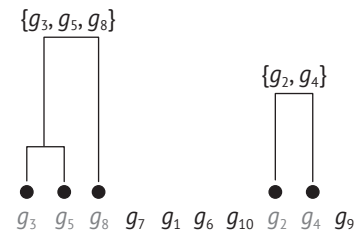


Рис. 8.38 Алгоритм **HierarchicalClustering** в действии. (Вверху слева) Матрица расстояний на следующем шаге (вверху слева) с минимальным элементом, показанным красным, соответствующим генам g_3 и g_5. (Вверху справа) Объединение одноэлементных кластеров, содержащих g_3 и g_5. (В центре слева) Обновленная матрица расстояний после вычисления D_{min} для нового кластера по отношению одного (одноэлементного) кластера к другому, минимальный элемент которой показан красным. (В центре справа) Объединение двух кластеров, соответствующих минимальному элементу. (Внизу) Обновление матрицы расстояний (слева) и объединение двух добавленных кластеров (справа). Последующие шаги будут восстанавливать дерево из рисунка на следующем шаге (внизу справа)

Упражнение. Примените **HierarchicalClustering** (с D_{avg}) к сокращенному набору данных из 230 генов дрожжей и разделите этот набор данных на шесть кластеров. Ожидаете ли вы, что эти кластеры будут примерно такими же, как кластеры, показанные на рисунках ниже? Если нет, то должны ли мы обеспокоиться?

☞ **Загрузить данные 8.7 (сокращенный набор данных диауксического сдвига)**

	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}
g_1	0.0	8.1	9.2	7.7	9.3	2.3	5.1	10.2	6.1	7.0
g_2	8.1	0.0	12.0	0.9	12.0	9.5	10.1	12.8	2.0	1.0
g_3	9.2	12.0	0.0	11.2	0.7	11.1	8.1	1.1	10.5	11.5
g_4	7.7	0.9	11.2	0.0	11.2	9.2	9.5	12.0	1.6	1.1
g_5	9.3	12.0	0.7	11.2	0.0	11.2	8.5	1.0	10.6	11.6
g_6	2.3	9.5	11.1	9.2	11.2	0.0	5.6	12.1	7.7	8.5
g_7	5.1	10.1	8.1	9.5	8.5	5.6	0.0	9.1	8.3	9.3
g_8	10.2	12.8	1.1	12.0	1.0	12.1	9.1	0.0	11.4	12.4
g_9	6.1	2.0	10.5	1.6	10.6	7.7	8.3	11.4	0.0	1.1
g_{10}	7.0	1.0	11.5	1.1	11.6	8.5	9.3	12.4	1.1	0.0

Рис. 8.39 Матрица расстояний

Анализ диауксического сдвига с иерархической кластеризацией

На рисунке ниже показаны векторы экспрессии для каждого из шести кластеров, полученных после применения **HierarchicalClustering** (с использованием D_{avg}) к набору данных дрожжей.

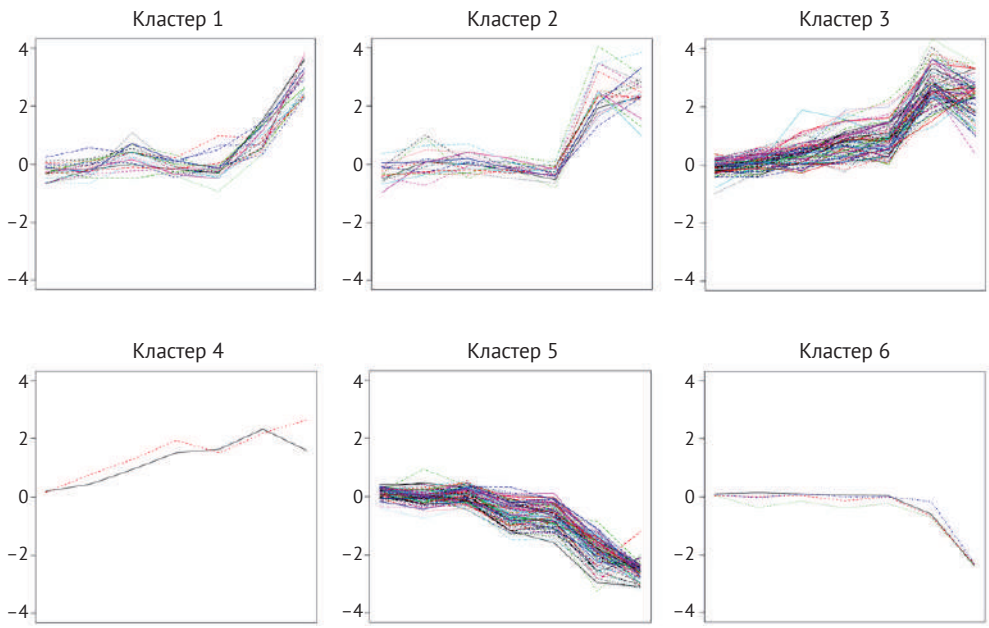


Рис. 8.40 Векторы экспрессии для каждого из шести кластеров, полученных после применения **HierarchicalClustering**

Их средние значения показаны ниже.

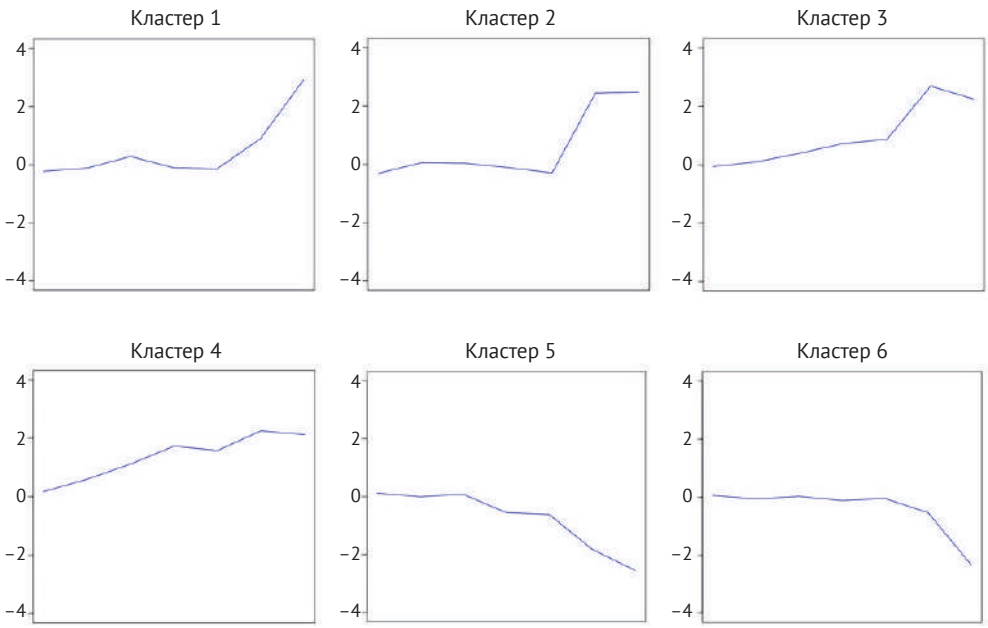


Рис. 8.41 Усредненные векторы экспрессии

Упражнение. Реализуйте **HierarchicalClustering** (с D_{min}, а не D_{avg}) и примените его для разделения набора данных экспрессии генов дрожжей на шесть кластеров. Чем результат отличается от рисунка, представленного выше?

ОСТАНОВИТЕСЬ и задумайтесь. **HierarchicalClustering** и алгоритм Ллойда (рис. 8.42) создали разные кластеры. Должны ли мы быть обеспокоены этим?

Биологов не обескураживает тот факт, что разные методы кластеризации могут создавать разные кластеры, потому что применение этих алгоритмов кластеризации часто является лишь первым шагом на пути к открытию (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Кластеризация и поврежденные клики** для еще одного подхода к кластеризации). По этой причине исследования экспрессии генов обычно сопровождаются экспериментальной работой, чтобы подтвердить, что вычисленные кластеры имеют биологический смысл. После создания кластеров дальнейшие исследования часто сосредотачиваются на конкретных генах внутри этих кластеров.

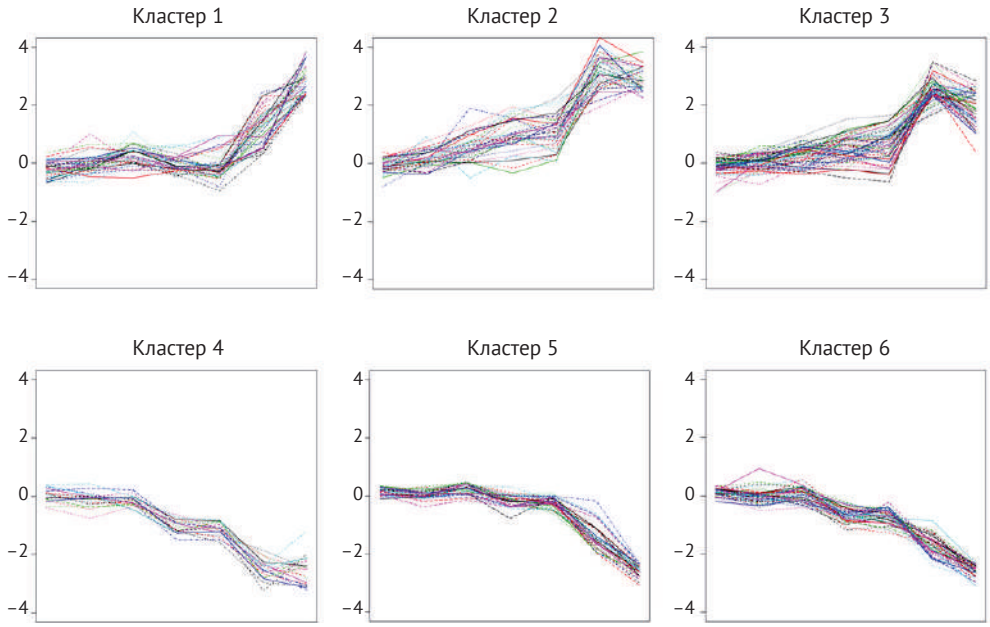


Рис. 8.42 Шесть кластеров генов дрожжей, созданных алгоритмом Ллойда. Обратите внимание, как они отличаются от шести кластеров, созданных **HierarchicalClustering** выше

Например, каждый из кластеров дрожжей, созданных **HierarchicalClustering**, можно дополнительно проанализировать, чтобы выявить подкластеры генов с еще более выраженными профилями экспрессии, чем гены во всем кластере. Например, кластер 1 содержит семь генов, демонстрирующих довольно слабые изменения в первых шести контрольных точках и всплеск экспрессии генов в последней контрольной точке. Биологи обнаружили, что шесть из этих семи генов имеют **регуляторный мотив элемента ответа на источник углерода (CSRE)** с консенсусной последовательностью CATTCATCCG в своих восходящих областях. Дальнейший анализ всего генома дрожжей показал, что только четыре других гена дрожжей имеют этот мотив в своей восходящей области, что позволяет предположить, что было хорошей идеей сгруппировать эти шесть генов в подкластер внутри кластера 1.

Однако более важным вопросом является понимание того, почему эти шесть генов связаны между собой. Дрожжи предпочитают использовать глюкозу в качестве источника энергии по сравнению с другими соединениями, такими как этанол, поэтому в присутствии глюкозы подавляется транскрипция генов, ответственных за метаболизм этого менее вкусного соединения. Таким образом, исследователи пришли к выводу, что мотив CSRE каким-то образом помогает дрожжам ощущать присутствие глюкозы и активирует шесть рассматриваемых генов, когда в организме заканчивается глюкоза, таким образом, выступая в качестве важного компонента диауксического сдвига.

Наконец, если вы считаете, что мы исчерпали все возможные способы кластеризации, взгляните еще раз на рисунок ниже. Хотя **HierarchicalClustering** с D_{min} может найти кластеры слева и посередине, ни один из алгоритмов кластеризации, с которыми мы сталкивались, не может определить кластеры справа. Кластеризация кажется простой задачей отчасти потому, что человеческий глаз очень хорошо умеет группировать точки в формы. В конце концов, исследователи компьютерного зрения все еще пытаются научить компьютеры имитировать наше визуальное восприятие мира, являющееся результатом миллионов лет эволюции.

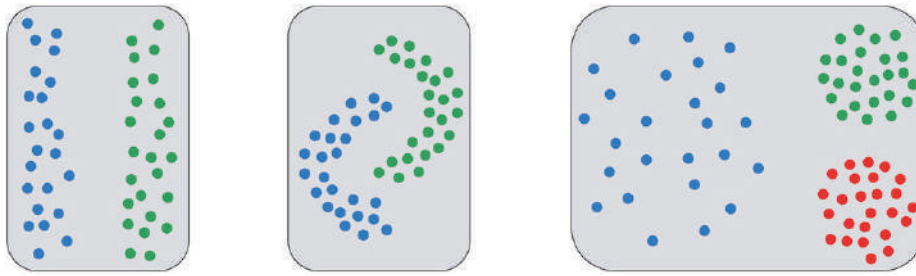


Рис. 8.43 Примеры кластеров

Эпилог. Кластеризация образцов опухоли

Как мы упоминали ранее, анализ экспрессии генов имеет широкий спектр применений, включая исследования рака. В 1999 году Ури Алон проанализировал данные об экспрессии 2000 генов из 40 опухолевых тканей толстой кишки и сравнил их с данными из тканей толстой кишки, принадлежащих 21 здоровому человеку, и все они были измерены в один момент времени. Мы можем представить его данные в виде матрицы экспрессии генов 2000×61, где первые 40 столбцов описывают образцы опухолей, а последние 21 столбец – нормальные образцы.

Теперь предположим, что вы провели эксперимент по экспрессии генов с образцом толстой кишки нового пациента, соответствующим 62-му столбцу в расширенной матрице экспрессии генов. Ваша цель – определить, есть ли у этого пациента опухоль толстой кишки. Поскольку разделение тканей на два кластера (опухолевые и здоровые) известно заранее, может показаться, что классифицировать образец от нового пациента несложно. Действительно, поскольку каждому пациенту соответствует точка в 2000-мерном пространстве, мы можем вычислить центр тяжести этих точек для образца опухоли и для здорового образца. После этого мы можем просто проверить, какой из двух центров тяжести находится ближе к новому образцу ткани.

В качестве альтернативы мы могли бы провести слепой анализ, предполагая, что мы еще не знаем классификацию образцов на раковые и здоровые, и проанализировать полученную матрицу экспрессии 2000×62, чтобы разде-

лить 62 образца на два кластера. Если мы получим кластер, состоящий преимущественно из раковых тканей, этот кластер может помочь нам диагностировать рак толстой кишки.

Заключительная задача. Эти методы могут показаться простыми, но маловероятно, что любой из них позволит надежно диагностировать нового пациента. Как вы думаете, почему это так? Имея матрицу экспрессии генов Алона 2000×61 и данные о генах нового пациента, выработайте лучший метод для подсчета вероятности наличия у этого пациента опухоли толстой кишки.

☞ **Загрузить данные 8.8 (40 образцов опухолевых тканей)**

☞ **Загрузить данные 8.9 (21 образец здоровых тканей)**

☞ **Загрузить данные 8.10 (неизвестный образец)**

Сопутствующие материалы

Полногеномная дупликация или серия дупликаций?

За полногеномной дупликацией (WGD) быстро следуют массовые потери и рекомбинации генов, что затрудняет реконструкцию предварительно дублированного генома. Действительно, как мы упоминали в основном тексте, только 13 % генов имеют дубликаты у современных *S. cerevisiae*. Как же тогда мы можем доказать, что *S. cerevisiae* действительно претерпела WGD, а не последовательность более мелких дупликаций?

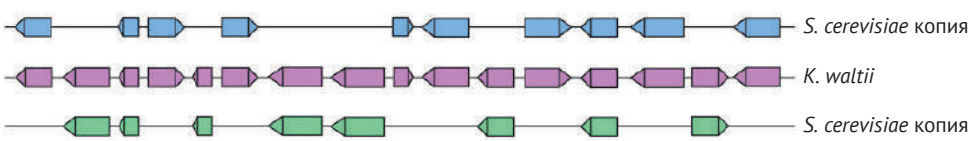


Рис. 8.44 Два синтенных блока *S.cerevisiae* (соответствующие гены которых показаны синим и зеленым цветом) выровнены с синтенным блоком *K. waltii* (гены которой показаны фиолетовым цветом). Хотя только три из 16 генов в этом синтенном блоке имеют две копии в *S. cerevisiae*, каждый ген в синтенном блоке *K. waltii* имеет копию по крайней мере в одном из двух синтенных блоков в *S. cerevisiae*. Фактически большинство синтенных блоков у *K. waltii* демонстрируют это явление, что позволяет предположить, что полногеномная дупликация действительно произошла на эволюционном пути от общего предка *S. cerevisiae* и *K. waltii* к *S. cerevisiae*

В 2004 году Манолис Келлис проанализировал *K. waltii*, родственный вид дрожжей. Сопоставив синтенные блоки *K. waltii* и *S. cerevisiae*, он обнаружил,

что почти каждый синтенный блок *K. waltii* соответствует двум областям *S. cerevisiae*. Обращаясь к возможному сценарию на рисунке ниже, Келлис утверждал, что во время эволюции дрожжей действительно произошла полногеномная дупликация.

ОСТАНОВИТЕСЬ и задумайтесь. Хотя Манолис Келлис утверждал в 2004 году, что приведенная выше схема свидетельствует о WGD, три года спустя Густаво Каэтано-Аноллес выразил сомнения по поводу вывода Келлиса. Можете ли вы придумать альтернативное объяснение приведенному ниже рисунку и предложить другой эволюционный сценарий, не требующий WGD?

Измерение экспрессии генов

В основном тексте мы упомянули, что столкнулись с тремя технологиями, которые можно использовать для измерения экспрессии генов. Во-первых, в масс-спектрометрическом эксперименте (см. главу «Антибиотики») биологи создают набор спектров и сопоставляют их с **протеомом**. Количество спектров, совпадающих с пептидами данного белка, дает представление об уровне экспрессии этого белка. Чтобы оценить экспрессию белка по этому прокси, биоинформатики должны учитывать различную длину белков, плохую фрагментацию некоторых пептидов (что приводит к трудностям в их идентификации) и другие практические проблемы.

Во-вторых, в эксперименте по **секвенированию РНК** мы генерируем риды из **транскриптома** или всех транскриптов РНК, присутствующих в клетке. Оценивая количество транскриптов РНК, кодирующих каждый белок, в образце, мы получаем прокси для экспрессии результирующего белка. В дополнение к транскрипции на продукцию белка в клетке влияет ряд процессов, таких как трансляция, посттрансляционные модификации и денатурация белка. Эти дополнительные факторы могут ослабить корреляцию между количеством транскрипта и экспрессией соответствующего ему белка.

В-третьих, мы можем использовать ДНК-микрочипы (см. главу «Мотивы»), несущие зонды (*k*-меры), нацеленные на каждый ген интересующего вида. Каждый зонд характеризуется интенсивностью, которая дает представление о количестве транскриптов данного гена, присутствующих в образце. Недостатком ДНК-микрочипов является то, что они нацелены только на идентификацию известных транскриптов и часто не могут оценить неизвестные транскрипты. Например, многие виды рака вызываются редкими мутациями и остаются незамеченными при использовании ДНК-микрочипов. Но, в конце концов, секвенирование РНК сейчас более привлекательно в исследованиях рака, и в настоящее время это доминирующая технология для анализа экспрессии генов.

ДНК-микрочипы

Микрочипы, которые ДеРизи использовал для изучения диауксического сдвига, были изготовлены следующим образом. После выделения многих транскриптов РНК, экспрессируемых в дрожжевых клетках, ДеРизи преобразовал каждый транскрипт РНК в **комплементарнаую ДНК (кДНК)** с помощью фермента, называемого **обратной транскриптазой**, и нанес эти кДНК на предметное стекло. Затем он гибридизовал кДНК с флуоресцентно меченной РНК из интересующего образца, чтобы измерить уровни экспрессии различных генов дрожжей.

Количество кДНК, отпечатанной на каждом элементе микрочипа, может сильно различаться, и ДеРизи должен был решить эту задачу, чтобы обеспечить возможность адекватного сравнения интенсивности флуоресценции между элементами микрочипов. Поэтому он гибридизировал две выборки, соответствующие двум разным временным меткам для каждого чипа (рис. 8.45). Затем он пометил образцы разными цветами флуоресцентных красителей, чтобы образцы можно было различить с помощью программного обеспечения для обработки изображений.

Значения экспрессии, полученные из микрочипа, представлены как отношение интенсивностей флуоресценции двух образцов. Таким образом, экспрессия измеряется как относительные изменения экспрессии отдельных генов между образцами и моментами времени. Например, если значение экспрессии гена равно 2, то в первом образце экспрессия этого гена в два раза выше; если значение выражения равно 1/3, то во втором примере выражение в три раза больше. Следуя ДеРизи, исследователи обычно логарифмируют эти коэффициенты экспрессии для создания матриц экспрессии.

Доказательство теоремы о центре тяжести

Обратите внимание, что, когда $k = 1$, задача кластеризации k-средних эквивалентна поиску единственной центральной точки x, которая минимизирует сумму квадратов расстояний от x до точек в *Data*.

Наша цель – показать, что центр тяжести набора точек *Data* – это единственная точка, которая результирует $Distortion(Data, x)$ по всем возможным центрам x. Поскольку квадрат евклидова расстояния между $DataPoint = (DataPoint_1, \ldots, DataPoint_m)$ и центром $x = (x_1, \ldots, x_m)$ равен $\sum_{1 \le j \le m}(DataPoint_j - x_j)^2$, мы имеем:

$$\begin{aligned} Distortion(Data, x) &= \frac{1}{n} \sum_{\text{all points DataPoint in Data}} d(DataPoint, x)^2 \\ &= \frac{1}{n} \sum_{\text{all points DataPoint in Data}} \sum_{j=1}^{m} (DataPoint_j - x_j)^2 \\ &= \frac{1}{n} \sum_{j=1}^{m} \sum_{\text{all points DataPoint in Data}} (DataPoint_j - x_j)^2. \end{aligned}$$

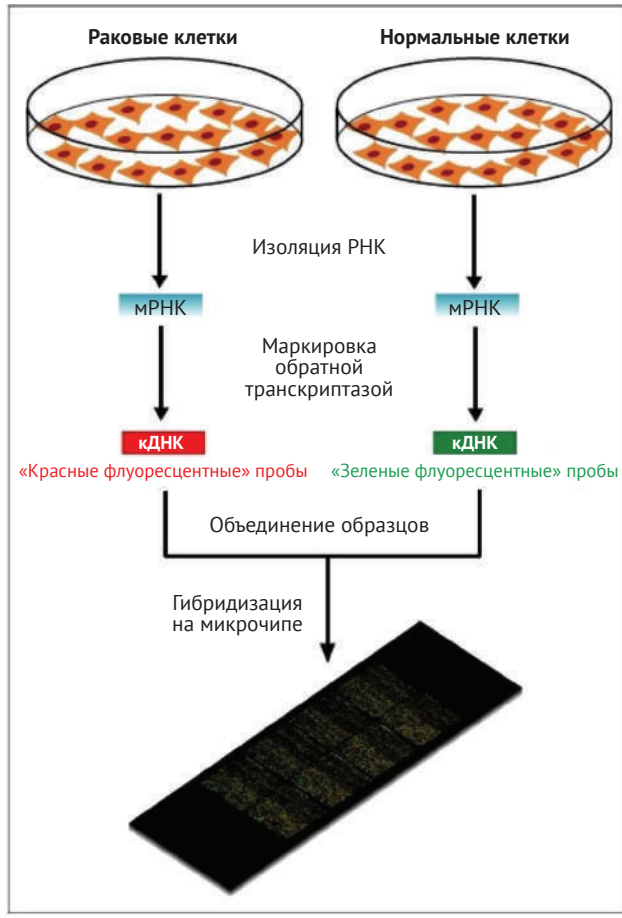


Рис. 8.45 Микрочип, на котором гибридизованы два образца флуоресцентно меченой РНК (красный и зеленый)

Последняя строка этой формулы подразумевает, что мы можем независимо минимизировать *Distortion*(*Data*, *x*) в каждом из *m* измерений, минимизируя каждое из *m* выражений

$$\sum_{\text{все точки } DataPoint \text{ в } Data} (DataPoint_j - x_j)^2.$$

Каждое из этих выражений представляет собой **вогнутую квадратичную функцию** одной переменной x_j. Таким образом, мы можем найти минимум этой функции, найдя точку, где производная функции равна нулю:

$$\sum_{\text{все точки } DataPoint \text{ в } Data} -2 \cdot (DataPoint_j - x_j) = 0.$$

Единственное решение этого уравнения дается выражением

$$x_j = 1/n \sum_{\text{все точки } DataPoint \text{ в } Data} DataPoint_j,$$

которое подразумевает, что j-я координата центра является средним значением j-х координат точек данных. Другими словами, единственное решение задачи кластеризации k-средних для $k = 1$ является центром тяжести всех точек данных.

Матрица экспрессии генов и матрица расстояний/сходств

Существует множество способов количественной счета сходства между векторами экспрессии $x = (x_1, \ldots, x_m)$ и y = $(y_1, \ldots, y_m)$. Одной из возможностей является скалярное произведение $\sum_{i=1}^{m} x_i \cdot y_i$. Другим является **коэффициент корреляции Пирсона** *PearsonCorrelation*(*x, y*), где

$$PearsonCorrelation(x, y) = \frac{\sum_{i=1}^{m} (x_i - \mu(x)) \cdot (y_i - \mu(y))}{\sqrt{\sum_{i=1}^{m} (x_i - \mu(x))^2 \cdot \sum_{i=1}^{m} (y_i - \mu(y))^2}}.$$

В приведенной выше формуле $\mu(x)$ обозначает среднее значение всех координат вектора x.

ОСТАНОВИТЕСЬ и задумайтесь. При заданном векторе x какие векторы y максимизируют и минимизируют *PearsonCorrelation*(*x, y*)?

Коэффициент корреляции Пирсона варьируется от –1 до 1, где –1 указывает на полную отрицательную корреляцию, 0 – на отсутствие корреляции, а 1 – на полную положительную корреляцию. Основываясь на коэффициенте корреляции Пирсона, мы можем определить расстояние Пирсона между векторами x и y как

$$PearsonDistance(x, y) = 1 - PearsonCorrelation(x, y).$$

Упражнение. Вычислите коэффициент корреляции Пирсона для следующих пар векторов:

- $(\cos \alpha, \sin \alpha)$ и $(\sin \alpha, \cos \alpha)$ для произвольного значения α;
- $\sqrt{0.75}$ и $-\sqrt{0.75}$.

Кластеризация и испорченные клики

В исследованиях экспрессионного анализа матрица подобия R часто преобразуется в **граф подобия** $G(R, \theta)$. Узлы этого графа представляют гены, а ребро соединяет гены i и j тогда и только тогда, когда сходство между ними ($R_{i,j}$) превышает пороговое значение θ.

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрим кластеризацию генов, которая удовлетворяет принципу правильной кластеризации: сходство между любыми двумя генами в одном кластере превышает θ, а сходство между любыми двумя генами в разных кластерах меньше θ. Как выглядит граф сходства $G(R, \theta)$ для этих генов?

Если кластеры удовлетворяют принципу правильной кластеризации, то должно быть некоторое значение θ такое, что каждая компонента связности $G(R, \theta)$ является **кликой клика** или графом, в котором каждая пара узлов соединена ребром (рис. 8.46). В общем случае граф, все компоненты связности которого являются кликами, называется **кликовым графом**

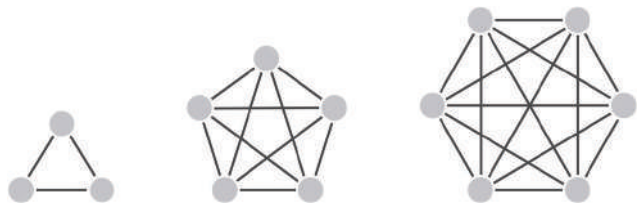


Рис. 8.46 Кликовый граф, состоящий из трех клик

Ошибки в данных выражения и отсутствие универсального порога θ часто приводят к искажению графов подобия, связные компоненты которых не являются кликами (рисунок ниже).

Гены из одного кластера также могут иметь значение сходства ниже θ, что приводит к удалению ребер из клики, либо гены из разных кластеров могут иметь значение сходства, превышающее θ, таким образом добавляя ребра между разными кликами. Это наблюдение заставляет нас задаться вопросом, как преобразовать искаженный граф подобия в кликовый граф, используя наименьшее количество добавлений и удалений ребер.

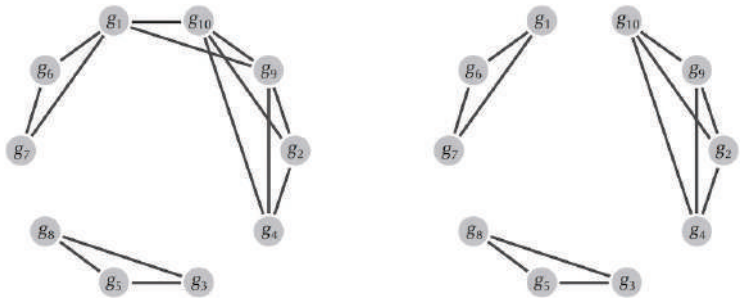


Рис. 8.47 (Слева) Один из возможных графов сходства генов. (Справа) Граф сходства можно преобразовать в кликовый граф (справа), удалив ребра (g_1, g_{10}) и (g_1, g_9)

Задача поврежденных клик: *найдите минимальное количество ребер, которые нужно добавить или удалить, чтобы преобразовать граф в кликовый граф.*

Input: граф.

Output: минимальное количество добавлений и удалений ребер, которые превращают этот граф в кликовый граф.

Задачу поврежденных клик трудно решить точно, поэтому были предложены некоторые эвристики. Алгоритм **«техника поиска сходства кластеров» (Cluster Affinity Search Technique, (CAST))**, описанный ниже, очень хорошо работает при кластеризации данных об экспрессии генов. Определим сходство между геном *i* и кластером *C* как среднее сходство между *i* и всеми генами в *C*:

$$R_{i,C} = \sum_{\text{all elements } i \text{ in cluster } C} \frac{R_{i,j}}{|C|}.$$

При заданном пороге θ ген *i* является **θ-близким** к кластеру *C*, если $R_{i,C} > \theta$, и θ-**удаленным** от *C* в противном случае. Кластер называется **согласованным**, если все гены в *C* θ-близки к *C*, а все гены, не входящие в *C*, являются θ-удаленными от *C*. Алгоритм **CAST** использует граф подобия *G* и порог θ для итеративного поиска согласованных кластеров, начиная с одного элемент кластера *C*, а затем добавление «ближайшего» гена не в *C* и удаление «самого отдаленного» гена в *C*. После того как найден согласованный кластер, все узлы в кластере *C* удаляются из графа сходства, и **CAST** выполняет итерацию по полученному результирующему меньшему графу.

```
CAST(R, ν)
  Graph ← G(R, ν)
  Clusters ← пустой набор
  while Graph является непустым
    C ← одноузловый кластер, состоящий из узла максимальной степени
    в Graph
    while существует ν-близкий ген i не в C или ν-удаленный ген i в C
      найти ближайший ν-близкий ген i не из C и добавить его в C
      найти самый дальний ν-удаленный ген i в C и удалить его из C
  добавить C в набор Clusters
  удалить узлы C из Graph
  return Clusters
```

Упражнение. Реализуйте **CAST** и используйте его для кластеризации сокращенного набора данных об экспрессии генов.

Библиографические примечания

Мягкий алгоритм *k*-средних для кластеризации, разработанный Bezdek, 1981[1], представляет собой вариант алгоритма максимизации ожидания, который был впервые предложен Ceppellini, Siniscalco, and Smith, 1955[2], и много раз заново открывался различными исследователями. Do and Batzoglou, 2008[3], написали отличный учебник по максимизации ожиданий, который вдохновил нас на обсуждение подбрасывания монеты. Алгоритм Ллойда для кластеризации методом *k*-средних был введен Lloyd, 1982[4]. Arthur and Vassilvitskii, 2007[5], разработали инициализацию *k*-means++ для кластеризации *k*-средних. Алгоритм CAST был разработан Ben-Dor, Shamir, and Yakhini, 1999[6].

DeRisi, Iyer, and Brown, 1997[7], выполнили первый крупномасштабный эксперимент по экспрессии генов для анализа диауксического сдвига (см. Cristianini

[1] https://www.researchgate.net/publication/233932672_Pattern_Recognition_With_Fuzzy_Objective_Function_Algorithms.
[2] https://www.ncbi.nlm.nih.gov/pubmed/13268982.
[3] https://www.nature.com/articles/nbt1406.
[4] https://ieeexplore.ieee.org/document/1056489.
[5] https://dl.acm.org/doi/10.5555/1283383.1283494.
[6] https://www.ncbi.nlm.nih.gov/pubmed/10582567.
[7] https://www.ncbi.nlm.nih.gov/pubmed/9381177.

and Hahn, 2007[1], где представлен превосходный анализ этого эксперимента). Eisen et al., 1998[2], описали первые применения иерархической кластеризации для анализа экспрессии генов. Alon et al., 1999[3], проанализировали паттерны экспрессии генов в опухолях толстой кишки.

Ohno, 1970[4], предложил модель полногеномной дупликации. Wolfe and Shields, 1997[5], представили первые убедительные аргументы в пользу полногеномной дупликации у дрожжей. Kellis, Birren, and Lander, 2004[6], предоставили дополнительные доказательства полногеномной дупликации путем анализа различных видов дрожжей. Однако эти аргументы не убедили Martin et al., 2007[7], опубликовавших опровержение. Thomson et al., 2005[8], восстановили последовательность древних алкогольдегидрогеназ дрожжей.

[1] https://www.cambridge.org/core/books/introduction-to-computational-genomics/863C62220C06825CE3B8F8E462D0390F.
[2] https://www.ncbi.nlm.nih.gov/pubmed/9843981.
[3] https://www.ncbi.nlm.nih.gov/pubmed/10359783.
[4] https://onlinelibrary.wiley.com/doi/abs/10.1002/tera.1420090224.
[5] https://www.ncbi.nlm.nih.gov/pubmed/9192896.
[6] https://www.ncbi.nlm.nih.gov/pubmed/15004568.
[7] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2134927/.
[8] https://www.ncbi.nlm.nih.gov/pubmed/15864308.

Глава 9

Как мы обнаруживаем локацию болезнетворных мутаций?

Комбинаторное выравнивание

Что вызывает синдром Одо?

Около 1 % детей рождается с умственной отсталостью, но это заболевание остается малоизученным, поскольку может быть вызвано целым рядом различных генетических нарушений. Одним из таких расстройств является **синдром Одо**, который вызывает невыразительное, «маскообразное» лицо. В 2011 году биологи решили генетическую загадку, лежащую в основе синдрома Одо, обнаружив несколько мутаций, общих для нескольких пациентов, которые исследователи использовали для идентификации единственной мутации, усекающей белок, ответственной за синдром Одо.

Открытие первопричины синдрома Одо представляет собой лишь одно из многих новых открытий, связанных с использованием **картирования ридов** для изучения генетических нарушений. При картировании ридов исследователи сравнивают секвенированные риды ДНК, взятые у человека, с **эталонным геномом человека**, чтобы найти, какие риды полностью соответствуют эталону, а какие риды указывают на мутации одного нуклеотида в другой (**однонуклеотидные полиморфизмы**, или **SNP**). Для получения дополнительной информации см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Эталонный геном человека**.

Эталонный геном является грубым упрощением видовой идентичности, поскольку в дополнение примерно к 3 млн SNP (0,1 % генома человека) люди различаются геномными рекомбинациями, вставками и делециями, которые могут охватывать тысячи нуклеотидов. Для получения дополнительной информации см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Рекомбинации, вставки и делеции в геномах человека**. Однако в этой главе мы сосредоточимся только на алгоритмах поиска SNP.

Но подождите, вы можете сказать, *почему бы не использовать один из алгоритмов, которые мы уже рассмотрели?* В конце концов, мы всегда можем собрать весь геном человека, а затем сравнить его с эталонным геномом. Однако методы секвенирования требуют значительных вычислительных ресурсов и несовершенны, поскольку они часто генерируют контиги, содержащие ошибки. В результате имеет смысл картировать риды отдельного человека с эталонным геномом человека, чтобы выяснить различия.

Чтобы понять, почему картирование ридов проще, чем сборка генома, давайте вернемся к аналогии с пазлом, который продается с изображением собранной картинки на коробке. Эта картинка значительно упрощает сборку пазла; для простого примера, если собранный пазл показывает солнце на голубом небе, вы можете автоматически переместить все ярко-желтые части и все светло-голубые части в верхнюю часть пазла.

Тем не менее, помимо секвенирования генома, на ум приходят два других метода картирования ридов. Во-первых, вы можете сопоставить каждый рид с эталонным геномом (используя подходящее выравнивание), чтобы найти наиболее похожую область. Во-вторых, вы можете применять алгоритмы приблизительного сопоставления паттернов, чтобы сопоставлять каждый рид по очереди с эталонным геномом.

ОСТАНОВИТЕСЬ и задумайтесь. Какие вычислительные проблемы могут возникнуть при использовании этих методов для картирования миллионов ридов с эталонным геномом человека?

Оба эти метода гарантированно решат задачу картирования ридов с эталонным геномом, но время их выполнения становится узким местом, когда мы изменяем масштаб до миллионов ридов. Поэтому наша цель в этой главе – выяснить, как использовать эталонный геном в качестве «фото на коробке пазла» для поиска SNP.

Введение во множественное выравнивание последовательностей

Из главы о сборке генома мы помним, что риды обычно имеют длину в несколько сотен пар оснований. Эти риды формируют набор строк *Patterns*, которые мы хотим сопоставить с текстом генома. Для каждой строки в *Patterns* мы сначала найдем все ее точные совпадения как подстроку *Text* (или сделаем вывод, что она не появляется в *Text*). При поиске причины генетического нарушения мы можем сразу же исключить из рассмотрения участки референсного генома, где встречаются точные совпадения. В эпилоге мы обобщим эту задачу, чтобы найти приблизительные совпадения, где одиночные нуклеотидные замены в ридах отделяют индивидуума от эталонного генома (или представляют ошибки в ридах).

Задача множественного выравнивания последовательностей: *найти все вхождения набора паттернов в текст.*

Input: строка *Text* и набор *Patterns*, содержащий (более короткие) строки.

Output: все начальные позиции в *Text*, где строка из *Patterns* появляется как подстрока.

Наивный подход к задаче множественного выравнивания последовательностей предполагает повторные применения алгоритма для (единичной) задачи выравнивания паттернов, с которой мы столкнулись при поиске источника репликации в бактериальных геномах. Этот алгоритм, который мы называем **BruteForcePatternMatching**, будет перемещать каждый паттерн вдоль по тексту, проверяя, соответствует ли подстрока, начинающаяся в каждой позиции текста, паттерну. Напомним, что время выполнения простого алгоритма для

одного паттерна равно $O(|Text| \cdot |Pattern|)$. Таким образом, общее время выполнения **BruteForcePatternMatching** для задачи множественного выравнивания последовательностей равно $O(|Text| \cdot |Patterns|)$, где $|Text|$ – это длина *Text* и $|Patterns|$ – это сумма длин всех строк в *Patterns*.

Задача с применением **BruteForcePatternMatching** для выравнивания ридов заключается в том, что $|Text|$ и $|Patterns|$ – чрезвычайно велики. В случае генома человека (3 Гб) общая длина всех ридов может превышать 1 Тб; в результате любой алгоритм со временем выполнения $O(|Text| \cdot |Patterns|)$ будет слишком долгим.

ОСТАНОВИТЕСЬ и задумайтесь. Оценка $O(|Text| \cdot |Patterns|)$ представляет время выполнения алгоритма **BruteForcePatternMatching** в наихудшем случае. А каково время в *усредненном случае*?

Объединение Patterns в префиксное дерево

Построение префиксного дерева Trie

Причина, по которой время выполнения **BruteForcePatternMatching** настолько велико, заключается в том, что каждая строка в *Patterns* должна независимо проходить весь *Text*. Если вы думаете о *Text* как о длинной дороге, то **BruteForcePatternMatching** аналогичен загрузке каждого паттерна в свою машину при движении по *Text*, что является неэффективной стратегией. Вместо этого наша цель состоит в том, чтобы собрать паттерны в автобус, чтобы нам нужно было совершить только одну поездку от начала до конца *Text*. Говоря более формально, мы хотели бы организовать *Patterns* в структуру данных, чтобы предотвратить многократные проходы по *Text* и сократить время выполнения. С этой целью мы объединим *Patterns* в **«префиксное дерево»**, также называемое **«цифровое дерево»**, **«попытка»** или «**trie**». («Prefix Trie» переводится как «префиксная попытка»; так как она имеет структуру дерева, обычно используется термин «префиксное дерево». Однако, в дальнейшем мы столкнемся с терминами «Suffix Trie» и «Suffix Tree», которые переводятся как «суффиксная попытка» и «суффиксное дерево» соответственно. – *Прим. ред.*) *Trie*(*Patterns*) имеет следующие свойства (рис. 9.1).

- В префиксном дереве есть один корневой узел со степенью вхождения 0, обозначаемый корнем; все остальные узлы имеют степень входа 1.
- Каждое ребро *Trie*(*Patterns*) помечено буквой алфавита.
- Ребра, выходящие из данного узла, имеют разные обозначения.

- Каждая строка в *Patterns* записывается путем объединения букв по некоторому пути от корня вниз.
- Каждый путь от корня к **листу** или узлу с исходящей степенью 0 представляет собой строку из *Patterns*.

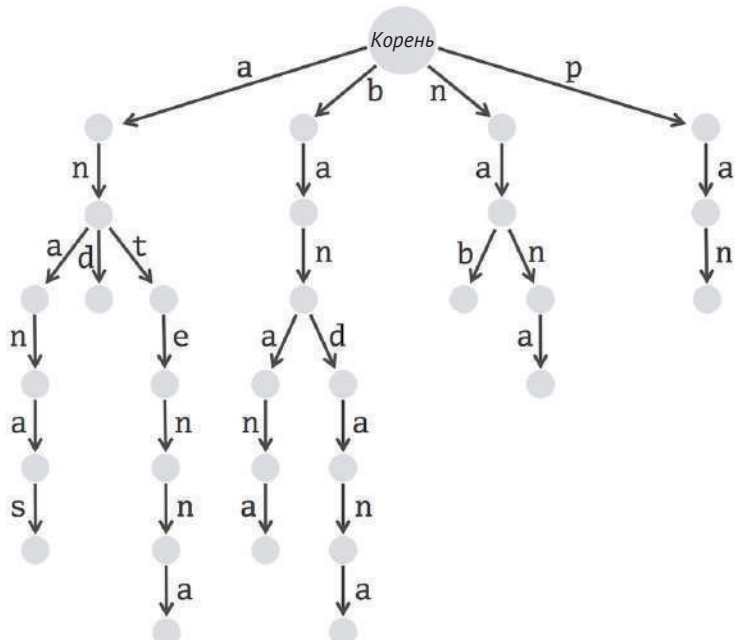


Рис. 9.1 Префиксное дерево для следующих *Patterns*: «ananas», «and», «antenna», «banana», «bandana», «nab», «nana», «pan»

Задача построения префиксного дерева: *построить префиксное дерево из набора паттернов*.

Input: набор строк *Patterns*.

Output: *Trie*(*Patterns*).

Самый очевидный способ построения *Trie*(*Patterns*) – это итеративное добавление каждой строки из *Patterns* в растущее префиксное дерево, как это реализовано с помощью следующего алгоритма.

```
TrieConstruction(Patterns)
    Trie ← граф, состоящий из единственного узла root
    for каждой строки Pattern в Patterns
        currentNode ← root
```

```
    for i ← 0 до|Pattern| – 1
        currentSymbol ← Pattern[i]
        if существует исходящее ребро от currentNode с обозначением
        currentSymbol
            currentNode ← конечный узел этого ребра
        else
            добавить новый узел newNode к Trie
            добавить новое ребро из currentNode к newNode с обозначением
            currentSymbol
            currentNode ← newNode
    return Trie
```

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем использовать префиксное дерево для решения задачи множественного выравнивания последовательностей?

Применение префиксного дерева к множественному выравниванию

Имея строку *Text* и *Trie*(*Patterns*), мы можем быстро проверить, соответствует ли какая-либо строка из *Patterns* префиксу *Text*. Для этого мы начинаем читать символы с начала *Text* и смотрим, на какую строку эти символы «реагируют» по мере продвижения по пути вниз от корня дерева, как показано на рис. 9.2. Для каждого нового символа в тексте если мы встречаем этот символ вдоль ребра, ведущего вниз от текущего узла, то продолжаем движение по этому ребру; в противном случае мы останавливаемся и делаем вывод, что ни одна строка в *Patterns* не соответствует префиксу *Text*. Если мы пройдем весь путь до листа, то последовательность, указанная этим путем, будет соответствовать префиксу *Text*.

Описанный алгоритм называется **PrefixTrieMatching**.

```
PrefixTrieMatching(Text, Trie)
    symbol ← первая буква текста
    υ ← корень Trie
    while до конца
        if υ является листом Trie
            output Pattern, написанный путем от корня до υ
        else if в Trie есть ребро (υ, ω), помеченное символом symbol
            symbol ← следующая буква Text
            υ ← ω
        else
            return «совпадений не найдено»
```

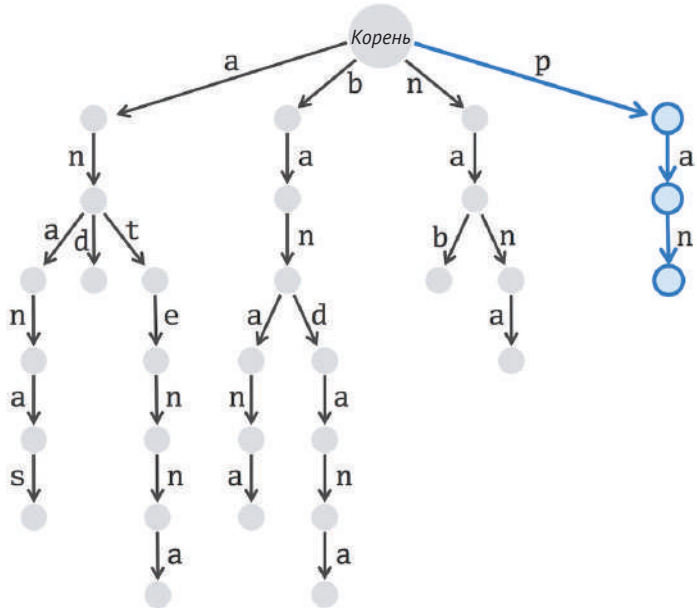


Рис. 9.2 Паттерн «pan» соответствует *Text* = «**pan**amabananas» при запуске в начале *Text*

ОСТАНОВИТЕСЬ и задумайтесь. Чтобы **PrefixTrieMatching** работал, мы сделали скрытое предположение, что ни одна строка в *Patterns* не является префиксом другой строки в *Patterns* (или, если уж на то пошло, длиннее, чем *Text*). Как это можно изменить, если *Patterns* – это произвольный набор строк? Подсказка: рассмотрите возможность добавления «кладовой» к паттернам в дереве на рис. 9.2.

PrefixTrieMatching определяет, соответствуют ли какие-либо строки в *Patterns* префиксу *Text*. Чтобы определить, соответствуют ли какие-либо строки в *Patterns* подстроке *Text*, начинающейся с позиции *i*, этот алгоритм должен сопоставить префикс суффикса *Text*, начинающегося с позиции *i*. Поэтому мы можем запустить **PrefixTrieMatching** для всех суффиксов *Text*. В результате, чтобы решить задачу множественного выравнивания последовательностей, мы просто итерируем **PrefixTrieMatching**|*Text*| раз, обрезая первый символ текста перед каждой новой итерацией (рис. 9.3).

```
TrieMatching(Text, Trie)
  while Text не пустой
    PrefixTrieMatching(Text, Trie)
    remove первый символ из Text
```

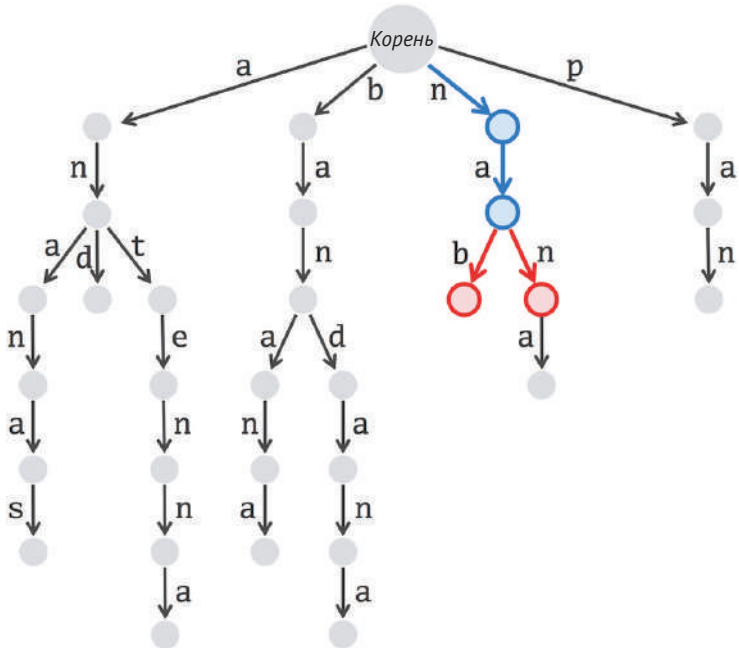


Рис. 9.3 В *Trie*(*Patterns*) не найдено совпадения с паттерном при запуске с третьего символа *Text* = «panamabananas»

Нам нужны шаги |*Patterns*| для создания *Trie*(*Patterns*), которое содержит не более |*Patterns*|+1 узлов. Каждая итерация **PrefixTrieMatching** занимает не более |*LongestPattern*| шагов, где *LongestPattern* – самая длинная строка в *Patterns*. **TrieMatching** делает |*Text*| общих вызовов **PrefixTrieMatching**, что делает общее количество шагов равным |*Patterns*| + |*Текст*|·|*LongestPattern*|. Эта рутина дает, по сравнению с |*Text*|·|*Patterns*|, большую скорость шагов, необходимых для **BruteForcePatternMatching**. Алгоритм Ахо–Корасик, разработанный в 1975 году, дополнительно сокращает количество шагов, необходимых после построения префиксного дерева, с *O*(|*Text*|·|*LongestPattern*|) до *O*(|*Text*|) шагов. Для получения дополнительной информации см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Алгоритм Ахо–Корасик**.

ОСТАНОВИТЕСЬ и задумайтесь. Видите ли вы какие-либо вычислительные проблемы с использованием **TrieMatching** для решения задачи множественного выравнивания последовательностей?

Несмотря на то что **TrieMatching** работает быстро, хранение префиксного дерева потребляет много памяти. Напомним, что **BruteForcePatternMatching** работает с одним ридом за раз, что снижает объем памяти, поскольку нам нужно хранить в памяти только геном. Тем не менее **TrieMatching** должен хранить в памяти все дерево, что пропорционально объему |*Patterns*|. Поскольку коллекция ридов для генома человека может занимать более 1 Тб, память, необходимая для хранения дерева, непомерно высока.

ОСТАНОВИТЕСЬ и задумайтесь. Как избежать многократного прохождения генома без необходимости объединять все риды в огромную структуру данных?

Предварительная обработка генома как альтернатива

Суффиксные попытки (suffix tries)

Поскольку для хранения *Trie*(*Patterns*) требуется слишком много памяти, давайте вместо этого преобразуем *Text* в структуру данных. Наша цель – сравнить каждую строку в *Patterns* с *Text* без необходимости проходить *Text* от начала до конца. Говоря более привычным языком, вместо того, чтобы упаковывать *Patterns* в автобус и ехать на большое расстояние вниз по *Text*, наша новая структура данных сможет «телепортировать» каждую строку *Patterns* непосредственно к ее вхождениям в *Text*.

«Суффиксная попытка», дерево суффиксов, обозначаемое как *SuffixTrie*(*Text*), представляет собой дерево, сформированное из всех суффиксов *Text* (рис. 9.4). С этого момента мы добавляем знак «$» к тексту, чтобы отметить конец текста (в этом выборе символа нет ничего особенного). Мы также пометим каждый лист результирующего дерева начальной позицией суффикса, путь которого через дерево заканчивается на этом листе (используя индексацию на основе 0). Таким образом, когда мы дойдем до листа, мы сразу узнаем, откуда взялся этот суффикс в тексте.

Упражнение. Создайте суффикс для слова «papa», не добавляя сначала знак $, чтобы обозначить конец текста. Где пути, соответствующие каждому суффиксу «papa»? Теперь вы понимаете, почему мы сначала добавляем «$» в конец текста? (Подсказка: попробуйте найти суффикс «pa» в своем дереве.)

Рис. 9.4 *SuffixTrie*(«panamabananas$»), метки листьев (соответствующие начальным позициям суффиксов) варьируются от 0 до 13

Использование суффиксных попыток для сопоставления последовательностей

Чтобы сопоставить одну строку *Pattern* с *Text*, обратите внимание на то, что если образец соответствует подстроке *Text*, начинающейся с позиции *i*, то *Pattern* также должен появиться в начале суффикса *Text*, начинающегося с позиции *i*. Поэтому мы можем определить, встречается ли *Pattern* в *SuffixTrie*(*Text*), начав с корня и написав символы *Pattern* вниз. Если мы можем найти путь в суффик-

се, обозначающем *Pattern*, то мы знаем, что *Pattern* должен встречаться в тексте (рис. 9.5). Затем можем перебрать все строки в *Patterns*.

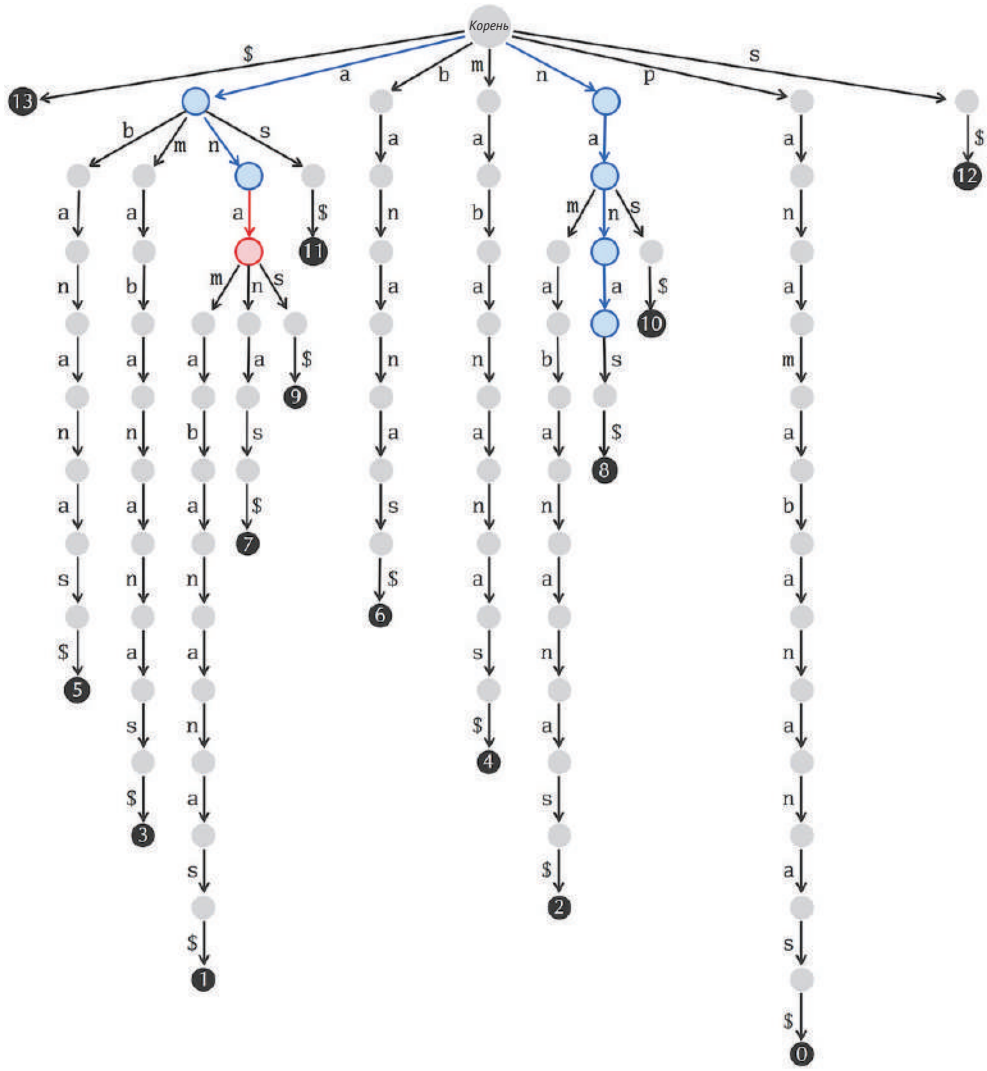


Рис. 9.5 Обработка слова «antenna» через *SuffixTrie*(«panamabananas$») не дает совпадения, так как суффикс «panamabananas$$» не начинается с «**ant**»; тем не менее прохождение «nanas» через дерево суффиксов находит совпадение: «panamaba**nanas$$**»

ОСТАНОВИТЕСЬ и задумайтесь. Посмотрите на это *SuffixTrie* еще раз. Оно показывает, как найти «nanas» в «panamabananas$», но не говорит нам, где встречается в тексте «nanas». Как нам получить эту информацию?

Чтобы определить, где *Pattern* появляется в тексте, сначала предположим, что *Pattern* соответствует *Pattern* в листе *SuffixTrie*(*Pattern*). В этом случае *Pattern* должен появиться в *Text* как суффикс, и мы можем обратиться к метке на этом листе, чтобы определить начальную позицию суффикса. Например, вставка «nanas» в *SuffixTrie*(«panamabananas$»), воспроизведенная на рис. 9.5, показывает, что этот паттерн соответствует суффиксу, начинающемуся с позиции 8 слова «panamaba**nanas**$».

Если путь, следующий по *Pattern*, останавливается перед листом в некотором узле v *SuffixTrie*(*Text*), тогда *Pattern* может встретиться в *Text* более одного раза. Чтобы найти эти совпадения, следуйте по всем путям от v вниз до листьев *SuffixTrie*(*Text*), которые будут указывать все начальные позиции *Pattern* в *Text*. Например, «ana» соответствует пути в *SuffixTrie*(«panamabananas$»), заканчивающемуся на внутреннем узле, как показано на рис. 9.6. Этот путь можно расширить до трех разных листьев с метками 1, 7 и 9, соответствующих трем вхождениям «ana»: «p**ana**mabananas$», «panamab**ana**nas$» и «panamaban**ana**s$».

ОСТАНОВИТЕСЬ и задумайтесь. Сколько времени выполнения и памяти потребуется для построения *SuffixTrie*(*Text*)?

Напомним, что создание *Trie*(*Patterns*) требовало $O(|Patterns|)$ времени выполнения и памяти. Соответственно, время выполнения и память, необходимые для построения *SuffixTrie*(*Text*), равны общей длине всех суффиксов в *Text*. Есть $|Текст|$текстовых суффиксов длиной от 1 до $|Text|$ и имеющих общую длину $|Text| \cdot (|Text| + 1)/2$, что равно $O(|Text|^2)$. Таким образом, нам нужно сократить как время построения, так и требования к памяти суффиксных попыток *SuffixTrie*, чтобы сделать их практичными.

Суффиксные деревья (suffix trees)

Не будем терять надежду на суффиксные попытки. Мы можем уменьшить количество ребер в *SuffixTrie*(*Text*), объединив ребра на любом неветвящемся пути в одно ребро. Затем мы помечаем это ребро конкатенацией символов на консолидированных ребрах, как показано на рис. 9.7. Результирующая структура данных называется **суффиксным деревом**, или *SuffixTree*(*Text*).

Чтобы сопоставить один *Pattern* с *Text*, мы вставляем *Pattern* в *SuffixTree*(*Text*) с помощью того же процесса, который используется для суффиксных попыток. Как и в случае с ними, мы можем использовать метки листьев, чтобы найти начальные позиции успешно совпадающих паттернов.

Упражнение. Докажите, что *SuffixTree*(*Text*) имеет точно $|Text|$ листьев и не более |Текст| других узлов.

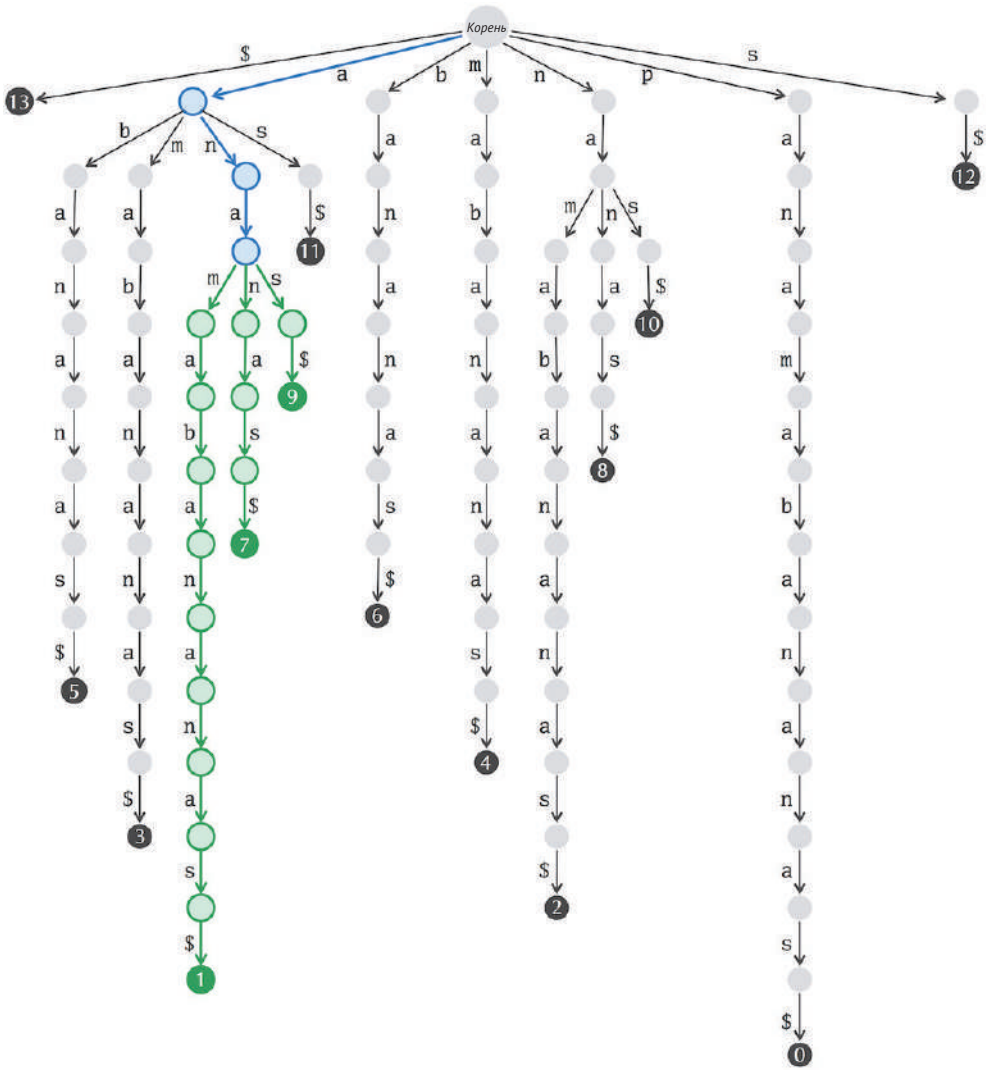


Рис. 9.6 Все пути, начинающиеся с «ana», показывают три начальных позиции «ana» в «p**ana**mab**anana**s$». Продолжение этих путей до листьев (показанных зеленым цветом) показывает, что начальные позиции этих вхождений – 1, 7 и 9

По сравнению с суффиксной попыткой, которая может иметь квадратичное число узлов на протяжении *Text*, число узлов в *SuffixTree*(*Text*) не превышает 2·|*Text*|. Следовательно, для *SuffixTree*(*Text*) требуется память всего лишь *O*(|*Text*|).

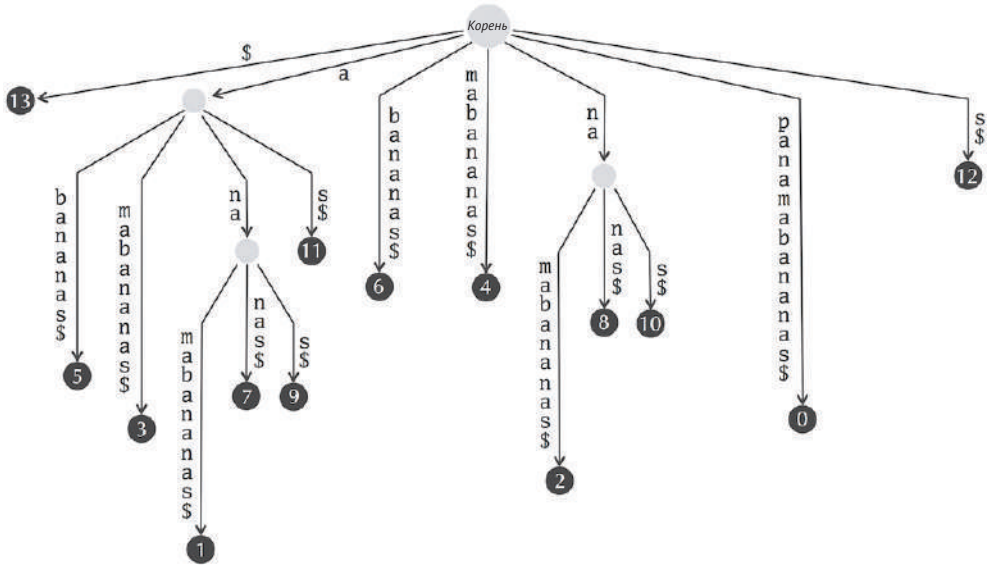


Рис. 9.7 *SuffixTree*(«panamabananas$»)

ОСТАНОВИТЕСЬ и задумайтесь. Подождите секунду. Суффиксные деревья *Suffix Trees* кажутся косметической модификацией суффиксных деревьев *Suffix Trie*. Поскольку нам по-прежнему необходимо хранить в памяти все конкатенированные метки ребер, почему *Suffix Trees* должны быть гораздо более эффективными с точки зрения использования памяти, чем *Suffix Trie*?

Суффиксные деревья *Suffix Trees* экономят память, потому что им не нужно хранить конкатенированные метки ребер из каждого неветвящегося пути. Например, суффиксному дереву не требуется десять байтов для хранения ребра, помеченного «mabananas$» в *SuffixTree*(«panamabananas$») (воспроизведено ниже); вместо этого достаточно сохранить указатель на позицию 4 «panamabananas$», а также длину «mabananas$». Кроме того, суффиксные деревья можно построить за линейное время без необходимости сначала строить *Suffix Trie*! Мы не будем просить вас реализовать этот алгоритм быстрого построения суффиксного дерева, потому что он довольно сложный.

Задача построения суффиксного дерева: *построить суффиксное дерево строки.*

Input: строка *Text*.

Output: *SuffixTree*(*Text*).

Мы надеемся, что вам интересно, как суффиксное дерево может экономить память, поскольку оно должно хранить потенциально очень длинные строки на каждом ребре. Было бы неэффективно с точки зрения использования памяти сначала построить суффиксную попытку, а затем объединить каждый неветвящийся путь в одно ребро, сохраняя его метку в памяти. На практике исследователи реализуют суффиксное дерево, сохраняя указатели на места в тексте, а не сохраняя потенциально длинные подстроки. Чтобы увидеть, как эта идея может быть реализована, посмотрите раздел **ЗАРЯДНАЯ СТАНЦИЯ: Построение суффиксного дерева**.

Хотя суффиксное дерево снижает требования к памяти с $O(|Text|^2)$ до $O(|Text|)$, в среднем оно по-прежнему требует примерно в 20 раз больше памяти, чем *Text* (когда длина *Text* порядка длины генома человека). Для человеческого генома размером 3 Гб оперативная память компьютера в 60 Гб – это огромное улучшение по сравнению с 1 Тб, который нам нужен для работы с попыткой, созданной для всех ридов, но это по-прежнему представляет собой проблему с памятью для большинства машин. Это раскрывает темную тайну «*O* большого», которая заключается в том, что оно игнорирует постоянные множители. Для длинных строк, таких как геном человека, нам нужно будет обратить внимание на этот постоянный фактор, поскольку выражение $O(|Text|)$ применимо как к алгоритму с памятью $2 \cdot |Text|$, так и алгоритму с памятью $1000 \cdot |Text|$.

Тем не менее, прежде чем увидеть, как мы можем еще больше сократить объем памяти, необходимый для множественного сопоставления паттернов, мы просим вас решить три задачи, показывающие, как суффиксные деревья можно применять к другим задачам сопоставления паттернов. Первая такая задача – это задача с самым длинным повтором.

Задача на самый длинный повтор: *найти самый длинный повтор в строке.*

Input: строка *Text*.

Output: самая длинная подстрока *Text*, которая появляется в *Text* более одного раза.

Второе дополнительное упражнение, которое мы рассмотрим, – это задача о самой длинной общей подстроке.

Задача на самую длинную общую подстроку: *найти самую длинную подстроку, общую для двух строк.*

Input: строки $Text_1$ и $Text_1$.

Output: самая длинная подстрока, встречающаяся как в $Text_1$, так и в $Text_2$.

Одним из способов решения задачи самой длинной общей подстроки может быть построение двух деревьев суффиксов, одного для $Text_1$ и одного для $Text_1$. Посмотрите раздел **ЗАРЯДНАЯ СТАНЦИЯ: решение задачи на самую длинную общую подстроку**, чтобы узнать о более элегантном решении.

Наконец, мы просим вас решить задачу о самой короткой подстроке, не являющейся общей.

Задача о самой короткой подстроке, не являющейся общей:
найти самую короткую подстроку одной строки, которая не встречается в другой строке.

Input: строки $Text_1$ и $Text_2$.

Output: самая короткая подстрока $Text_1$, которая не появляется в $Text_2$.

Суффиксные массивы

В 1993 году Уди Манбер и Джин Майерс представили **суффиксные массивы** как альтернативу суффиксным деревьям по эффективности использованиея памяти. Чтобы построить *SuffixArray*(*Text*), мы сначала отсортируем все суффиксы в *Text* лексикографически, предполагая, что «$» стоит первым в алфавите. Например, ниже приведен отсортированный список суффиксов *Text* = «panamabananas$» вместе с их начальными позициями в *Text*.

Суффиксный массив (рис. 9.8) представляет собой список начальных позиций этих отсортированных суффиксов:

SuffixArray(«panamabananas$») = (13, 5, 3, 1, 7, 9, 11, 6, 4, 2, 8, 10, 0, 12).

Стартовая позиция	Отсортированные суффиксы
13	$
5	abananas$
3	amabananas$
1	anamabananas$
7	ananas$
9	anas$
11	as$
6	bananas$
4	mabananas$
2	namabananas$
8	nanas$
10	nas$
0	panamabananas$
12	s$

Рис. 9.8 Отсортированный список суффиксов *Text* = «panamabananas$» вместе с их начальными позициями в *Text*; эти начальные позиции образуют суффиксный массив текста

Задача построения суффиксного массива: *построить суффиксный массив строки.*

Input: строка *Text*.

Output: *SuffixArray*(*Text*) в виде набора целых чисел, разделенных пробелами.

Задача построения суффиксного массива может быть легко решена после сортировки всех суффиксов текста, но, поскольку даже самые быстрые алгоритмы сортировки массива из n элементов требуют $O(n \cdot \log n)$ сравнений, сортировка всех суффиксов занимает $O(|Text| \cdot \log (|Text|))$ сравнений, каждое из которых занимает $O(|Text|)$ времени. Однако существует более быстрый алгоритм, который строит суффиксные массивы за линейное время (без предварительного построения суффиксного дерева) и требует примерно в пять раз меньше памяти, чем суффиксные деревья, что снижает требования к памяти с 60 Гб для генома человека до 12 Гб.

ОСТАНОВИТЕСЬ и задумайтесь. Если у вас есть суффиксное дерево, можете ли вы быстро преобразовать его в суффиксный массив? Имея суффиксный массив, можете ли вы быстро преобразовать его в суффиксное дерево?

Как следует из предыдущего вопроса, суффиксные массивы и суффиксные деревья практически эквивалентны: любой алгоритм, использующий суффиксные массивы, можно преобразовать в алгоритм, использующий суффиксные деревья, и наоборот. Дополнительные сведения см. в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Суффиксные массивы и суффиксные деревья**.

Выравнивание паттерна с суффиксным массивом

Создав суффиксный массив строки *Text*, мы можем использовать его для быстрого поиска всех вхождений строки *Pattern* в *Text*. Во-первых, напомним, что при выравнивании паттерна с суффиксной попыткой мы заметили, что все совпадения *Pattern* в *Text* должны происходить в начале суффиксов *Text*. Во-вторых, обратите внимание, что после сортировки суффиксов *Text* суффиксы, начинающиеся с *Pattern*, группируются вместе. Например, в рисунке суффиксного массива для *Text* = «panamabananas$» (рис. 9.8) *Pattern* = «ana» встречается в начале суффиксов «anamabananas$», «ananas$» и «anas$»; эти суффиксы встречаются в трех последовательных строках и соответствуют начальным позициям 1, 7 и 9 в тексте.

Вопрос в том, как найти эти начальные позиции для произвольной строки *Pattern* без необходимости сохранять отсортированные суффиксы *Text*. Следу-

ющий алгоритм, называемый **PatternMatchingWithSuffixArray**, идентифицирует первый и последний индекс суффиксного массива, соответствующие суффиксам, начинающимся с *Pattern*. Эти индексы обозначаются как *first* (первый) и *last* (последний) соответственно. Он предлагает разновидность общего метода поиска, называемого **бинарным поиском**, который находит точку данных в отсортированном наборе данных путем итеративного деления данных пополам и определения половины, в которой находится точка данных (подробнее см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Бинарный поиск**).

В приведенном ниже псевдокоде используется обозначение $String_1 < String_2$, чтобы указать, что $String_1$ лексикографически меньше, чем $String_2$. Мы также говорим, что *Pattern* соответствует суффиксу *Text*, если первые |*Pattern*| символы этого суффикса соответствуют *Pattern*. Обратите внимание, что все суффиксы текста отличаются друг от друга и от *Pattern* (который не содержит знака «$»). В результате первый цикл **while** в псевдокоде устанавливает *minIndex* = *maxIndex* + 1, так что *Pattern* находится «между» суффиксами, соответствующими *SuffixArray*(*maxIndex*) и *SuffixArray*(*minIndex*). То есть выполняются следующие два свойства:

суффикс из *Text*, начинающийся с позиции *SuffixArray*(*maxIndex*) < *Pattern*;
суффикс из *Text*, начинающийся с позиции *SuffixArray*(*minIndex*) > *Pattern*.

```
PatternMatchingWithSuffixArray(Text, Pattern, SuffixArray)
  minIndex ← 0
  maxIndex ← |Text| − 1
  while minIndex ≤ maxIndex
    midIndex ← ⌊(minIndex + maxIndex)/2⌋
    if Pattern > суффикс строки Text, начиная с позиции SuffixArray(midIndex)
      minIndex ← midIndex + 1
    else
      maxIndex ← midIndex − 1
  if Pattern соответствует суффиксу Text, начиная с позиции SuffixArray(midIndex)
    first ← midIndex
  else
    return Pattern не появляется в Text
  minIndex ← first
  maxIndex ← |Text| − 1
  while minIndex ≤ maxIndex
    midIndex ← ⌊(minIndex + maxIndex)/2⌋
    if Pattern > суффикса в Text начиная с позиции SuffixArray(midIndex)
      minIndex ← midIndex + 1
    else
      maxIndex ← midIndex − 1
  last ← maxIndex
  return (first, last)
```

Преобразование Барроуза–Уилера

Сжатие генома

Суффиксные массивы значительно сократили объем памяти, необходимый для эффективного поиска в текстах, и до начала этого века они представляли собой передовые достижения в области сопоставления паттернов. Можем ли мы быть настолько амбициозными, чтобы искать структуру данных, которая будет кодировать текст, используя память, приблизительно равную длине текста, и при этом обеспечивая быстрое сопоставление с паттерном?

Чтобы ответить на этот вопрос, давайте отвлечемся и рассмотрим, казалось бы, не связанную с этим тему **сжатия текста**. В одном простом методе сжатия, называемом **кодированием повторов (кодирование длинных серий)**, мы заменяем серию из k последовательных вхождений символа s только двумя символами: k, за которым следует s. Например, кодирование длинных серий сжимало бы строку TTTTTGGGAAAACCCCCCA до 5T3G4A6C1A.

Кодирование повторов хорошо работает для строк, имеющих много повторов, но геномы не имеют много серий из одного нуклеатида. Что у них есть, так это повторы, как мы узнали при сборке геномов. Поэтому было бы хорошо, если бы мы могли сначала манипулировать геномом, чтобы преобразовать повторы в серии, а затем применить кодирование длины серий к полученной строке.

Наивный способ создания серий в строке состоит в том, чтобы изменить порядок символов строки лексикографически. Например, TACGTAACGATACGAT станет AAAACCCGGGTTTT, которое затем можно будет сжать в 5A3C3G4T. Этот метод даст файл генома человека размером 3 Гб с использованием всего четырех символов.

ОСТАНОВИТЕСЬ и задумайтесь. Чего плохого в применении этого метода сжатия к геномам?

Упорядочивание символов строки лексикографически не подходит для сжатия, потому что многие разные строки будут сжаты в одну и ту же строку. Например, строки GCATCATGCAT и ACTGACTACTG, а также любые строки с одинаковым количеством нуклеотидов переупорядочиваются в AAACCCGGTTT. В результате мы не можем **распаковать** сжатую строку однозначно, т. е. инвертировать операцию сжатия, чтобы получить исходную строку.

Построение преобразования Барроуза–Уилера

Рассмотрим другой метод преобразования повторов строки в серии, предложенный Майклом Барроузом и Дэвидом Уилером в 1994 году. Во-первых, определить все возможные **циклические повороты текста**; **циклический**

поворот определяется отсечением суффикса в конце *Text* и добавлением этого суффикса в его начало. Далее – аналогично массивам суффиксов – упорядочить все циклические повороты *Text* лексикографически, чтобы сформировать матрицу символов |*Text*|×|*Text*|, которую мы называем **матрицей Барроуза–Уилера** и обозначаем *M*(*Text*). На рисунке ниже показано построение *M*(*Text*).

Циклические повороты	*M*(«panamabananas$»)
panamabananas$	$ p a n a m a b a n a n a s
$panamabananas	a b a n a n a s $ p a n a m
s$panamabanana	a m a b a n a n a s $ p a n
as$panamabanan	a n a m a b a n a n a s $ p
nas$panamabana	a n a n a s $ p a n a m a b
anas$panamaban	a n a s $ p a n a m a b a n
nanas$panamaba	a s $ p a n a m a b a n a n
ananas$panamab	b a n a n a s $ p a n a m a
bananas$panama	m a b a n a n a s $ p a n a
abananas$panam	n a m a b a n a n a s $ p a
mabananas$pana	n a n a s $ p a n a m a b a
amabananas$pan	n a s $ p a n a m a b a n a
namabananas$pa	p a n a m a b a n a n a s $
anamabananas$p	s $ p a n a m a b a n a n a

Рис. 9.9 Все циклические повороты «panamabananas$» (слева) и матрица Барроуза–Уилера *M*(«panamabananas$») всех лексикографически упорядоченных циклических поворотов (справа)

Обратите внимание, что первый столбец *M*(*Text*) содержит символы *Text*, упорядоченные лексикографически, что представляет собой наивную рекомбинацию *Text*, которую мы уже описали. В свою очередь, второй столбец *M*(*Text*) содержит вторые символы всех циклических поворотов *Text*, и поэтому он также представляет (другую) рекомбинацию символов *Text*. То же рассуждение применимо, чтобы показать, что любой столбец *M*(*Text*) представляет собой некоторую рекомбинацию символов *Text*. Нас интересует последний столбец *M*(*Text*), называемый **преобразованием Барроуза–Уилера** строки *Text*, или *BWT*(*Text*), которое показано красным на рисунке ниже.

Задача построения преобразования Барроуза–Уилера: *построить преобразование Барроуза–Уилера строки.*

Input: строка *Text*.

Output: *BWT*(*Text*).

Циклические повороты	*M*(«panamabananas$»)
panamabananas$	$ p a n a m a b a n a n a **s**
$panamabananas	a b a n a n a s $ p a n a **m**
s$panamabanana	a m a b a n a n a s $ p a **n**
as$panamabanan	a n a m a b a n a n a s $ **p**
nas$panamabana	a n a n a s $ p a n a m a **b**
anas$panamaban	a n a s $ p a n a m a b a **n**
nanas$panamaba	a s $ p a n a m a b a n a **n**
ananas$panamab	b a n a n a s $ p a n a m **a**
bananas$panama	m a b a n a n a s $ p a n **a**
abananas$panam	n a m a b a n a n a s $ p **a**
mabananas$pana	n a n a s $ p a n a m a b **a**
amabananas$pan	n a s $ p a n a m a b a n **a**
namabananas$pa	p a n a m a b a n a n a s **$**
anamabananas$p	s $ p a n a m a b a n a n **a**

Рис. 9.10 *BWT*(«panamabananas$») задается последним столбцом *M*(«panamabananas$»): «smnpbnnnaaaaa$a»

Может показаться странным, почему нас так интересует последний столбец *M*(*Text*). Почему не седьмой столбец или предпоследний столбец? Потерпите, и вы увидите, что последний столбец этой матрицы обладает уникальным свойством: вы можете реконструировать *Text* из *BWT*(*Text*). Оказывается, ни один другой столбец *M*(*Text*) не обладает этим свойством.

От повторов к сериям

Если мы повторно рассмотрим преобразование Барроуза–Уилера «panamabananas$», мы сразу же заметим, что оно создало серию «aaaaa» в *BWT*(«panamabananas») = «smnpbnnn**aaaaa**$a» (рис. 9.10).

ОСТАНОВИТЕСЬ и задумайтесь. Почему, по вашему мнению, преобразование Барроуза–Уилера выдало эту серию?

Представьте, что мы берем преобразование Барроуза–Уилера из статьи Уотсона и Крика 1953 года о структуре двойной спирали ДНК. Слово «and» часто повторяется в английском языке, а это значит, что при формировании всех возможных циклических поворотах статьи Уотсона и Крика мы увидим большое количество поворотов, начинающихся с «and…». В свою очередь, мы будем наблюдать много поворотов, которые начинаются с «nd…» и заканчиваются на «…a». Когда все циклические повороты *Text* лексикографически отсортирова-

ны для формирования *M*(*Text*), все строки, начинающиеся с «nd...» и заканчивающиеся «...a», будут иметь тенденцию слипаться. Как показано на рисунке ниже, это слипание создает ряды «a» в последнем столбце *M*(*Text*), который, как мы знаем, является *BWT*(*Text*).

```
nd Corey (1).  They kindly made their manuscript availa ..... a
nd criticism, especially on interatomic distances.  We  ..... a
nd cytosine.  The sequence of bases on a single chain d ..... a
nd experimentally (3,4) that the ratio of the amounts o ..... u
nd for this reason we shall not comment on it.  We wish ..... a
nd guanine (purine) with cytosine (pyrimidine).  In oth ..... a
nd ideas of Dr.  M. H. F. Wilkins, Dr.  R. E. Franklin  ..... a
nd its water content is rather high.  At lower water co ..... a
nd pyrimidine bases.  The planes of the bases are perpe ..... a
nd stereochemical arguments.  It has not escaped our no ..... a
nd that only specific pairs of bases can bond together  ..... u
nd the atoms near it is close to Furberg's 'standard co ..... a
nd the bases on the inside, linked together by hydrogen ..... a
nd the bases on the outside.  In our opinion, this stru ..... a
nd the other a pyrimidine for bonding to occur.  The hy ..... a
nd the phosphates on the outside.  The configuration of ..... a
nd the ration of guanine to cytosine, are always very c ..... a
nd the same axis (see diagram).  We have made the usual ..... u
nd their co-workers at King's College, London.  One of  ..... a
```

Рис. 9.11 Несколько последовательных строк, выбранных из *M*(*Text*), где *Text* – это статья Уотсона и Крика 1953 года о двойной спирали ДНК. Строки, начинающиеся с «nd...», часто заканчиваются на «...a» из-за того, что слово «and» часто встречается в английском языке, что приводит к появлению «a» в *BWT*(*Text*). Каждая строка на этом рисунке содержит длинную строку, обозначенную «...», которая не помещается на странице. Экземпляры «u» в последнем столбце соответствуют словам «round» и «found»

Подстрока «ana» в «panamabananas$» играет роль «and» в статье Уотсона и Крика и объясняет три из пяти вхождений «a» в повторении «aaaaa» в *BWT*(«panamabananas$») = «smnpbnn**aaaaa**$a». Когда преобразование Барроуза–Уилера применяется к геному, оно преобразует множество повторов генома в серии. Как мы уже предлагали, после применения преобразования Барроуза–Уилера мы можем применить дополнительный метод сжатия, такой как кодирование повторов, чтобы еще больше уменьшить объем требуемой памяти.

Первая попытка инвертирования преобразования Барроуза–Уилера

Прежде чем мы забежим вперед, вспомните, что сжатие генома не имеет большого значения, если мы не можем его однозначно распаковать. В частности, если существует пара геномов, которые преобразование Барроуза–Уилера

сжимает в одну и ту же строку, то распаковать эту строку однозначно мы не сможем. Но оказывается, что преобразование Барроуза–Уилера – обратимо!

ОСТАНОВИТЕСЬ и задумайтесь. Сможете ли вы найти (уникальную) строку, преобразование Барроуза–Уилера которой даст «enwvpeoseu$llt»? Это может быть «newtloveslupe$», «elevenplustwo$», «unwellpesovet$» или что-то совсем другое.

Рассмотрим показательный пример *BWT*(*Text*) = «ard$rcaaaabb». Во-первых, напомним, что первый столбец *M*(*Text*) – это просто лексикографическая перестановка символов в *BWT*(*Text*), т. е. «$aaaaabbcdrr». Для удобства мы в дальнейшем будем использовать сокращения *FirstColumn* и *LastColumn* (т. е. *BWT*(*Text*)) при обращении к первому и последнему столбцам *M*(*Text*) соответственно.

Мы знаем, что первая строка *M*(*Text*) представляет собой циклический поворот *Text*, начинающийся с «$», которое происходит в конце *Text*. Таким образом, если мы определяем первую строку *M*(*Text*), то можем переместить «$» в конец этой строки и воспроизвести *Text*. Но как нам определить оставшиеся символы в этой первой строке, если мы знаем только *FirstColumn* и *LastColumn*?

ОСТАНОВИТЕСЬ и задумайтесь. Используя первый и последний столбцы матрицы Барроуза–Уилера, показанной ниже, можете ли вы найти первый символ *Text*?

```
$ ? ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? b
```

Рис. 9.12 Матрица Барроуза–Уилера

Обратите внимание, что первый символ в *Text* должен следовать за «$» при любом циклическом повороте текста. Поскольку «$» встречается как четвертый символ *LastColumn* = «ard$rcaaaabb», мы знаем, что если мы перейдем на один символ вправо от конца четвертой строки *M*(*Text*), то мы «обернемся» и при-

дем к четвертому символу *FirstColumn*, который является «a» в «$aaaaabbcdrr». Следовательно, это «a» принадлежит первой позиции текста.

Рис. 9.13 Определение символа первой позиции текста

ОСТАНОВИТЕСЬ и задумайтесь. Какой символ скрывается во второй позиции *Text*?

Следуя той же логике «обтекания», следующим символом *Text* должен быть первый символ в строке *M*(*Text*), оканчивающейся на «a». Беда только в том, что пять рядов оканчиваются на «a», и мы не знаем, какой из них правильный! Если предположить, что эта «a» является седьмым символом «ard$rc**a**aaabb», то мы получим «**b**» во второй позиции *Text* (левая матрица вверху на рис. 9.14). С другой стороны, если догадываемся, что это «a» является девятым символом «ard$rcaa**a**abb», то мы получим «**c**» во второй позиции *Text* (правая матрица вверху на рис. 9.14). Наконец, если догадаемся, что это «a» является десятым символом «ard$rcaaa**a**bb», то мы получим «**d**» во второй позиции *Text* (нижняя матрица на рис. 9.14).

ОСТАНОВИТЕСЬ и задумайтесь. Взгляните еще раз на три альтернативы выше. Какой бы вы сделали выбор среди «b», «c» и «d» для второго символа *Text*?

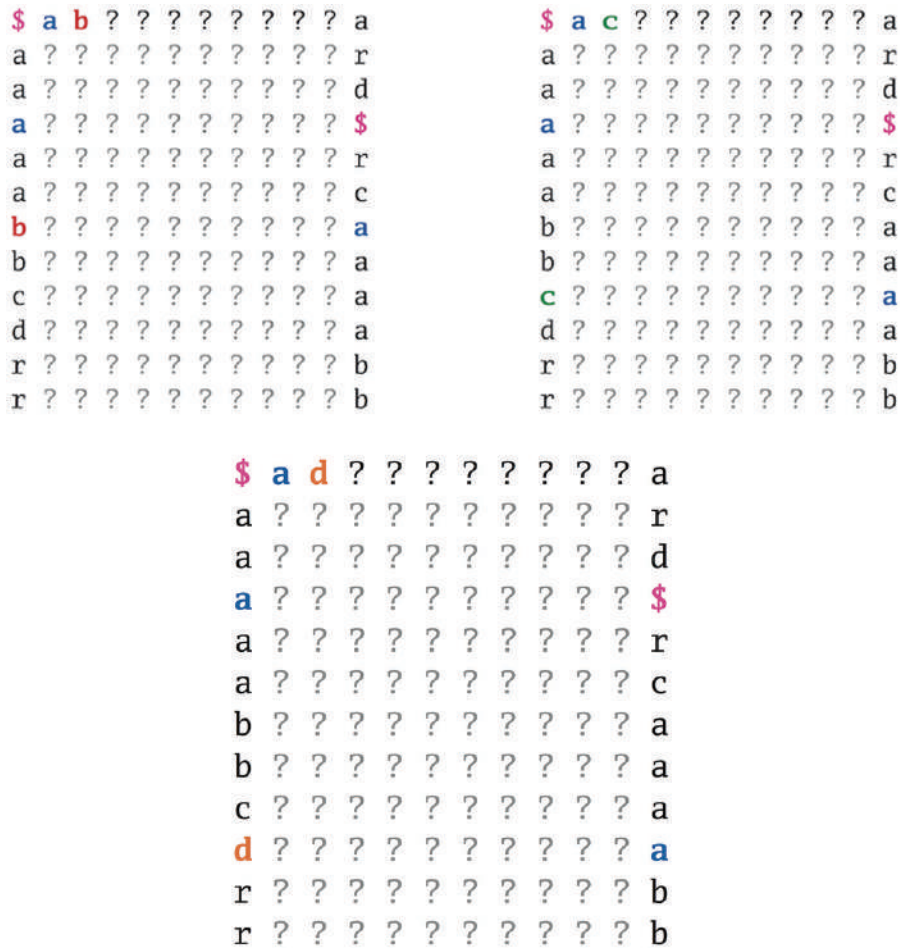


Рис. 9.14 Определение символа второй позиции текста

Свойство «первый–последний» и инвертирование преобразования Барроуза–Уилера

Свойство «первый–последний»

Чтобы определить оставшиеся символы *Text*, нам нужно использовать тонкое свойство *M*(*Text*), которое может показаться совершенно не связанным с инвертированием преобразования Барроуза–Уилера. Ниже мы проиндексирова-

ли вхождения каждого символа в *FirstColumn* с нижними индексами в соответствии с порядком их появления. Когда *Text* = «panamabananas$», в *FirstColumn* появляются шесть экземпляров «a», как показано ниже.

$ p a n a m a b a n a n a s
a_1 b a n a n a s $ p a n a m
a_2 m a b a n a n a s $ p a n
a_3 n a m a b a n a n a s $ p
a_4 n a n a s $ p a n a m a b
a_5 n a s $ p a n a m a b a n
a_6 s $ p a n a m a b a n a n
b a n a n a s $ p a n a m a
m a b a n a n a s $ p a n a
n a m a b a n a n a s $ p a
n a n a s $ p a n a m a b a
n a s $ p a n a m a b a n a
p a n a m a b a n a n a s $
s $ p a n a m a b a n a n a

Рис. 9.15 Преобразование матрицы *M*(*Text*)

Рассмотрим «$\mathbf{a}_1$» в *FirstColumn*, который появляется в начале циклической ротации «$\mathbf{a}_1$bananas$panam». Если циклически ротировать эту строку, то получится «panam$\mathbf{a}_1$bananas$». Таким образом, «$\mathbf{a}_1$» в *FirstColumn* фактически является третьим вхождением «a» в «panamabananas$». Мы можем легко определить позиции остальных пяти экземпляров «a» в «panamabananas$»:

$$p\mathbf{a}_3n\mathbf{a}_2m\mathbf{a}_1b\mathbf{a}_4n\mathbf{a}_5n\mathbf{a}_6s\$$$

ОСТАНОВИТЕСЬ и задумайтесь. Где находятся три экземпляра «n» из *FirstColumn* (т. е. «n_1», «n_2» и «n_3») в «panamabananas$»?

Чтобы найти «$\mathbf{a}_1$» в *LastColumn*, нам нужно циклически ротировать вторую строку *M* («panam$\mathbf{a}_1$bananas$»). Эта ротация преобразует «$\mathbf{a}_1$bananas$panam» в «bananas$panam$\mathbf{a}_1$», что соответствует восьмой строке матрицы.

ОСТАНОВИТЕСЬ и задумайтесь. Где находятся остальные пять экземпляров «a» в *LastColumn*?

$ p a n a m a b a n a n a s
a_1 b a n a n a s $ p a n a m
a_2 m a b a n a n a s $ p a n
a_3 n a m a b a n a n a s $ p
a_4 n a n a s $ p a n a m a b
a_5 n a s $ p a n a m a b a n
a_6 s $ p a n a m a b a n a n
b a n a n a s $ p a n a m a_1
m a b a n a n a s $ p a n a
n a m a b a n a n a s $ p a
n a n a s $ p a n a m a b a
n a s $ p a n a m a b a n a
p a n a m a b a n a n a s $
s $ p a n a m a b a n a n a

Рис. 9.16 Определение «a» в *LastColumn*

Надеюсь, вы видели, что *LastColumn* может быть записан как «smnpbnn$a_1a_2a_3a_4a_5$a_6$»:

$ p a n a m a b a n a n a s
a_1 b a n a n a s $ p a n a m
a_2 m a b a n a n a s $ p a n
a_3 n a m a b a n a n a s $ p
a_4 n a n a s $ p a n a m a b
a_5 n a s $ p a n a m a b a n
a_6 s $ p a n a m a b a n a n
b a n a n a s $ p a n a m a_1
m a b a n a n a s $ p a n a_2
n a m a b a n a n a s $ p a_3
n a n a s $ p a n a m a b a_4
n a s $ p a n a m a b a n a_5
p a n a m a b a n a n a s $
s $ p a n a m a b a n a n a_6

Рис. 9.17 *LastColumn*

Обратите внимание, что шесть экземпляров «a» появляются в точно таком же порядке в *FirstColumn* и *LastColumn*. Это наблюдение не случайность. Наоборот, этот принцип справедлив для любой строки *Text* и любого символа, который мы выбираем.

Свойство «первый–последний» (First-Last): *k*-е вхождение символа в *FirstColumn* и *k*-е вхождение этого символа в *LastColumn* соответствуют одной и той же позиции этого символа в *Text*.

Чтобы понять, почему свойство «первый–последний» должно быть истинным, рассмотрим строки *M*(«panamabananas$»), начинающиеся с «a»:

Рис. 9.18 Строки *M*(«panamabananas$»), начинающиеся с «a»

Эти строки уже упорядочены лексикографически, поэтому если мы отрежем «a» в начале каждой строки, то остальные строки все равно должны быть упорядочены лексикографически:

Рис. 9.19 Лексикографическое упорядочение остальных строк

Добавление «a» обратно в конец каждой строки не должно изменять лексикографический порядок этих строк:

Рис. 9.20 Добавление «a» обратно в конец каждой строки

Но это всего лишь строки *M*(«panamabananas$»), содержащие «a» в *LastColumn*! В результате *k*-е вхождение «a» в *FirstColumn* соответствует *k*-му вхождению «a» в *LastColumn*. Этот аргумент обобщается для любого символа и любой строки *Text*, которая устанавливает свойство «первый–последний».

Использование свойства «первый–последний» для инвертирования преобразования Барроуза–Уилера

Свойство «первый–последний» интересно, но как мы можем использовать его для инвертирования *BWT*(*Text*) = «ard$rcaaaabb»? Вернемся к тому, где мы были в нашей попытке восстановить первую строку *M*(*Text*) и проиндексировать вхождения каждого символа в *FirstColumn* и *LastColumn*:

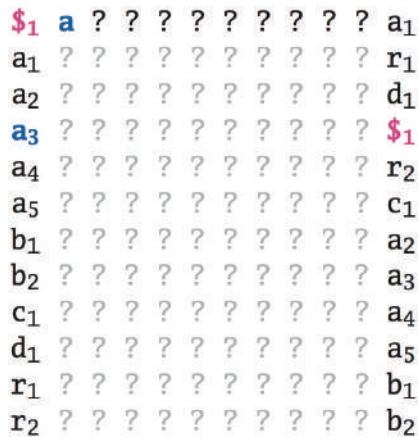


Рис. 9.21 Восстановление первой строки

Свойство «первый–последний» показывает, где в *LastColumn* скрывается «a_3»:

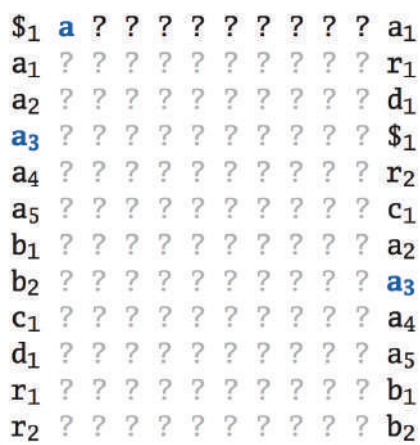


Рис. 9.22 Определение позиции «a_3»

Поскольку мы знаем, что «a_3» находится в конце восьмой строки, мы можем обвести эту строку, чтобы определить, что «b_2» следует за «a_3» в тексте. Таким образом, вторым символом *Text* является «**b**», который теперь мы можем добавить к первой строке *M*(*Text*):

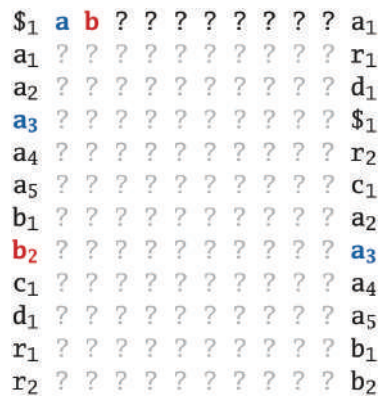


Рис. 9.23 Определение второго символа *Text*

На этом и следующем шагах мы иллюстрируем многократное применение свойства «первый–последний» для восстановления все большего количества символов из *Text*.

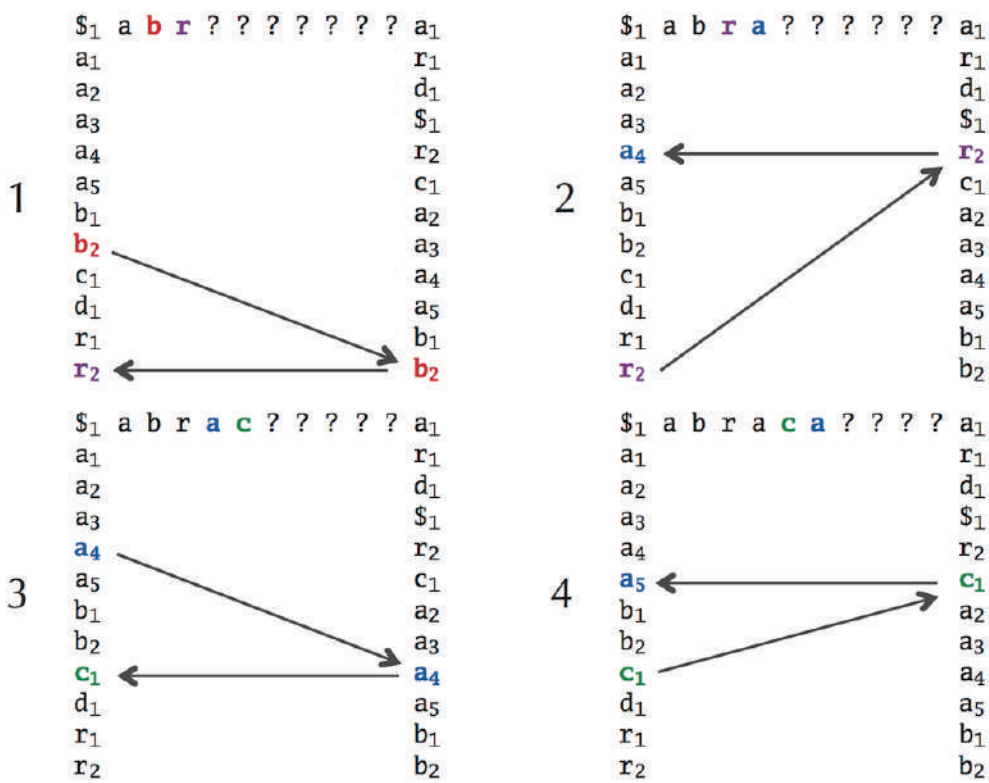


Рис. 9.24 Многократное применение свойства «первый–последний» для восстановления все большего количества символов из *Text*

Строка, которую мы пытались восстановить, – это «abracadabra$».

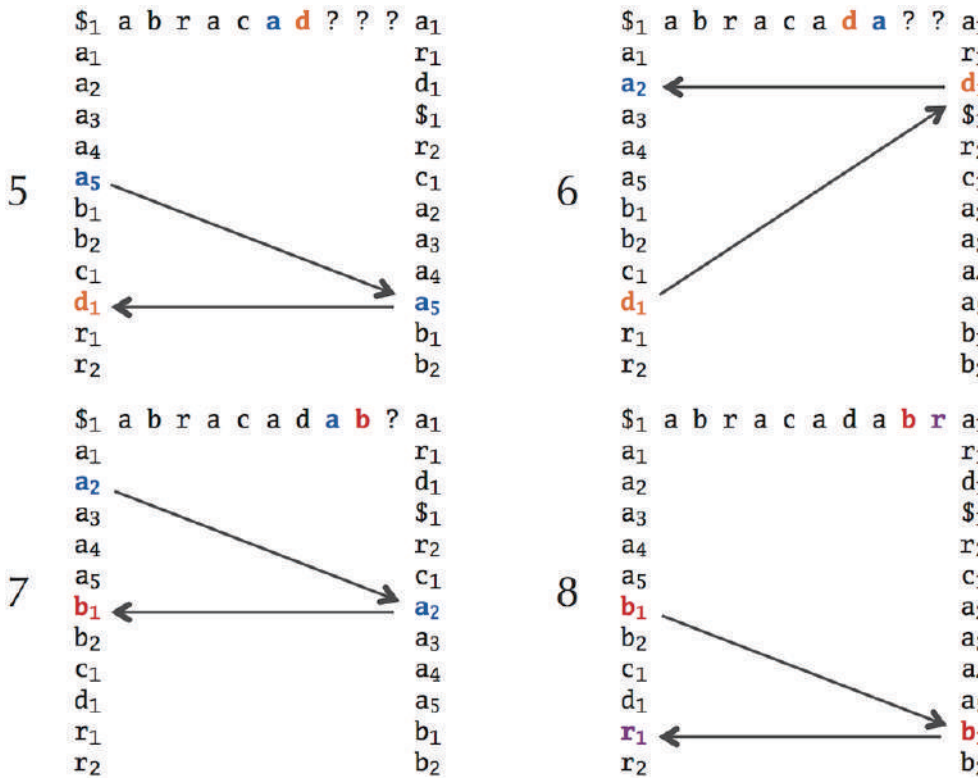


Рис. 9.25 Дальнейшее восстановление строки

Ниже приведена схема (рис. 9.26), показывающая, как мы можем использовать свойство «первый–последний», инвертируя «smnpbnnnaaaaa$a», чтобы получить *Text* = «panamabananas$».

Упражнение. Восстановите строку, преобразование Барроуза–Уилера которой равно «enwvpeoseu$llt». (Не забудьте $ в конце!)

Теперь вы готовы реализовать обратное преобразование Барроуза–Уилера.

Задача обратного преобразования Барроуза–Уилера:
восстановить строку по ее преобразованию Барроуза–Уилера.

Input: строка *Transform* (с одним символом «$»).

Output: строка *Text* такая, что *BWT*(*Text*) = *Transform*.

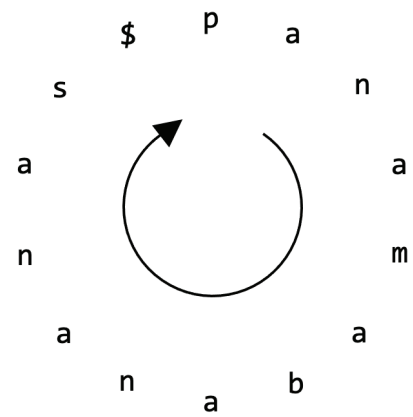


Рис. 9.26 Использование свойства «первый–последний» для получения *Text*

ОСТАНОВИТЕСЬ и задумайтесь. Можно ли инвертировать любую строку (имеющую один символ «$») с помощью обратного преобразования Барроуза–Уилера?

Сопоставление последовательностей с помощью преобразования Барроуза–Уилера

Первая попытка сопоставления паттернов Барроуза–Уилера

Преобразование Барроуза–Уилера может быть захватывающим, но как оно может помочь нам уменьшить объем памяти, необходимый для сопоставления паттернов?

Идея, лежащая в основе метода Барроуза–Уилера, основана на наблюдении, что каждая строка *M*(*Text*) начинается с другого суффикса *Text*. Поскольку эти суффиксы уже упорядочены лексикографически, любые совпадения *Pattern* в *Text* будут появляться в начале последовательных строк *M*(*Text*), как показано на рисунке ниже.

Теперь у нас есть набросок метода сопоставления *Pattern* с *Text*. Создайте *M*(*Text*) и определите строки, начинающиеся с первого символа *Pattern*. Среди этих строк определите, какие из них имеют второй элемент, соответствующий второму символу *Pattern*. Будем продолжать этот процесс, пока не найдем, какие строки *M*(*Text*) начинаются с *Pattern*.

ОСТАНОВИТЕСЬ и задумайтесь. Что не так с этим методом?

Рис. 9.27 Поскольку строки *M*(*Text*) упорядочены лексикографически, суффиксы, начинающиеся с одной и той же строки («**ana**»), в матрице слипаются

Перемещение по последовательности назад

Задача с этим предлагаемым методом сопоставления паттернов заключается в том, что мы не можем позволить себе хранить всю матрицу *M*(*Text*), которая имеет $|Text|^2$ элемента. Чтобы уменьшить требования к памяти, давайте запретим себе доступ к какой-либо информации в *M*(*Text*), кроме *FirstColumn* и *LastColumn*. Используя эти два столбца, мы попытаемся сопоставить *Pattern* с *Text*, двигаясь назад по *Pattern*. Например, если мы хотим сопоставить *Pattern* = «**ana**» с *Text* = «panamabananas$», то сначала идентифицируем строки *M*(*Text*), начинающиеся с «a», последней буквы «**ana**»:

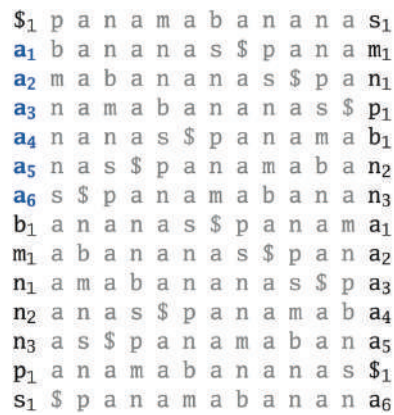


Рис. 9.28 Идентификация строк *M*(*Text*), начинающихся с «a»

По мере того как мы движемся назад через «**ana**», мы будем искать строки *M*(*Text*), начинающиеся с «**na**». Чтобы сделать это, не зная всей матрицы

M(*Text*), мы снова используем тот факт, что символ в *LastColumn* должен предшествовать символу *Text*, находящемуся в той же строке в *FirstColumn*. Таким образом, нужно идентифицировать только те строки *M*(*Text*), которые начинаются с «a» и заканчиваются «**an**»:

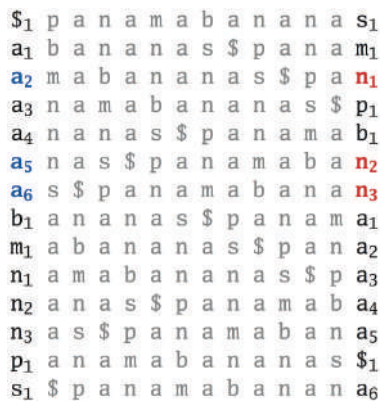


Рис. 9.29 Поиск строк *M*(*Text*), начинающихся с «**na**»

Свойство «первый–последний» сообщает нам, где найти три выделенных «**n**» в *FirstColumn*, как показано ниже. Все три строки заканчиваются на «a», что дает три вхождения «**ana**» в *Text*.

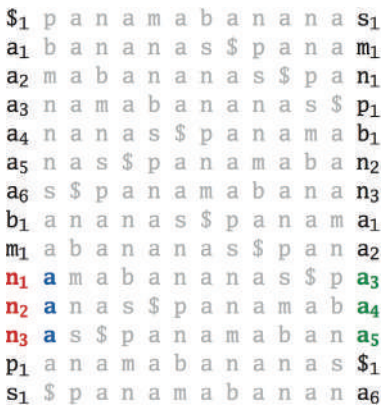


Рис. 9.30 Три вхождения «**ana**» в *Text*

Выделенные вхождения «**a**» в *LastColumn* соответствуют третьему, четвертому и пятому вхождениям «**a**» в этом столбце, а свойство «первый–последний» говорит нам, что они должны соответствовать третьему, четвертому и пятому вхождениям «a» в *FirstColumn*, который идентифицирует три совпадения «**ana**»:

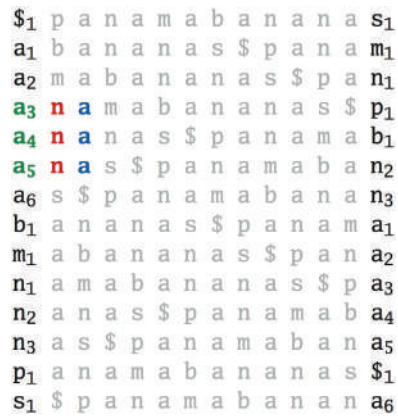


Рис. 9.31 Три совпадения «ana»

Упражнение. Сопоставьте *Pattern* = «banana» с *Text* = «panamabananas$», пройдя назад по *Pattern* с помощью преобразования Барроуза–Уилера для *Text*.

Маппинг «последний–первый»

Теперь мы знаем, как использовать *BWT*(*Text*) для поиска всех совпадений *Pattern* с *Text* путем прохода *Pattern* в обратном направлении. Однако каждый раз, когда мы идем назад, нам нужно отслеживать, где спрятаны совпадения суффикса *Pattern* в начале *M*(*Text*). К счастью, мы знаем, что на каждом шаге строки *M*(*Text*), соответствующие суффиксу *Pattern*, объединяются в последовательные строки *M*(*Text*). Это означает, что набор всех совпадающих строк раскрывается только двумя указателями, верхним и нижним: верхний содержит индекс первой строки *M*(*Text*), которая соответствует текущему суффиксу Pattern, а нижний содержит индекс последней строки из *M*(*Text*), который соответствует этому суффиксу. На рисунке ниже показан процесс обновления указателей; пройдя назад через *Pattern* = «ana», у нас есть *top* = 3 и *bottom* = 5. После прохода *Pattern* мы можем вычислить общее количество совпадений *Pattern* в *Text*, вычислив *bottom* – *top* + 1 (например, существует 5 – 3 + 1 = 3 совпадения «ana» в «panamabananas$»).

Давайте сосредоточимся на том, как указатели обновляются от одного этапа к другому. Рассмотрим переход от второй таблицы к третьей на рисунке ниже; как мы узнали, что нужно обновить указатели (*top* = 1, *bottom* = 6) на (*top* = 9, *bottom* = 11)? Мы ищем первое и последнее вхождение «n» в диапазоне позиций от *top* = 1 до *bottom* = 6 в *LastColumn*. Первое появление «n» в этом диапазоне – «n_1» (в позиции 2), а последнее – «n_3» (позиция 6).

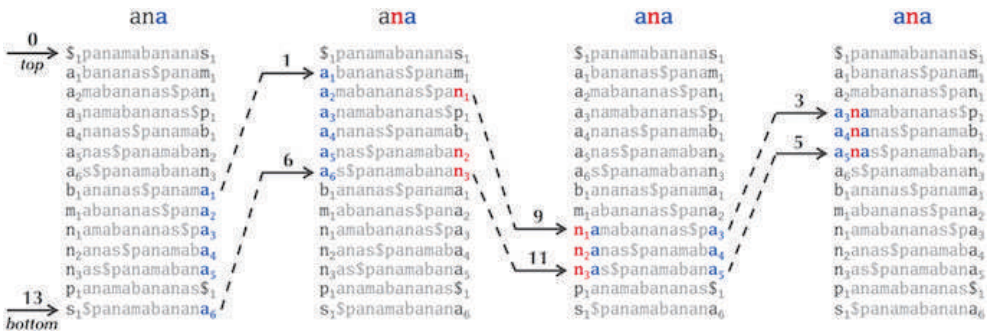


Рис. 9.32 Указатели *top* и *bottom* содержат индексы первой и последней строк *M*(*Text*), соответствующие текущему суффиксу *Pattern* = «ana». В диаграмме показано, как эти указатели обновляются при переходе назад по «ana» и поиске совпадений подстрок в «panamabananas$»

Чтобы обновить верхний и нижний указатели, нам нужно определить, где в *FirstColumn* встречаются «$\mathbf{n_1}$» и «$\mathbf{n_3}$». **Массив Last-to-First**, обозначаемый *LastToFirst*(*i*), отвечает на следующий вопрос: если задан символ в позиции *i* в *LastColumn*, какова его позиция в *FirstColumn*? Для нашего текущего примера *LastToFirst*(2) = 9, поскольку символ в позиции 2 *LastColumn*(«$\mathbf{n_1}$») соответствует позиции 9 в *FirstColumn*, как показано в рис. 9.33. Точно так же *LastToFirst*(6) = 11, так как символ в позиции 6 *LastColumn*(«$\mathbf{n_3}$») встречается в позиции 11 в *FirstColumn*. Следовательно, с помощью маппинга «последний–первый» мы можем быстро обновить указатели (*top* = 1, *bottom* = 6) на (*top* = 9, *bottom* = 11).

i	*FirstColumn*	*LastColumn*	LASTTOFIRST(*i*)
0	$\$_1$	s_1	13
1	a_1	m_1	8
2	a_2	n_1	9
3	a_3	p_1	12
4	a_4	b_1	7
5	a_5	n_2	10
6	a_6	n_3	11
7	b_1	a_1	1
8	m_1	a_2	2
9	n_1	a_3	3
10	n_2	a_4	4
11	n_3	a_5	5
12	p_1	$\$_1$	0
13	s_1	a_6	6

Рис. 9.33 Для данного символа в позиции *i LastColumn* выравнивание Last-to-First идентифицирует позицию этого символа в *FirstColumn*

Теперь мы готовы описать **BWMatching**, алгоритм, который подсчитывает общее количество совпадений с *Pattern* в тексте, при том, что единственная информация, которую мы имеем, – это *FirstColumn* и *LastColumn* в дополнение к отображению Last-to-First. Указатели *top* и *bottom* обновляются в следующем псевдокоде.

```
BWMatching(LastColumn, Pattern, LastToFirst)
    top ← 0
    bottom ← |LastColumn| − 1
    while top ≤ bottom
        if Pattern не пустой
            symbol ← последняя буква в Pattern
            удалить последнюю букву в Pattern
            if позиции сверху вниз в LastColumn содержат вхождение symbol
                topIndex ← первая позиция symbol среди позиций от top к bottom
                в LastColumn
                bottomIndex ← последняя позиция symbol среди позиций от top
                к bottom в LastColumn
                top ← LastToFirst(topIndex)
                bottom ← LastToFirst(bottomIndex)
            else
                return 0
        else
            return bottom − top + 1
```

Делаем сопоставление паттернов по Барроузу–Уилеру быстрее

Замена маппинга «последний–первый» оценочными массивами

Если вы реализовали **BWMatching** в предыдущем разделе, вы, вероятно, обнаружили, что алгоритм работает медленно. Причина его медлительности заключается в том, что обновление указателей *top* и *bottom* занимает много времени, поскольку требует проверки каждого символа в *LastColumn* между *top* и *bottom* на каждом шаге. Чтобы улучшить **BWMatching**, введем функцию $Count_{symbol}(i, LastColumn)$, которая выдает количество вхождений *symbol* в первых *i* позициях *LastColumn*. Например, $Count_{«n»}(10, «smnpbnnaaaaa\$a») = 3$, а $Count_{«a»}(4, «smnpbnnnaaaaa\$a») = 0$. Ниже показаны массивы, содержащие $Count_{symbol}(i, «smnpbnnnaaaaa\$a»)$ для каждого *symbol*, встречающегося в «panamabananas$».

i	*FirstColumn*	*LastColumn*	LastToFirst(i)	Count						
				$	a	b	m	n	p	s
0	$$_1$$	$$s_1$$	13	0	0	0	0	0	0	0
1	$$a_1$$	$$m_1$$	8	0	0	0	0	0	0	1
2	$$a_2$$	$$n_1$$	9	0	0	0	1	0	0	1
3	$$a_3$$	$$p_1$$	12	0	0	0	1	1	0	1
4	$$a_4$$	$$b_1$$	7	0	0	0	1	1	1	1
5	$$a_5$$	$$n_2$$	10	0	0	1	1	1	1	1
6	$$a_6$$	$$n_3$$	11	0	0	1	1	2	1	1
7	$$b_1$$	$$a_1$$	1	0	0	1	1	3	1	1
8	$$m_1$$	$$a_2$$	2	0	1	1	1	3	1	1
9	$$n_1$$	$$a_3$$	3	0	2	1	1	3	1	1
10	$$n_2$$	$$a_4$$	4	0	3	1	1	3	1	1
11	$$n_3$$	$$a_5$$	5	0	4	1	1	3	1	1
12	$$p_1$$	$$\$_1$$	0	0	5	1	1	3	1	1
13	$$s_1$$	$$a_6$$	6	1	5	1	1	3	1	1
				1	6	1	1	3	1	1

Рис. 9.34 Массивы, содержащие $Count_{symbol}(i,$ «smnpbnnnaaaaa$a»)

Упражнение. Вычислите массивы *Count* для *BWT*(«abracadabra$»).

Теперь мы покажем, как обновлять указатели, не проверяя каждый символ в *LastColumn* между *top* и *bottom* на каждом шаге. Говорят, что *k*-е вхождение *symbol* в столбец матрицы имеет **ранг** *k* в этом столбце. Для *Text* = «panamabananas$» снова взглянув на таблицу, приведенную выше, обратите внимание, что первое и последнее вхождения *symbol* в диапазоне позиций от *top* к *bottom* в *LastColumn* имеют соответствующие ранги.

$$Count_{symbol}(top, LastColumn) + 1,$$

а также

$$Count_{symbol}(bottom + 1, LastColumn).$$

Например, когда *top* = 1, *bottom* = 6 и *symbol* = «n», мы имеем $Count_{«n»}(top - 1, LastColumn) + 1 = 1$ и $Count_{«n»}(bottom + 1, LastColumn) = 3$. Вхождения «n», имеющие ранги 1 и 3, расположены на позициях 2 и 6 *LastColumn*, подразумевая, что мы должны обновить *top* до *LastToFirst*(2) = 9 и *bottom* до *LastToFirst*(6) = 11.

Теперь давайте вернемся к псевдокоду для **BWMatching**.

```
BWMatching(LastColumn, Pattern, LastToFirst)
  top ← 0
  bottom ← |LastColumn| − 1
  while top ≤ bottom
```

```
    if Pattern не пустой
        symbol ← последняя буква в Pattern
        удалить последнюю букву из Pattern
        if позиции top к bottom в LastColumn содержат вхождение symbol
            topIndex ← первая позиция symbol среди позиций от top до bottom
            в LastColumn
            bottomIndex ← последняя позиция symbol среди позиций от top до
            bottom в LastColumn
            top ← LastToFirst(topIndex)
            bottom ← LastToFirst(bottomIndex)
        else
            return 0
    else
        return bottom − top + 1
```

Четыре зеленые строки можно переписать следующим образом:

```
topIndex ← позиция symbol ранга Count_symbol(top, LastColumn) + 1 в LastColumn
bottomIndex ← позиция symbol ранга Count_symbol(bottom + 1, LastColumn)
в LastColumn
top ← LastToFirst(topIndex)
bottom ← LastToFirst(bottomIndex)
```

Исключив переменные *topIndex* и *bottomIndex*, мы можем сократить эти четыре строки до двух строк:

```
top ← LastToFirst(позиция symbol ранга Count_symbol(top, LastColumn) + 1
в LastColumn)
bottom ← LastToFirst(позиция symbol ранга Count_symbol(bottom + 1, LastColumn)
в LastColumn)
```

Обратите внимание, что эти две строки псевдокода просто вычисляют позицию *symbol* с рангом *i* в *FirstColumn* на основе его позиций в *LastColumn*. Эту задачу можно компактно описать без отображения «первый–последний» следующими двумя строками:

```
top ← позиция symbol ранга Count_symbol(top, LastColumn) + 1 в FirstColumn
bottom ← позиция symbol ранга Count_symbol(bottom + 1, LastColumn) в FirstColumn
```

Для *top* = 1, н *bottom* = 6 и *symbol* = «n», символы «n» с рангами $Count_{«n»}(top, LastColumn) + 1 = 1$ и $Count_{«a»}(bottom + 1, LastColumn) = 3$ расположены в позициях 9 и 11 *FirstColumn*. Поэтому мы обновляем *top* = 9 и *bottom* = 11, как и раньше.

ОСТАНОВИТЕСЬ и задумайтесь. Нужно ли нам хранить всю *FirstColumn* в памяти, чтобы выполнить предыдущие две строки псевдокода?

Удаление первого столбца матрицы Барроуза–Уилера

BWMatching требует, чтобы мы использовали *FirstColumn*. Мы можем уменьшить объем памяти, необходимый для хранения информации, содержащейся в *FirstColumn*, определив *FirstOccurrence*(*symbol*) как первую позицию *symbol* в *FirstColumn*. Если *Text* = «panamabananas$», то *FirstColumn* равен «$aaaaaabmnnnps», а массив, содержащий все значения *FirstOccurrence*, равен [0, 1, 7, 8, 9, 12, 13], как показано ниже. Массив *FirstOccurrence* можно вычислить непосредственно из *LastColumn*, что устраняет необходимость в *FirstColumn*, а для строк ДНК любой длины этот массив содержит всего пять элементов.

Две строки псевдокода, о которых шла речь выше, теперь можно переписать следующим образом:

$top \leftarrow FirstOccurrence(symbol) + Count_{symbol}(top, LastColumn)$
$top \leftarrow FirstOccurrence(symbol) + Count_{symbol}(bottom + 1, LastColumn) - 1$

Когда $top = 1$, $bottom = 6$ и $symbol$ = «n», мы имеем $FirstOccurrence$(«n») = 9, $Count_{«n»}(top, LastColumn) = 0$ и $Count_{«n»}(bottom + 1, LastColumn) = 3$, что снова означает, что ($top = 1$, $top = 6$) будет обновлено до ($top = 9 + 0 = 9$, $top = 9 + 3 - 1 = 11$).

i	*FirstColumn*	FIRSTOCCURRENCE
0	$\$_1$	0
1	a_1	1
2	a_2	
3	a_3	
4	a_4	
5	a_5	
6	a_6	
7	b_1	7
8	m_1	8
9	n_1	9
10	n_2	
11	n_3	
12	p_1	12
13	s_1	13

Рис. 9.35 *FirstColumn и FirstOccurrence*

В процессе упрощения зеленых строк псевдокода из **BWMatching** мы также устранили необходимость в *FirstColumn* и *LastToFirst*, что привело к созданию более эффективного алгоритма под названием **BetterBWMatching**.

```
BetterBWMatching(FirstOccurrence, LastColumn, Pattern, Count)
    top ← 0
    bottom ← |LastColumn| − 1
    while top ≤ bottom
        if Pattern не пустой
            symbol ← последняя буква в Pattern
            удалить последнюю букву в Pattern
            if позиции от top до bottom в LastColumn содержат вхождения symbol
                top ← FirstOccurrence(symbol) + Count_symbol(top, LastColumn)
                bottom ← FirstOccurrence(symbol) + Count_symbol(bottom + 1, LastColumn) − 1
            else
                return 0
        else
            return bottom − top + 1
```

Вам может быть интересно, почему мы назвали этот алгоритм «более эффективным», потому что, если нужно вычислять массивы *Count* по мере продвижения, вы не получите ускорения во время выполнения. Если необходимо предварительно вычислить эти массивы, вам нужно будет хранить их в памяти, которая занимает много места. Имейте это в виду.

Где находятся совпадающие паттерны?

Мы надеемся, что вы заметили ограничение **BetterBWMatching**, – этот алгоритм подсчитывает количество вхождений *Pattern* в *Text*, но где эти вхождения расположены в *Text*, он нам не сообщает! Чтобы найти совпадения с паттерном, идентифицированные **BetterBWMatching**, мы снова можем использовать суффиксный массив, как показано на рис. 9.36. Например, суффиксный массив сразу же находит три совпадения «ana» в «panamabananas$».

Суффиксный массив упрощает нашу работу, но помните, что изначально мы использовали преобразование Барроуза–Уилера, чтобы *уменьшить* объем памяти, используемый суффиксным массивом для выравнивания с паттерном. Если мы добавим суффиксный массив к выравниванию с паттерном на основе Барроуза–Уилера, то вернемся к тому, с чего начали!

Устройство для экономии памяти, которое мы будем использовать, некрасиво, но полезно. Мы создадим **неполный суффиксный массив** *Text*, обозначив его $SuffixArray_K(Text)$, который содержит только значения, кратные некоторому положительному целому числу K (рис. 9.37). В реальных приложениях частичные суффиксные массивы часто строятся для $K = 100$, что снижает использование памяти в 100 раз по сравнению с полным суффиксным массивом.

Дополнительные сведения см. в разделе **ЗАРЯДНАЯ СТАНЦИЯ: Построение неполного суффиксного массива**.

M(*Text*)	SUFFIXARRAY(*Text*)
$ p a n a m a b a n a n a s	13
a b a n a n a s $ p a n a m	5
a m a b a n a n a s $ p a n	3
a n a m a b a n a n a s $ p	1
a n a n a s $ p a n a m a b	7
a n a s $ p a n a m a b a n	9
a s $ p a n a m a b a n a n	11
b a n a n a s $ p a n a m a	6
m a b a n a n a s $ p a n a	4
n a m a b a n a n a s $ p a	2
n a n a s $ p a n a m a b a	8
n a s $ p a n a m a b a n a	10
p a n a m a b a n a n a s $	0
s $ p a n a m a b a n a n a	12

Рис. 9.36 Суффиксы «panamabananas$», которые начинают строки *M*(«panamabananas$»), выделены, а суффиксы, начинающиеся с «ana», показаны зеленым цветом. Суффиксный массив содержит начальную позицию каждого суффикса в тексте

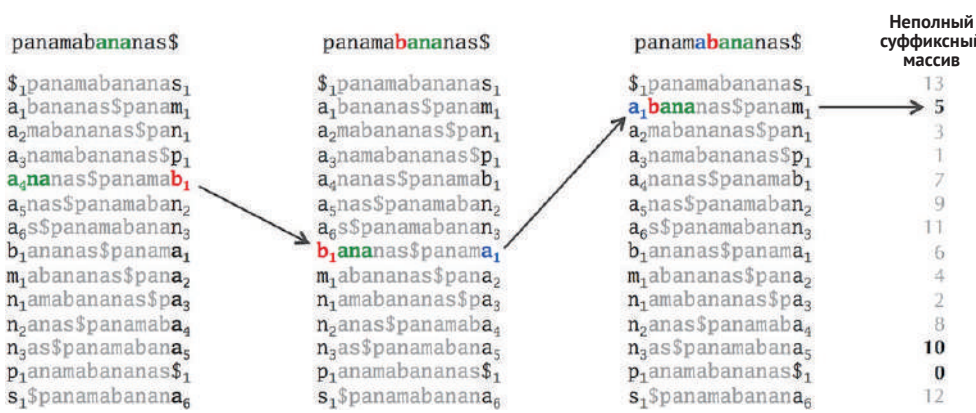


Рис. 9.37 Одно из совпадений «ana» в «panamabananas$» выделено зеленым (справа). Это совпадение происходит в начальной позиции 7, но мы не знаем этого сразу, потому что у нас нет такой роскоши, как сохранение всего суффиксного массива. Однако, пройдя назад, мы обнаружим, что «ana» предшествует «b_1», которому, в свою очередь, предшествует «a_1». Частичный суффиксный массив выше, сгенерированный для $K = 5$, указывает, что «a_1» встречается в позиции 5 «panamabananas». Поскольку нам потребовалось два шага, чтобы пройти назад к «a_1», мы заключаем, что это появление «ana» начинается в позиции 5 + 2 = 7

ОСТАНОВИТЕСЬ и задумайтесь. Неполный суффиксный массив уменьшает объем памяти в K раз, но также замедляет поиск совпадения с паттерном из-за «возвратов», что показано на рисунке ниже. Каково число шагов в наихудшем случае, которое необходимо сделать при ходьбе задом наперед? Как это повлияет на время выполнения (O большое) нашего алгоритма Pattern Matching?

Барроуз и Уиллер устанавливают контрольные точки

Теперь мы обсудим, как улучшить алгоритм **BetterBWMatching** (повторенный еще раз ниже) путем решения компромисса между предварительным вычислением значений $Count_{symbol}(i, LastColumn)$ (требуется значительный объем памяти) и вычислением этих значений по ходу работы (требуется значительное время выполнения).

```
BetterBWMatching(FirstOccurrence, LastColumn, Pattern, Count)
    top ← 0
    bottom ← |LastColumn| − 1
    while top ≤ bottom
        if Pattern не пустой
            symbol ← последняя буква в Pattern
            удалить последнюю букву в Pattern
            if позиции от top до bottom в LastColumn содержат вхождения symbol
                top ← FirstOccurrence(symbol) + Count_symbol(top, LastColumn)
                bottom ← FirstOccurrence(symbol) + Count_symbol(bottom + 1, LastColumn) − 1
            else
                return 0
        else
            return bottom − top + 1
```

Баланс, которого мы добиваемся, аналогичен тому, который используется для частичного суффиксного массива. Вместо того чтобы хранить $Count_{symbol}(i, LastColumn)$ для всех позиций i, мы будем хранить массивы $Count$ только тогда, когда i делится на C, где C – константа; эти массивы называются **массивами контрольных точек**. Когда C велико (на практике C обычно равно 100) и алфавит мал (например, 4 нуклеотида), для массивов контрольных точек требуется лишь часть памяти, используемой $BWT(Text)$.

i	*LastColumn*	Count						
		$	a	b	m	n	p	s
0	s_1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	m_1	0	0	0	0	0	0	1
2	n_1	0	0	0	1	0	0	1
3	p_1	0	0	0	1	1	0	1
4	b_1	0	0	0	1	1	1	1
5	n_2	**0**	**0**	**1**	**1**	**1**	**1**	**1**
6	n_3	0	0	1	1	2	1	1
7	a_1	0	0	1	1	3	1	1
8	a_2	0	1	1	1	3	1	1
9	a_3	0	2	1	1	3	1	1
10	a_4	**0**	**3**	**1**	**1**	**3**	**1**	**1**
11	a_5	0	4	1	1	3	1	1
12	$\$_1$	0	5	1	1	3	1	1
13	a_6	1	5	1	1	3	1	1
		1	6	1	1	3	1	1

Рис. 9.38 Массивы контрольных точек *Count* для *Text* = «panamabananas$» и *C* = 5 выделены жирным шрифтом. Если мы хотим вычислить $Count_{»a»}$(13, «smnpbnnnaaaaa$a»), то массив контрольных точек в позиции 10 говорит нам, что есть 3 вхождения «a» до позиции 10 «smnpbnnn**aaa**aa$a». Затем проверяем, присутствует ли «a» в позиции **10** (да), **11** (да) и **12** (нет) *LastColumn*, чтобы сделать вывод, что $Count_{»a»}$(13, «smnpbnnnaaaaa$a») = **3** + **2** = 5

Что насчет времени выполнения? Используя массивы контрольных точек, мы можем вычислить верхний и нижний указатели за постоянное число шагов (т. е. меньше, чем *C*). Поскольку для каждого *Pattern* требуется не более |*Pattern*| обновлений указателя, модифицированный алгоритм **BetterBWMatching** теперь требует времени выполнения $O(|Patterns|)$, что аналогично использованию префиксных или суффиксных массивов.

Кроме того, теперь нам нужно хранить в памяти только следующие данные: *BWT*(*Text*), *FirstOccurrence*, частичный суффиксный массив и массивы контрольных точек. Для хранения этих данных требуется память, приблизительно равная $1{,}5 \cdot |Text|$. Таким образом, мы, наконец, сократили объем памяти, необходимый для решения задачи множественного выравнивания для миллионов ридов секвенирования, до разумного диапазона.

Поскольку резко падающая стоимость секвенирования ДНК стала общеизвестной, легко не оценить прогресс, достигнутый в вычислительной части маппинга ридов. Итак, прежде чем перейти к приблизительному сопоставлению паттернов, мы хотели бы сделать паузу и подумать о том, насколько далеко продвинулись алгоритмы маппинга ридов и аппаратное обеспечение, на котором их применяют. В 1975 году передовой алгоритм Ахо–Корасик рекламировался

как требующий 15 мин для выравнивания всего 24 английских слов со словарем. Всего через поколение, когда методы, основанные на методах Барроуза–Уилера, были введены в картографирование ридов, те же 15 мин были использованы для картирования почти десяти миллионов ридов эталонного генома. Увидев, как далеко мы продвинулись за последние четыре десятилетия, можно только представить, где мы окажемся еще через четыре десятилетия.

Эпилог. Устойчивое к несовпадениям картирование рида

Сведение приблизительного сопоставления с паттерном к точному

В этом разделе мы вернемся к цели идентификации SNP в индивидуальном геноме по сравнению с эталонным геномом. Для этого нам нужно обобщить задачу приблизительного сравнения паттернов из предыдущей главы на случай нескольких паттернов.

Задача на множественное приблизительное сопоставление паттернов: *найти все приблизительные совпадения Patterns с текстом.*

Input: строка *Text*, набор *Patterns* (более коротких) строк и целое число *d*.

Output: все начальные позиции в тексте, где строка из *Patterns* появляется как подстрока не более чем с *d* несовпадениями.

Начнем с простого наблюдения: если *Pattern* встречается в *Text* с одним несовпадением, то мы можем разделить *Pattern* на две половины, одна из которых встречается точно в *Text*, как показано ниже.

Pattern acttggct
Text ...ggcactaggctcc...

Таким образом, мы можем определить, встречается ли *Pattern* с одним несовпадением в *Text*, разделив *Pattern* на две половины и затем найдя точные совпадения этих более коротких строк. Если мы находим совпадение с одной из этих половин *Pattern*, то затем проверяем, встречается ли вся строка *Pattern* с единственной мутацией.

Этот метод можно легко обобщить для приблизительного сопоставления с паттерном с $d > 1$ несовпадениями; если *Pattern* приблизительно соответствует подстроке *Text* не более чем с *d* несовпадениями, то *Pattern* и *Text* должны

иметь общий хотя бы один k-мер для достаточно большого значения k. Например, если мы ищем *Pattern* длины 20 не более чем с $d = 3$ несовпадениями, то мы можем разделить этот паттерн на четыре части длины 20/(3+1)=5 и искать точные совпадения этих более коротких подстроках:

Pattern acttaggctcgggataatcc
Text ...actaagtctcgggataagcc...

Это наблюдение полезно, поскольку оно сводит *приблизительное* сопоставление с паттерном к точному сопоставлению с более короткими последовательностями, что позволяет нам использовать быстрые алгоритмы, предназначенные для точного сопоставления с паттерном, но не применимые к приблизительному, такие как методы, основанные на суффиксных деревьях и суффиксных массивах.

ОСТАНОВИТЕСЬ и задумайтесь. Если *Pattern* имеет длину 23 и появляется в *Text* с тремя несовпадениями, можем ли мы заключить, что *Pattern* имеет тот же 6-мер, что и *Text*? Можем ли мы заключить, что он имеет общий 5-мер с *Text*?

Остается вопрос, как найти максимальное значение k, которое гарантирует, что любой *Pattern* длины n с d несовпадениями с *Text* будет иметь k-мерную подстроку, точно совпадающую с *Text*.

Теорема. Если две строки длины n совпадают не более чем с d несовпадениями, то они должны иметь общий k-мер длины $k = \lfloor n/(d + 1) \rfloor$.

Доказательство. Разделите первую строку на $d + 1$ подстроку, где первые d подстрок содержат ровно k символов, а последняя подстрока содержит не менее k символов. Для случая $d = 3$ здесь представлено разбиение строки из 23 нуклеотидов ($n = 23$) на $d + 1 = 4$ подстроки, где первые три подстроки имеют $k = \lfloor n/(d + 1) \rfloor = 23/4 = 5$ символов, а последняя подстрока имеет восемь символов.

acttaggctcgggataatccgga

Если вы распределите d несовпадений между позициями строки, то несовпадения могут повлиять не более чем на d подстрок, что оставит по крайней мере одну подстроку (длиной не менее k) неизменной. Эта подстрока является общей для обеих строк, что и требовалось доказать. ■

Теперь у нас есть схема алгоритма выравнивания строки *Pattern* длины n с *Text* не более чем с d несовпадениями. Сначала мы разделим *Pattern* на $d + 1$ сегментов длины $k = \lfloor n/(d + 1) \rfloor$, называемых начальными числами, или **семенами (seed)**. После определения того, какие начальные числа точно соответствуют *Text* (обнаружение начального числа, **seed detection**), мы пытаемся расширить начальные значения в обоих направлениях, чтобы проверить, встречается ли в *Text* паттерн не более чем с d несовпадениями.

BLAST: Сравнение последовательности с базой данных

Использование общих k-меров для поиска сходства между биологическими последовательностями имеет некоторые недостатки. Например, два белка могут иметь схожие функции, но не иметь общих k-меров, даже при малых значениях k.

Базовый инструмент поиска местного выравнивания (Basic Local Alignment Search Tool, или **BLAST)** – это эвристика, которая может найти сходство между белками, даже если все аминокислоты в одном белке мутировали по сравнению с другим белком. BLAST настолько быстр, что его часто используют для сопоставления белка со всеми другими известными белками в базах данных белков. Статья, посвященная BLAST, была опубликована в 1990 году и, получив более 40 000 цитирований, стала одной из самых цитируемых научных статей из когда-либо опубликованных.

Чтобы увидеть, как работает BLAST, давайте возьмем целое число k и строки $x = x_1 \ldots x_n$ и $y = y_1 \ldots y_m$ для сравнения (представляющих, скажем, два белка). Мы определяем **сегментную пару** x и y как пару, образованную k-мером из x и k-мером из y. Оценка сегментной пары, соответствующей k-мерам, начинающимся с позиции i в x и позиции j в y, равна

$$\sum_{t=0}^{k-1} Score(x_{i+t}, y_{j+t}),$$

где $Score(x_{i+t}, y_{j+t})$ определяется матрицей счета, такой как оценочная матрица PAM, которую мы ввели, когда говорили о выравнивании последовательностей. **Локально максимальная сегментная пара** – это сегментная пара, чей счет нельзя увеличить, удлиняя или укорачивая строки в паре. BLAST пытается найти не сегментную пару x и y с наивысшим счетом, а все локально максимальные сегментные пары в этих строках со счетом выше некоторого порога.

Для каждого k-мера в строке запроса BLAST быстро находит набор всех k-меров, которые имеют счет выше заданного порога по сравнению с этим k-мером, и объединяет эти наборы (для всех k-меров в строке запроса), чтобы получить список потенциальных начальных чисел (семян). Если порог счета высок, то набор всех результирующих сегментных пар, образованных строкой запроса и строками в базе данных, не слишком велик. В этом случае в базе данных можно искать точные вхождения этих k-меров с высоким счетом из этого набора, создавая начальный набор семян. Это пример задачи множественного выравнивания паттернов, которую мы научились быстро решать.

Упражнение. В матрице счета PAM_{250} только пять 3-меров набирают выше 23 по сравнению с CFC: CIC, CLC, CMC, CWC и CYC. Сколько 3-меров набрали больше 20 по сравнению с CFC?

☞ **Загрузить данные 9.1 (матрица счета PAM_{250})**

После обнаружения семян BLAST пытается расширить их список (с учетом вставок и удалений), чтобы получить локально максимальные сегментные пары.

Упражнение. Для данной матрицы счета PAM_{250}, аминокислотного k-мера *Peptide* и порогового значения θ разработайте эффективный алгоритм нахождения точного числа k-меров, имеющих счет более чем θ по сравнению с *Peptide*.

Приблизительное сопоставление последовательностей с помощью преобразования Барроуза–Уилера

Чтобы расширить метод Барроуза–Уилера к приблизительному сопоставлению, мы не будем останавливаться, когда столкнемся с несовпадением. Наоборот, будем двигаться дальше, пока либо не найдем приблизительное совпадение, либо не превысим предел d несовпадений.

На рис. 9.39 показан поиск «asa» в «panamabananas$» не более чем с одним несовпадением. Давайте сначала продолжим, как и в случае точного совпадения с паттерном, проходя по «asa» в обратном направлении с помощью преобразования Барроуза–Уилера. Найдя шесть вхождений «a», мы идентифицируем шесть неточных вхождений «sa»: «**p**a», «**m**a», «**b**a» и три вхождения «**n**a». Мы отмечаем, что эти три строки накопили несовпадение, а затем продолжаем со всеми шестью строками.

На следующем шаге пять неточных вхождений «sa» могут быть расширены до неточных вхождений «asa» только с одним несовпадением: «a**m**a», «a**b**a» и три вхождения «a**n**a». Мы не можем расширить «**p**a», которое исключаем из рассмотрения. Как правило, нам не удается расширить растущий паттерн либо из-за того, что мы достигли конца *Text* (как показано попадание в «$» в случае «**p**a»), либо из-за превышения максимально допустимого количества несовпадений.

На практике эта эвристика сталкивается с осложнениями. Нам не стоит начинать допускать несовпадающие строки на ранних этапах **BetterBWMatching**, иначе придется рассматривать слишком много необоснованных строк-кандидатов. Поэтому мы можем потребовать, чтобы суффикс *Pattern* некоторой пороговой длины точно соответствовал *Text*. Кроме того, метод становится трудоемким при использовании больших значений d, так как мы должны исследовать множество неточных совпадений. Практические приложения часто ограничивают значение d не более чем 3.

Теперь вы должны быть готовы разработать свой собственный подход к решению проблемы множественного приблизительного выравнивания паттернов и использовать это решение для солпоставления реальных ридов секвенирования.

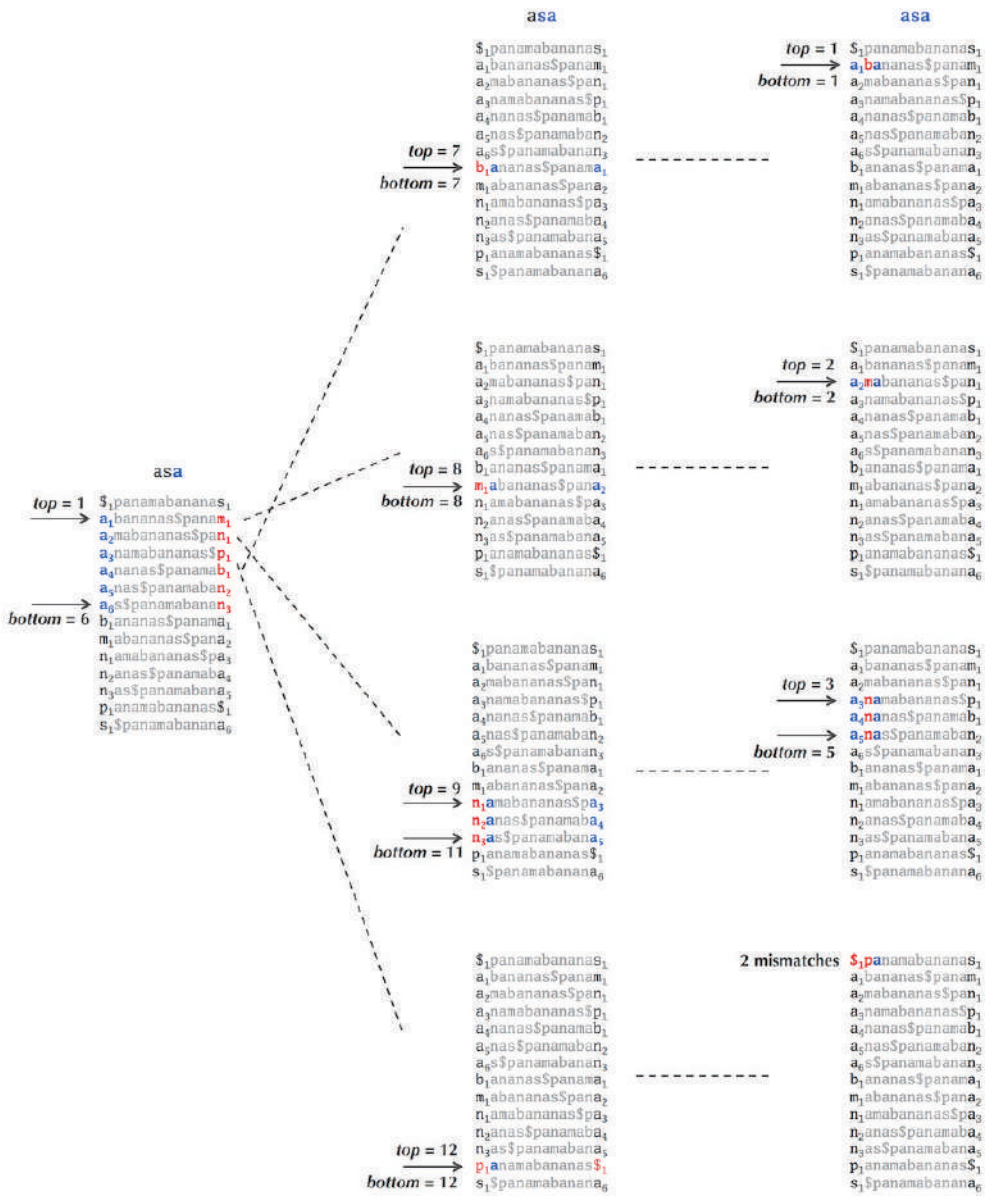


Рис. 9.39 Использование преобразования Барроуза–Уилера для поиска приблизительных совпадений паттерна «asa» в «panamabananas$» с одним несовпадением. (Слева) Мы идентифицируем шесть вхождений «a». (Посередине) Работая в обратном порядке, мы находим четыре различных неточных частичных совпадения: «ba», «ma», «na» (три вхождения) и «ra». (Справа) Дополнительные несовпадения в частичных совпадениях, которые мы нашли, не допускаются, но эти частичные совпадения можно расширить, чтобы получить пять неточных совпадений «asa»

Заключительная задача. Имея преобразование Барроуза–Уилера и неполный суффиксный массив бактериального генома *Mycoplasma pneumoniae* вместе с набором ридов, найдите все риды, встречающиеся в геноме, не более чем с одним несовпадением.

☞ **Загрузить данные 9.2**

Сопутствующие материалы

Построение суффиксного дерева

Чтобы построить суффиксное дерево, сначала мы изменим конструкцию суффиксного дерева следующим образом. Хотя каждое ребро в дереве суффиксов помечено одним символом *Symbol*(*edge*), неясно, откуда этот символ появился в *Text*. Поэтому мы добавим еще одну метку для каждого ребра (обозначенную как *Position*(*edge*)), относящуюся к положению этого символа в *Text*. Если ребро в дереве соответствует более чем одной позиции в *Text*, мы назначим ему минимальную начальную позицию.

Например, рассмотрим модифицированную суффиксную попытку (suffix trie) для *Text* = «panamabananas», показанную на рис. 9.40 на следующем шаге. Есть пять ребер, помеченных буквой «m» (окрашены в фиолетовый цвет), все они помечены позицией **4**, поскольку это единственное появление «m» в *Text*. С другой стороны, зеленое ребро, соответствующее «**n**», – это совсем другая история. Следуя каждому пути от этого ребра вниз к листьям, мы видим, что оно соответствует вхождениям «**n**» в суффиксы «a**n**amabananas$», «a**n**anas$» и «a**n**as$» на позициях **2**, **8** и **10**. В результате присваиваем этому ребру минимум из этих позиций.

Следующий псевдокод строит измененную суффиксную попытку строки *Text* путем перебора суффиксов *Text* от самого длинного до самого короткого. Получив суффикс, он пытается определить суффикс, перемещаясь вниз по дереву, следуя граничным меткам, насколько это возможно, пока уже не может идти дальше. В этот момент он добавляет оставшуюся часть суффикса к дереву trie в виде пути к листу вместе с позицией каждого символа в суффиксе.

```
ModifiedSuffixTrieConstruction(Text)
    Trie ← граф, состоящий из одного узла root
    for i ← 0 до |Text| – 1
        currentNode ← root
        for j ← i до |Text| – 1
            currentSymbol ← j-й символ Text
            if есть исходящее из currentNode ребро, обозначенное currentSymbol
                currentNode ← конечный узел этого ребра
```

```
        else
            добавить новый узел newNode к Trie
            добавить ребро newEdge, соединяющее currentNode с newNode в Trie
            Symbol(newEdge) ← currentSymbol
            Position(newEdge) ← j
            currentNode ← newNode
    if currentNode является листом дерева Trie
        присвоить метку i этому листу
return Trie
```

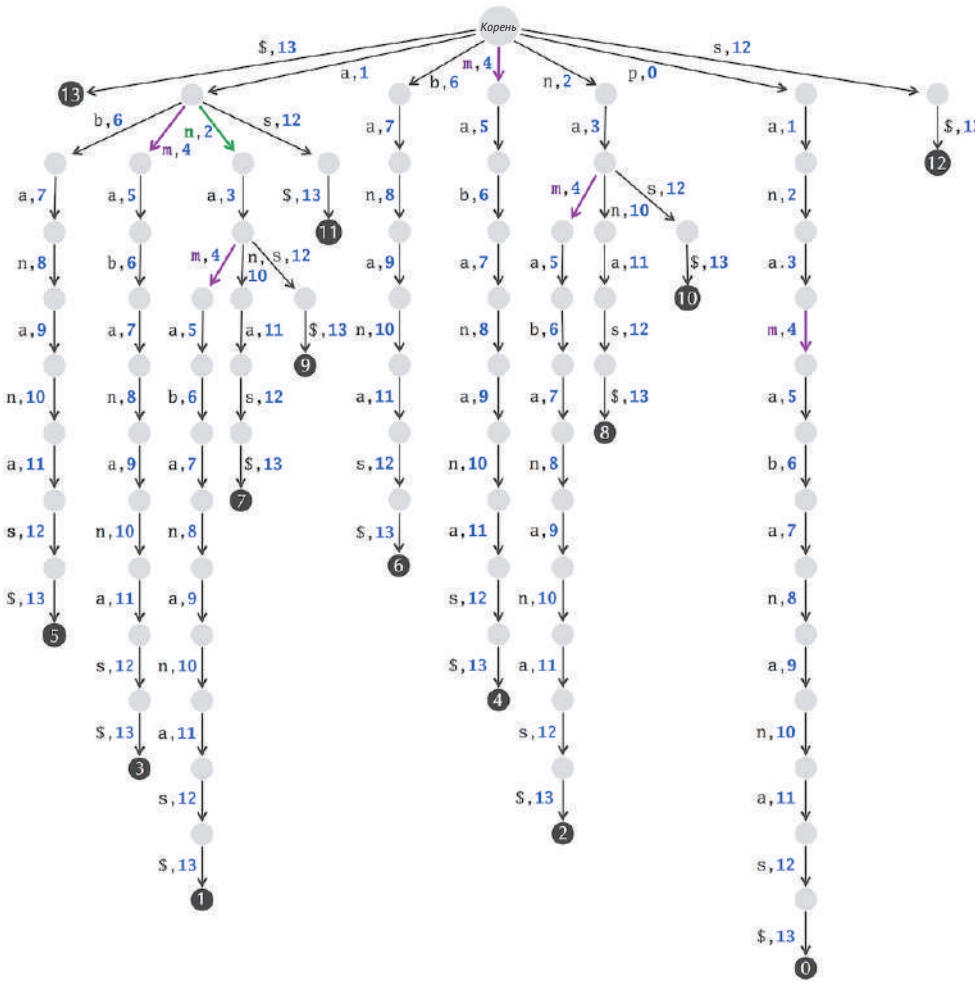


Рис. 9.40 Модифицированная суффиксная попытка для *Text* = «panama-bananas$». Каждому ребру назначается минимальная позиция, которой соответствует символ ребра в *Text* (показаны синим цветом)

Теперь мы можем преобразовать модифицированную суффиксную попытку в суффиксное дерево следующим образом. В суффиксном дереве из основного текста каждое ребро в *SuffixTree*(«panamabananas$») обозначается строкой символов *String*(*edge*). Как мы уже упоминали в основном тексте главы, хранение всех этих строк требует большого объема памяти, поэтому вместо этого мы будем обозначать ребро двумя целыми числами: начальная позиция первого вхождения *String*(*edge*) в *Text*, обозначаемая как *Position*(*edge*), и его длина, обозначаемая как *Length*(*edge*). Для измененного суффиксного дерева *Text* = «panamabananas$», показанного на рисунке ниже, эти два целых числа окрашены в синий и красный цвета соответственно. Например, ребро с меткой «mabananas$» из исходного суффиксного дерева помечено как *Position*(*edge*) = **4** и *Length*(*edge*) = **10** на рисунке ниже. В исходном суффиксном дереве есть два ребра, помеченных как «na», и оба они помечены как *Position*(*edge*) = **2** и *Length*(*edge*) = **2** на рисунке ниже.

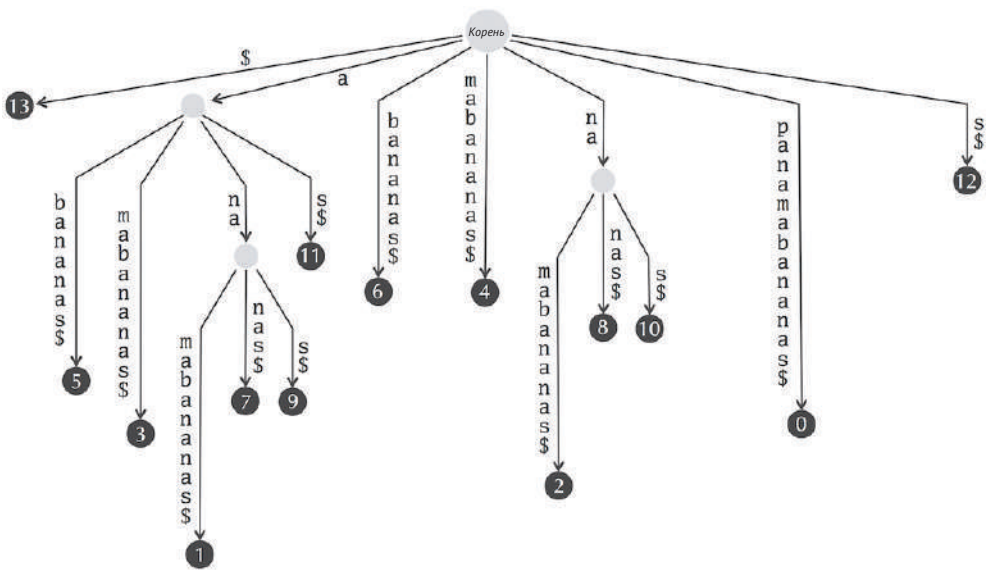


Рис. 9.41 (Вверху) Суффиксное дерево *Text* = «panamabananas$», воспроизведенное из основного текста главы. (Внизу) Модифицированное суффиксное дерево *Text* = «panamabananas$». Для каждого ребра начальная позиция подстроки, которой оно соответствует в *Text*, показана синим цветом, а длина этой подстроки показана красным

Следующий псевдокод создает суффиксное дерево, используя модифицированную суффиксную попытку, сконструированную **ModifiedSuffixTrieConstruction**. Этот алгоритм объединит каждый неветвящийся путь измененной суффиксной попытки в одно ребро. Чтобы узнать больше о неветвящихся путях, вспомните раздел **ЗАРЯДНАЯ СТАНЦИЯ: Максимальное количество неветвящихся путей на графе**.

```
ModifiedSuffixTreeConstruction(Text)
    Trie ← ModifiedSuffixTrieConstruction
    for каждого неветвящегося пути Path в Trie
        замените Path одним ребром e, соединяющим первый и последний узлы
        Path
        Position(e) ← Position(первое ребро Path)
        Length(e) ← количество ребер Path
    return Trie
```

Решение задачи самой длинной общей подстроки

Наивный подход к поиску самой длинной общей подстроки строк $Text_1$ и $Text_2$ заключается в построении одного суффиксного дерева для $Text_1$ и другого для $Text_2$. Вместо этого мы добавим «#» в конец $Text_1$, добавим «$» в конец $Text_2$, а затем построим единое суффиксное дерево для конкатенации $Text_1$ и $Text_2$ (рис. 9.42). Мы окрашиваем лист в этом суффиксном дереве в синий цвет, если он помечен начальной позицией суффикса, начинающегося в $Text_1$; окрашиваем лист в красный цвет, если он помечен начальной позицией суффикса, начинающегося в $Text_2$.

Мы также окрашиваем оставшиеся узлы суффиксного дерева в синий, красный и фиолетовый цвета в соответствии со следующими правилами:

- узел окрашен в синий или красный цвет, если листья в его поддереве (т. е. в поддереве под ним) все синие или все красные соответственно;
- узел окрашен в фиолетовый цвет, если его поддерево содержит как синие, так и красные листья.

Мы используем $Color(v)$ для обозначения цвета узла v.

На рис. 9.42 есть три фиолетовых узла (кроме корня), из рис. 9.41, и строки, написанные от корня до каждого из этих узлов, – это «a», «ana» и «na». Обратите внимание, что эти три подстроки являются общими для $Text_1$ = «panama» и $Text_2$ = «bananas». Это не случайно, как показано в следующих двух упражнениях.

Упражнение. Докажите, что путь, заканчивающийся фиолетовым узлом в суффиксном дереве $Text_1$ и $Text_2$, представляет собой подстроку, общую для $Text_1$ и $Text_2$.

Упражнение. Докажите, что путь, заканчивающийся синим (соответственно красным) узлом в суффиксном дереве $Text_1$ и $Text_2$, образует подстроку, которая появляется в $Text_1$, но не в $Text_2$ (соответственно, в $Text_2$, но не в $Text_1$).

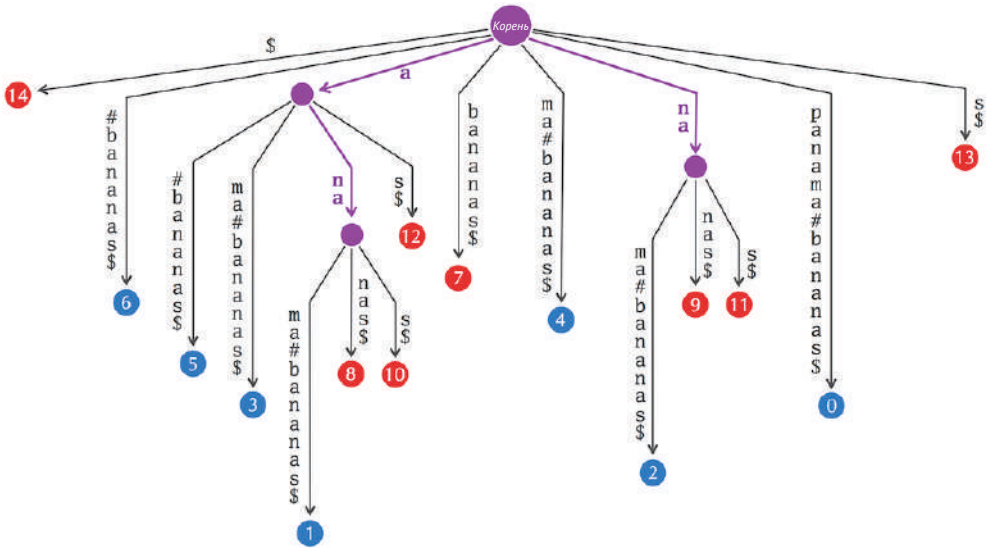


Рис. 9.42 *SuffixTree*(«**panama#bananas$**»), созданный для $Text_1$ = «**panama**» и $Text_2$ = «**bananas**». Листья, соответствующие суффиксам, начинающимся с «**panama#**», окрашены в синий цвет; листья, соответствующие суффиксам, начинающимся с «**bananas**», окрашены в красный. Каждая строка символов, написанная от корня до фиолетового узла, соответствует подстроке, общей для $Text_1$ и $Text_2$

Эти два упражнения подразумевают, что для того, чтобы найти самую длинную общую подстроку между $Text_1$ и $Text_2$, нам нужно проверить все фиолетовые узлы, а также строки, написанные путями, ведущими к фиолетовым узлам. Самая длинная такая строка дает решение проблемы самой длинной общей подстроки.

Алгоритм **TreeColoring**, псевдокод которого показан ниже и проиллюстрирован дальше, окрашивает узлы суффиксного дерева от листьев вверх. Этот алгоритм предполагает, что листья суффиксного дерева помечены как «синие» или «красные», а все остальные узлы помечены как «серые». Узел в дереве называется **зрелым**, если он серый, но не имеет серых потомков.

```
TreeColoring(ColoredTree)
  while ColoredTree имеет зрелые узлы
    for для каждого зрелого узла v in ColoredTree
      if если существуют разноцветные потомки v
        Color(v) ← «purple»
      else
        Color(v) ← цвет всех потомков v
  return ColoredTree
```

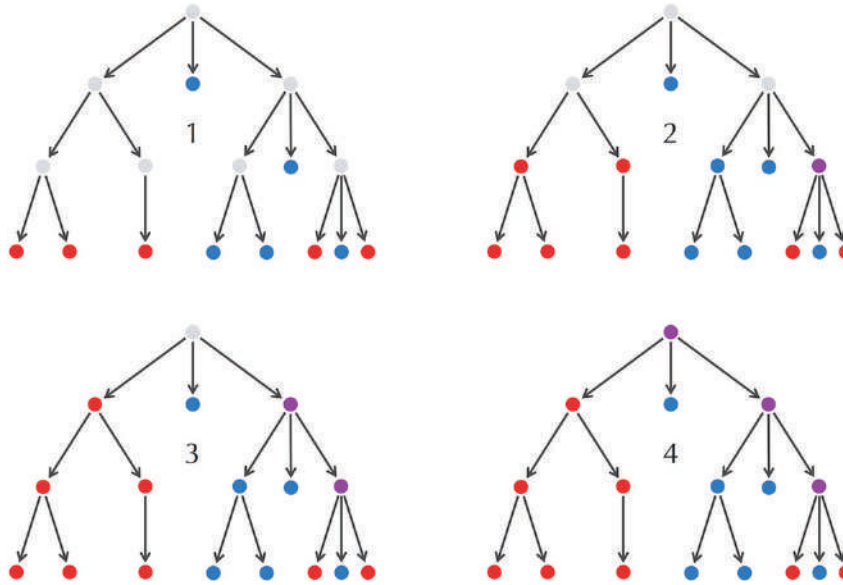


Рис. 9.43 Иллюстрация шагов, предпринятых **TreeColoring** для дерева с начальной окраской листьев в красный или синий цвет (вверху слева)

Задача раскраски дерева: *раскрасьте внутренние узлы дерева, используя цвета его листьев.*

Input: список соответствия, за которым следуют цветные метки для конечных узлов.

Output: цветные метки для всех узлов, в любом порядке.

Упражнение. Измените алгоритм раскраски, чтобы найти самую длинную общую подстроку из более чем двух строк.

Построение частичного суффиксного массива

Чтобы построить частичный суффиксный массив $SuffixArray_k(Text)$, нам сначала нужно построить полный суффиксный массив, а затем сохранить только те элементы этого массива, которые делятся на K, вместе с их индексами i. Это показано ниже для *Text* = «panamabananas» при $K = 5$, где $SuffixArray_k(Text)$ соответствует элементам, помеченным полужирным шрифтом.

I	0	**1**	2	3	4	5	6	7	8	9	10	**11**	**12**	13
SuffixFrray (*Text*)	13	**5**	3	1	7	9	11	6	4	2	8	**10**	**0**	12

Эталонный геном человека

Когда геномы собираются из ДНК, взятой у ряда доноров, эталонный геном представляет собой мозаику донорских геномов. Существующий эталонный геном человека был получен от 13 добровольцев в Соединенных Штатах; его продолжают улучшать, исправляя ошибки и заполняя оставшиеся пробелы (на данный момент их более сотни).

Эталонный геном часто используется в качестве образца, на котором можно быстро собрать новые индивидуальные геномы. В регионах с высоким разнообразием эталонный геном может значительно отличаться от индивидуальных геномов. Примером области генома человека с высоким разнообразием является **главный комплекс гистосовместимости**, семейство генов, обеспечивающих работу иммунной системы. Поскольку эти гены имеют необычно большое количество альтернативных форм, у двух особей вряд ли когда-либо будет точно такой же набор генов из этого комплекса. Сравнение эталонного генома и индивидуальных геномов человека обычно выявляет около 3 млн различий SNP, и около 0,1 % индивидуального генома человека вообще не могут быть выравниваны с эталонным геномом.

Излишне говорить, что эталонный геном, полученный всего от 13 человек в Соединенных Штатах, является очень предвзятым представителем миллиардов людей, живущих во всем мире. Эта предвзятость ограничивает выводы, которые можно сделать на основе эталонного генома в развивающейся области персонализированной медицины. По этой причине исследователи сейчас работают над новой формой эталонного генома, содержащей тысячи отдельных геномов человека. Этот эталонный **пангеном** можно представить в виде гигантского графа, в котором отдельные геномы соответствуют путям.

Рекомбинации, вставки и делеции в геномах человека

До недавнего времени биологи сосредоточивались в основном на небольших мутациях в геноме человека, предполагая, что рекомбинации и вставки относительно редки. В 2005 году Эван Эйхлер удивил биологов, обнаружив сотни рекомбинаций и вставок, разделяющих геномы двух особей. Это открытие было важно, потому что рекомбинации и вставки часто являются символами болезни; например, повторные вставки триплета нуклеотидов CAG увеличивают тяжесть болезни Хантингтона.

В 2013 году Гертон Лантер раскрыл истинные масштабы вставок в человеческой популяции, выявив более миллиона вставок в группе более чем из ста человек. Интересно, что он обнаружил, что более половины вставок приходится всего на 4 % генома; другими словами, некоторые области генома человека представляют собой «горячие точки вставок». По мере роста каталога рекомбинаций человеческого генома биологи получают возможность идентифицировать часто мутировавшие гены, а также вовлекать перестановки и вставки при диагностике сложных заболеваний.

Алгоритм Ахо–Корасик

Алгоритм Ахо–Корасик для задачи множественного сопоставления паттернов был разработан Альфредом Ахо и Маргарет Корасик в 1975 году. Время работы их алгоритма равно $O(|Patterns| + |Text| + m)$, где m – обнаруженное количество совпадений.

Представьте, что попытка trie скользит по *Text* = «bantenna». Как и **TrieMatching**, алгоритм Ахо–Корасик начинает с корня и пытается построить путь, состоящий из префикса «bantenna». Эта попытка терпит неудачу после трех узлов («**ban**tenna»). В **TrieMatching** мы снова начинаем с корня и пытаемся найти совпадение, начиная со 2-й позиции «bantenna».

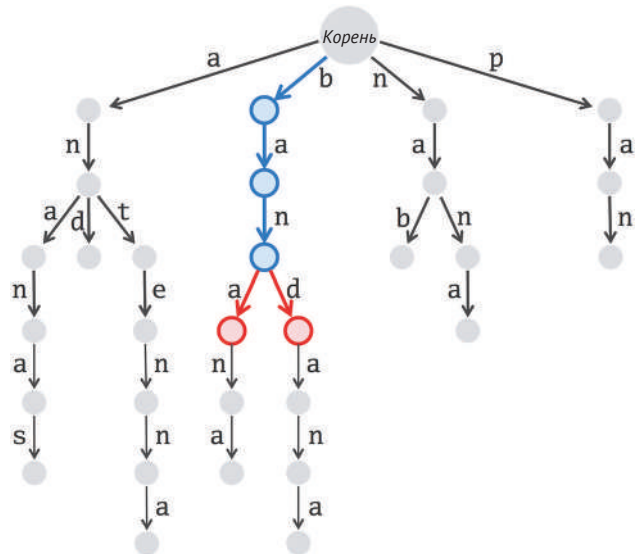


Рис. 9.44 TrieMatching

Однако мы упустили важную информацию: первые два символа «**an**tenna» *мы уже нашли* при поиске по «**ban**tenna». Таким образом, более разумная стратегия состоит в том, чтобы перейти прямо к узлу «an», а затем продолжить движение вниз по дереву. Действительно, эта стратегия в конечном итоге будет соответствовать паттерну «**antenna**». Мы можем реализовать эту стратегию «забегания вперед», дополнив дерево **ребром отказа**, соединяющим узел «ban» с узлом «an» (показано на рисунке ниже). В более общем случае ребра отказа формируются путем соединения узла v с узлом ω, если ω является самым длинным суффиксом v, который появляется в дереве. После создания всех ребер отказа алгоритм Ахо–Корасик следует ребрам отказа во время выравнивания с паттерном всякий раз, когда обнаруживается несовпадение, чтобы избежать возврата к корню, что экономит время.

Упражнение. Постройте все ребра отказа для дерева, показанного на рисунке ниже.

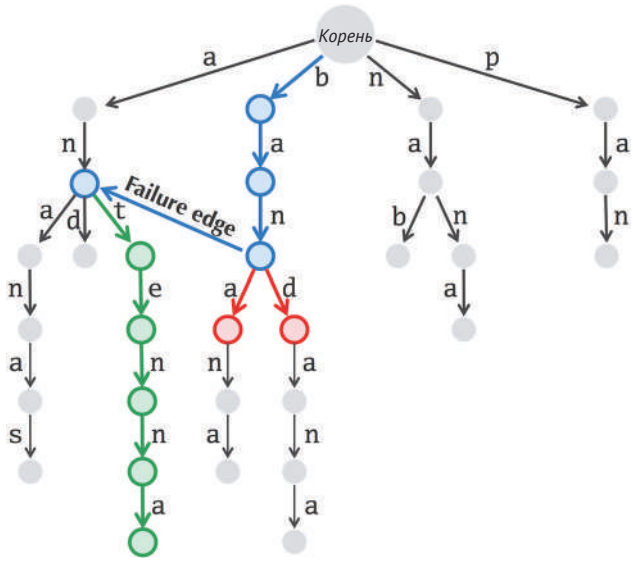


Рис. 9.45 Рассматриваемая в основном тексте суффиксная попытка с дополнительным ребром отказа. Ребро отказа позволяет нам непосредственно перейти с пути, начинающегося с «ban», на путь, начинающийся с «an»

Суффиксные массивы и суффиксные деревья

Мы можем построить *SuffixArray*(*Text*) из *SuffixTree*(*Text*), применяя **предварительный проход** *SuffixTree*(*Text*) до тех пор, пока исходящие ребра из каждого узла суффиксного дерева расположены лексикографически.

Для корневого дерева узел ω называется **дочерним** по отношению к узлу v, если в дереве есть ребро, соединяющее v с ω. Предварительный проход дерева в прямом порядке включает в себя посещение узла дерева, начиная с корня, а затем рекурсивный проход в прямом порядке поддеревьев, корневых в каждом из его дочерних элементов, слева направо (рис. 9.46).

Предварительные проходы выполняются с помощью следующего псевдокода.

```
Preorder(Tree, Node)
    посетить Node
    for каждого дочернего элемента Node' из Node, слева направо
        Preorder(Tree, Node')
```

Принимая порядок посещаемых листьев при обходе дерева суффиксов в прямом порядке, мы получаем массив суффиксов, как показано на рис. 9.47.

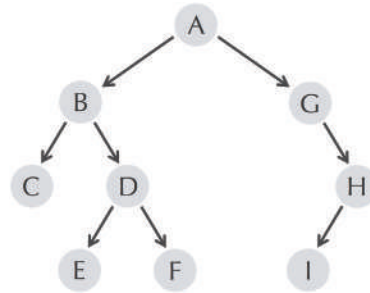
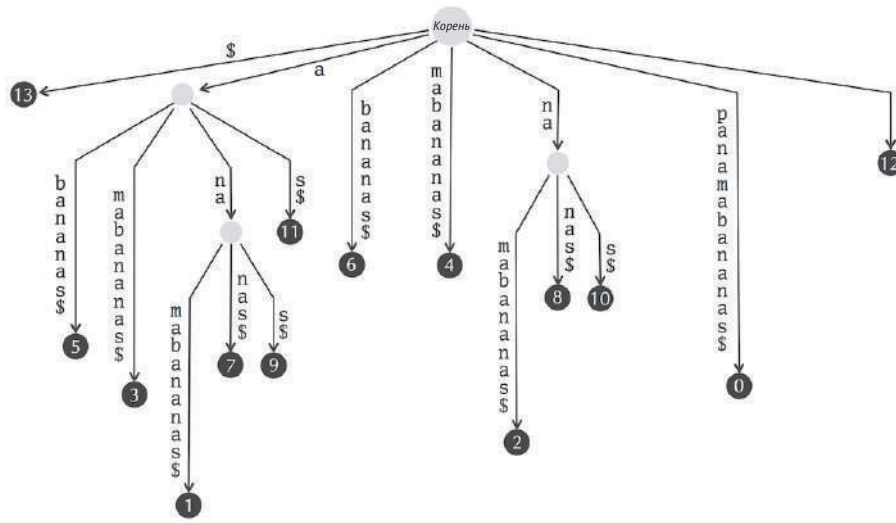


Рис. 9.46 Предварительный проход в прямом порядке вышеприведенного дерева посещает его узлы в порядке возрастания их меток



Стартовая позиция	Отсортированные суффиксы
13	$
5	abananas$
3	amabananas$
1	anamabananas$
7	ananas$
9	anas$
11	as$
6	bananas$
4	mabananas$
2	namabananas$
8	nanas$
10	nas$
0	panamabananas$
12	s$

Рис. 9.47 Полученный суффиксный массив

И наоборот, *SuffixTree*(*Text*) может быть построен за линейное время из *SuffixArray*(*Text*) с использованием массива **самого длинного общего префикса**, **LCP-массива** текста, *LCP*(*Text*), в котором хранится длина самого длин-

ного общего префикса, разделяемого последовательными лексикографически упорядоченными суффиксами *Text*. Например, *LCP*(«panamabananas$») равно (0, 0, 1, 1, 3, 3, 1, 0, 0, 0, 2, 2, 0, 0), как показано на рис. 9.48.

Задача построения суффиксного дерева из суффиксного массива: *построить суффиксное дерево из суффиксного массива и LCP-массива строки.*

Input: строка *Text*, ее суффиксный массив и массив *LCP*.

Output: суффиксное дерево *Text*.

LCP-массив	Отсортированные суффиксы
0	$
0	abananas$
1	amabananas$
1	anamabananas$
3	ananas$
3	anas$
1	as$
0	bananas$
0	mabananas$
0	namabananas$
2	nanas$
2	nas$
0	panamabananas$
0	s$

Рис. 9.48 LCP-массив и отсортированные суффиксы

Имея суффиксный массив *SuffixArray* и LCP-массив *LCP* строки *Text*, суффиксное дерево *SuffixTree*(*Text*) можно построить за линейное время, используя алгоритм, показанный на рисунке далее. После построения **частичного суффиксного дерева** для i лексикографически наименьших суффиксов (обозначается $SuffixTree_i(Text)$) этот алгоритм итеративно вставляет $(i+1)$-й суффикс в это дерево, чтобы сформировать $SuffixTree_{i+1}(Text)$.

Мы определяем **спуск узла** v в суффиксном дереве, обозначаемый $Descent(v)$, как длину конкатенации всех меток пути от корня до этого узла. Мы предполагаем, что спуски всех узлов в растущем частичном суффиксном дереве были предварительно вычислены.

Начнем с $SuffixTree_0(Text)$, которое мы определяем как дерево, состоящее только из корня. Чтобы вставить $(i + 1)$-й суффикс (соответствующий элементу

SuffixArray(*i* + 1) в массиве суффиксов *Text*) в *SuffixTree*$_i$(*Text*), нам нужно знать, где путь, представляющий этот суффикс, «расщепляется» от уже построенного частичного суффиксного дерева *SuffixTree*$_i$(*Text*). Как показано на рис. 9.49, это разделение происходит на фиолетовом узле при вставке «anamabananas$» в *SuffixTree*$_3$(*Text*) и на фиолетовом ребре, помеченном «namabananas$», при вставке «ananas$» в *SuffixTree*$_4$(*Text*); таким образом, это ребро разбивается на ребра, помеченные «na» и «mabananas$» в *SuffixTree*$_5$(Text).

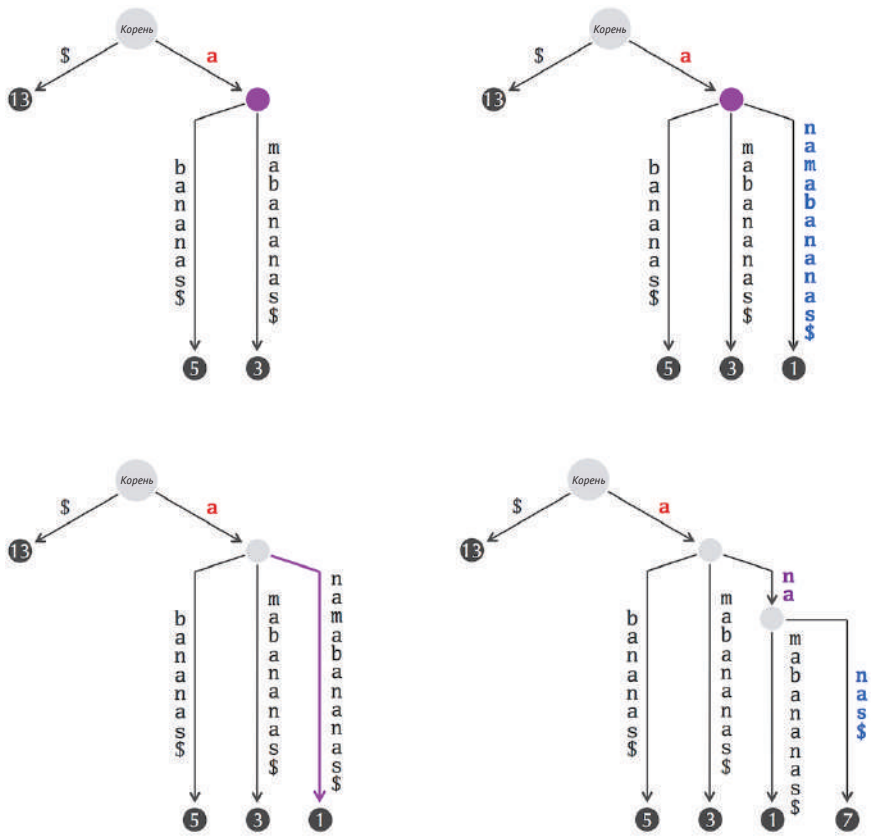


Рис. 9.49 (Вверху слева) *SuffixTree*$_3$(*Text*) для *Text* = «panamabananas$». Четвертый лексикографически упорядоченный суффикс *Text* – «**anamabananas$**», и мы можем вставить только его первую букву в это суффиксное дерево. Наша точка остановки находится в фиолетовом узле, поэтому мы создаем новый узел, ответвляющийся от этого узла с ребром, помеченным «**namabananas$**», чтобы получить *SuffixTree*$_4$(*Text*) (вверху справа). (Внизу слева) Пятым лексикографически упорядоченным суффиксом *Text* является «**a**n**a**n**as$**», первые три символа которого мы можем вставить в *SuffixTree*$_4$(*Text*), пока не достигнем точки остановки в середине ребра «namabananas$». (Внизу справа) Чтобы сформировать *SuffixTree*$_5$(*Text*), мы создаем новый узел в середине фиолетового ребра и разветвляемся от этого узла на два ребра: одно помечено как «mabananas$», чтобы сохранить суффикс «anamabananas$», а другое помечено как «**nas$**» для написания нового суффикса «**a**n**a**n**as$**». В общем, количество красных символов, которые мы записываем в *SuffixTree*$_i$(*Text*) для формирования *SuffixTree*$_{i+1}$(*Text*), задается (*i* + 1)-м входом в массиве LCP *Text*

Чтобы найти узел/ребро, где разбивается частичное суффиксное дерево, мы пройдем по самому правому пути в частичном дереве суффиксов (т. е. последнему добавленному пути), начиная с ранее вставленного листа, помеченного как *SuffixArray*(*i*), до самого корня. Мы остановимся, когда встретим первый узел v такой, что *Descent*(v) ≤ *LCP*(*i* + 1). После этого мы должны рассмотреть два случая, *Descent*(v) = *LCP*(*i* + 1) (расщепление происходит в узле v) или *Descent*(v) < *LCP*(*i* + 1) (расщепление происходит на ребре, ведущем от v).

Если *Descent*(v) = *LCP*(*i* + 1), то конкатенация меток на пути от корня к v равна самому длинному общему префиксу суффиксов, соответственно, *SuffixArray*(*i*) и *SuffixArray*(*i* + 1). В этом случае мы вставляем *SuffixArray*(*i* + 1) как новый лист x, соединенный с v, и помечаем ребро (v, x) суффиксом *Text*, начиная с позиции *SuffixArray*(*i* + 1) + *LCP*(*i* + 1). Таким образом, метка ребра состоит из оставшихся символов суффикса, соответствующего *SuffixArray*(*i* + 1), который еще не представлен конкатенацией меток пути, соединяющего корень с v. Это завершает построение частичного суффиксного дерева $SuffixTree_{j+1}(Text)$.

Если *Descent*(v) < *LCP*(*i* + 1), то конкатенация меток на пути от корня к v содержит меньше символов, чем самый длинный общий префикс суффиксов, соответствующих *SuffixArray*(*i*) и *SuffixArray*(*i* + 1). Поэтому возникает вопрос, как восстановить эти недостающие символы. Мы обозначаем самое правое ребро, ведущее из v в $SuffixTree_i(Text)$, как (v, ω) и утверждаем, что отсутствующие символы представляют собой префикс метки этого ребра. В этом случае мы разделяем это ребро и строим $SuffixTree_{i+1}(Text)$, как описано ниже.

1. Удалите ребро (v, ω) из $SuffixTree_i(Text)$.
2. Добавьте новый внутренний узел y и новое ребро (v, y), помеченное подстрокой *Text*, начиная с позиции *SuffixArray*(*i* + 1) + *Descent*(v) и заканчивая позицией *SuffixArray*(*i*) + *LCP*(*i* + 1) – 1. Новая метка формируется из последних *LCP*(*i* + 1) – *Descent*(v) символов самого длинного общего префикса *SuffixArray*(*i*) и *SuffixArray*(*i* + 1). Таким образом, конкатенация меток на пути от корня к y теперь является самым длинным общим префиксом *SuffixArray*(*i*) и *SuffixArray*(*i* + 1).
3. Определите *Descent*(y) как *LCP*(*i* + 1).
4. Соедините ω с вновь созданным внутренним узлом y ребром (y, ω), помеченным подстрокой *Text*, начинающейся в позиции *SuffixArray*(*i*) + *LCP*(*i* + 1) и заканчивающейся в позиции *SuffixArray*(*i*) + *Descent*(ω) – 1. Новая метка состоит из оставшихся символов удаляемого ребра (v, ω), которые не использовались в качестве метки ребра (v, y).
5. Добавьте *SuffixArray*(*i* + 1) в качестве нового листа x, а также ребро (y, x), которое помечено суффиксом *Text*, начинающимся с позиции *SuffixArray*(*i* + 1) + *LCP*(*i* + 1). Метка этого ребра состоит из оставшихся символов суффикса, соответствующего *SuffixArray*(*i* + 1), которые еще не представлены конкатенацией меток на пути от корня к v.

ОСТАНОВИТЕСЬ и задумайтесь. Докажите, что время работы этого алгоритма равно $O(|Text|)$.

Бинарный поиск

В игровом шоу «Цена верна» есть испытание на время, называемое «Игра с часами», в котором участник повторяет предположения о цене предмета, а ведущий сообщает участнику только то, выше или ниже истинная цена, чем последняя предложенная.

Разумная стратегия игры с часами состоит в том, чтобы выбрать разумный диапазон цен, в пределах которого должна находиться цена предмета, а затем угадать цену, находящуюся посередине между этими двумя крайностями. Если это предположение неверно, то участник немедленно исключает половину диапазона возможных цен. Затем участник делает предположение в середине оставшегося диапазона цен, снова исключая его половину. Повторение этой стратегии быстро дает цену предмета.

☞ **Посмотреть на YouTube https://www.youtube.com/watch?v=oc9H8bo8yg0**

Эта стратегия «Игры с часами» мотивирует алгоритм бинарного поиска находить позицию элемента *key* в отсортированном массиве *Array*. Этот алгоритм, называемый **BinarySearch**, инициализируется установкой *minIndex* на величину 0, а *maxIndex* равным последнему индексу массива *Array*. Он устанавливает *midIndex* равным (*minIndex* + *maxIndex*)/2, а затем проверяет, больше или меньше *key*, чем *Array*(*midIndex*). Если *key* больше этого значения, то **BinarySearch** выполняет итерацию по подмассиву *Array* от *minIndex* до *midIndex* – 1; в противном случае **BinarySearch** перебирает подмассив *Array* от *midIndex* + 1 до *maxIndex*. Итерация в конечном итоге определяет положение *key*.

Например, если *key* = 9 и *Array* = (1, 3, 7, 8, 9, 12, 15), то **BinarySearch** сначала установит *minIndex* равным 0, *maxIndex* равным 6 и *midIndex* равным 3. Поскольку *key* больше, чем *Array*(*midIndex*) = 8, мы проверяем подмассив, элементы которого больше, чем *Array*(*midIndex*), устанавливая *minIndex* равным 4, так что *midIndex* пересчитывается как (4 + 6)/2 = 5. На этот раз *key* меньше, чем *Array*(*midIndex*) = 12, поэтому мы исследуем подмассив, элементы которого меньше этого значения. Этот подмассив состоит только из одного элемента, что и является *key*.

```
BinarySearch(Array, key, minIndex, maxIndex)
    while maxIndex ≤ minIndex
        midIndex ← ⌊(minIndex + maxIndex)/2⌋
        if Array(midIndex) = key
            return midIndex
        else if Array(midIndex) < key
            minIndex ← midIndex + 1
        else
            maxIndex ← midIndex – 1
    return «key не найден»
```

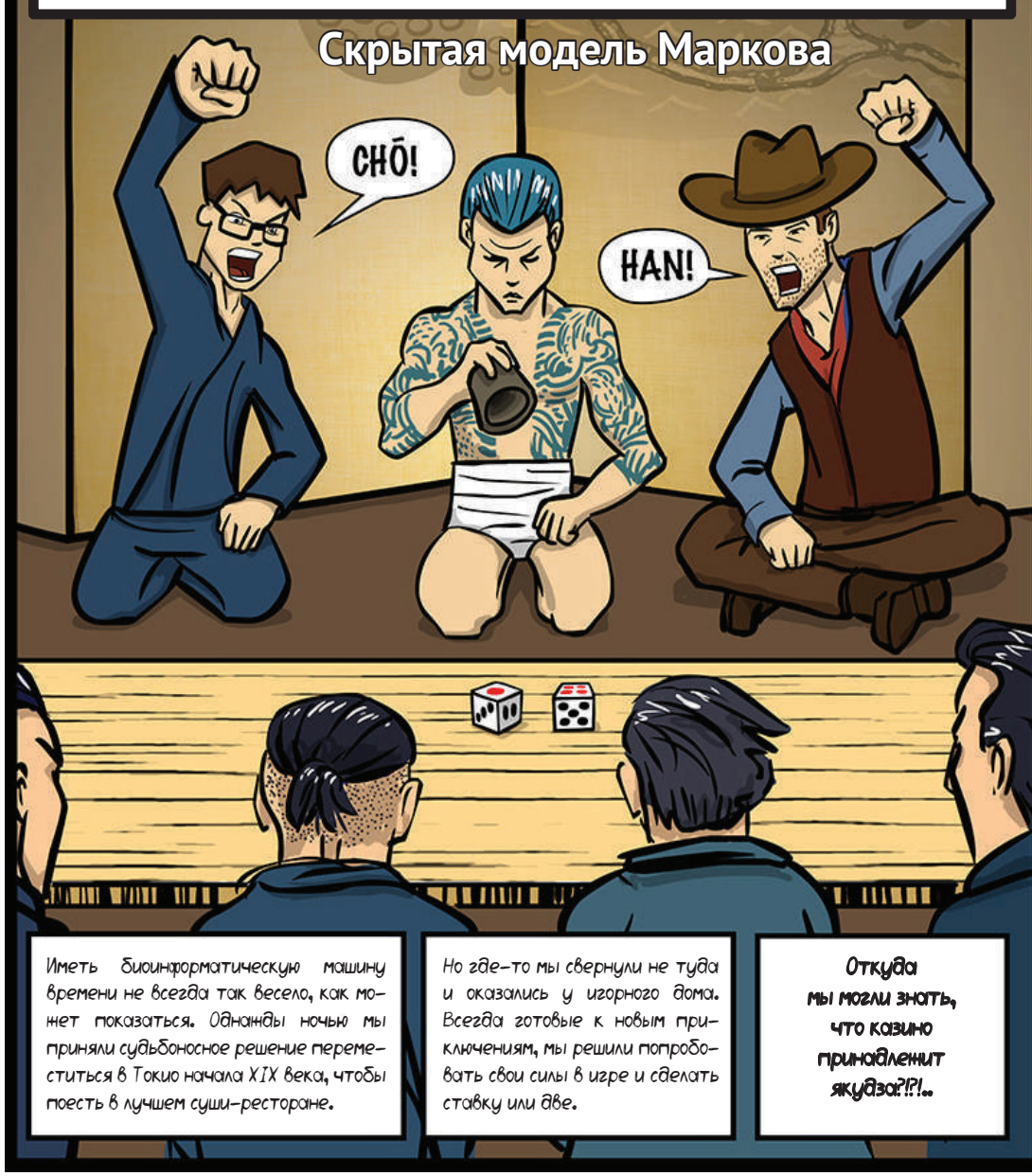
Библиографические примечания

Алгоритм Ахо–Корасик был представлен Aho and Corasick, 1975[1]. Суффиксные деревья были введены Weiner, 1973[2]. Суффиксные массивы были введены Manber and Myers, 1990[3]. Эффективная реализация преобразования Барроуза–Уилера была описана Ferragina and Manzini, 2000[4]. Генетическая причина синдрома Одо была выяснена Clayton-Smith et al., 2011[5]. Рекомбинации и вставки в геноме человека изучались Tuzun et al., 2005[6], и Montgomery et al., 2013[7]. BLAST, доминирующий инструмент поиска в базе данных в молекулярной биологии, был разработан Altschul et al., 1990[8].

[1] https://www.researchgate.net/publication/220423622_Efficient_string_matching_An_aid_to_bibliographic_search.
[2] https://www.researchgate.net/publication/229067733_Linear_Pattern_Matching_Algorithm.
[3] https://epubs.siam.org/doi/10.1137/0222058.
[4] https://stepik.org/lesson/240387/step/Opportunistic%20Data%20Structures%20with%20Applications.
[5] https://www.ncbi.nlm.nih.gov/pubmed/22077973.
[6] https://www.ncbi.nlm.nih.gov/pubmed/15895083.
[7] https://www.ncbi.nlm.nih.gov/pubmed/23478400.
[8] https://www.ncbi.nlm.nih.gov/pubmed/2231712.

Глава 10

Почему биологи до сих пор не разработали вакцину от ВИЧ?



Классификация фенотипа ВИЧ

Каким образом ВИЧ ускользает от иммунной системы человека?

В 1984 году министр здравоохранения и социальных служб США Маргарет Хеклер заявила, что вакцина против ВИЧ (HIV) будет получена в течение двух лет, заявив: «Еще одна ужасная болезнь скоро уступит место терпению, настойчивости и откровенной гениальности».

В 1997 году Билл Клинтон основал новый исследовательский центр в Национальном институте здравоохранения с целью разработки вакцины против ВИЧ. По его словам, «это больше не вопрос, сможем ли мы разработать вакцину против СПИДа, это просто вопрос, когда».

В 2005 году компания Merck начала клинические испытания вакцины против ВИЧ, но прекратила их через два года после того, как узнала, что вакцина фактически увеличивает риск заражения ВИЧ у некоторых вакцинированных.

Сегодня, несмотря на огромные инвестиции и продолжающиеся клинические испытания, мы все еще далеки от вакцины против ВИЧ, и 35 млн человек живут с этим заболеванием. Ученые добились больших успехов в разработке успешной **антиретровирусной терапии** – лекарственного коктейля, стабилизирующего состояние инфицированного пациента. Однако эта терапия не излечивает СПИД и не может предотвратить распространение ВИЧ, поэтому она не обещает создать настоящую вакцину для сдерживания эпидемии СПИДа.

Классические вакцины против вирусов часто изготавливаются из белков оболочки вируса. Эти вакцины стимулируют иммунную систему человека распознавать белки вируса как чужеродные, уничтожать их и запоминать, чтобы можно было идентифицировать и уничтожить вирус при более позднем контакте.

Однако белки оболочки вируса ВИЧ чрезвычайно изменчивы, потому что вирус должен быстро мутировать, чтобы выжить (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Эффект Красной Королевы**). Популяция ВИЧ у одного инфицированного человека быстро эволюционирует, чтобы уклониться от иммунной системы человека (рис. 10.1), не говоря уже о том, что штаммы ВИЧ, взятые у разных пациентов, представляют собой несколько сильно отличающихся подтипов. Следовательно, успешная вакцина против ВИЧ должна быть достаточно универсальной, чтобы учитывать эту изменчивость.

Стремясь противодействовать изменчивости ВИЧ, мы могли бы создать единый пептид, содержащий наименее вариабельные сегменты оболочечных белков, взятых из всех известных штаммов ВИЧ, и использовать этот пептид в качестве основы для универсальной вакцины, борющейся со всеми штаммами ВИЧ. Однако белки оболочки ВИЧ не только быстро мутируют, но и «маскируются» **гликозилированием**, посттрансляционной модификацией, которая часто делает эти белки невидимыми для иммунной системы человека (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Гликозилирование**). В результате все попытки разработать вакцину против ВИЧ пока не увенчались успехом.

Рис. 10.1 Множественное выравнивание короткой области белков gp120, взятых у одного ВИЧ-положительного пациента в девять разных моментов времени. Почти половина столбцов (показаны более темным текстом) с течением времени не сохраняются, что показывает, насколько быстро мутирует ВИЧ даже у отдельного хозяина. Аминокислоты, отличающиеся от наиболее часто встречающегося символа в столбце, показаны синим цветом

У ВИЧ всего девять генов, и в этой главе мы сосредоточимся на быстро мутирующем гене *env*, частота мутаций которого составляет от 1 до 2 % на нуклеотид в год. Белок, кодирующий ген *env*, мы разрезаем на **гликопротеин gp120** (примерно 480 аминокислот) и **гликопротеин gp41** (примерно 345 аминокислот). Вместе gp120 и gp41 образуют спайк оболочки вируса ВИЧ, который обеспечивает его проникновение в клетки человека.

Поскольку ВИЧ мутирует очень быстро, разные изоляты ВИЧ могут иметь разные фенотипы, что требует разных комбинаций препаратов. Например, вирусы ВИЧ (HIV) можно разделить на быстрореплицирующиеся изоляты, **индуцирующие синцитий (SI)**, и изоляты, медленно реплицирующиеся, **не индуцирующие синцитий (NSI)**. Во время инфекции вирусные белки, такие как gp120, которые используются ВИЧ для проникновения в клетку, переносятся на клеточную поверхность, где они могут вызвать слияние мембраны клетки-хозяина с соседними клетками. Это заставляет десятки человеческих клеток сливать свои клеточные мембраны в гигантский нефункциональный синцитий или аномальную многоядерную клетку (рис. 10.2). Этот механизм позволяет вирусу SI убивать множество клеток человека, заражая только одну.

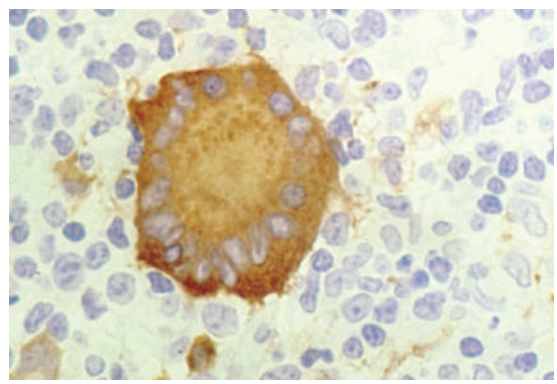


Рис. 10.2 Синцитий с несколькими ядрами у больного ВИЧ

Поскольку gp120 важен для классификации вируса как SI или NSI, биологи заинтересованы в определении того, какие аминокислоты gp120 можно использовать для этой классификации.

В 1992 году Жан-Жак Де Йонг проанализировал множественное выравнивание области **петли V3** в gp120 (рисунок ниже) и разработал правило 11/25, которое утверждает, что штамм ВИЧ с большей вероятностью будет иметь фенотип SI, если в положениях 11 или 25 его петли V3 находятся аминокислоты аргинин или лизин. Позже было показано, что и многие другие позиции влияют на фенотип SI/NSI.

Рис. 10.3 Множественное выравнивание области петли V3, взятой у 20 пациентов с ВИЧ. 11-й и 25-й столбцы выравнивания показаны более темным текстом; вхождения аргинина (R) или лизина (K) в этих столбцах показаны красным цветом. Правило 11/25 классифицирует шесть пациентов как инфицированных изолятом SI. Хотя петля V3 является важным и довольно консервативным сегментом gp120, уровень консервативности варьирует в разных позициях. Например, в то время как первая и последняя позиции чрезвычайно консервативны, позиции 11 и 25 демонстрируют высокую изменчивость

Ограничения метода выравнивания последовательностей

Еще до того, как биологи смогли приступить к изучению вопроса предсказания фенотипов ВИЧ по последовательностям gp120, они столкнулись с задачей построения точных множественных выравниваний этих последовательностей. Действительно, единственное смещение, размещение неправильной аминокислоты в положении, влияющем на фенотип SI/NSI, может привести к ошибочной классификации фенотипов ВИЧ. И мы уже знаем, что построение

множественного выравнивания сильно дивергированных последовательностей – сложная алгоритмическая задача.

Рисунок ниже на уровне величины логотипа мотива из петли V3 gp120 иллюстрирует, что некоторые позиции в gp120 относительно консервативны, тогда как другие чрезвычайно изменчивы. Кроме того, логотип этого мотива не учитывает вставки и делеции, преобладающие в других областях gp120, которые менее консервативны, чем петля V3. Эти вставки и делеции делают анализ gp120 еще более сложным.

Поскольку столбцы этого множественного выравнивания имеют разные уровни консервативности, мы сомневаемся в целесообразности использования одной и той же оценочной матрицы аминокислот (а также штрафов за вставки) в разных столбцах выравнивания. Лучшим подходом было бы использование *другого* метода к оценке в разных столбцах. Например, аминокислота, отличающаяся от R в положении 3 приведенного ниже выравнивания, должна подвергаться большему штрафу, чем аминокислота, отличающаяся от S в положении 11.

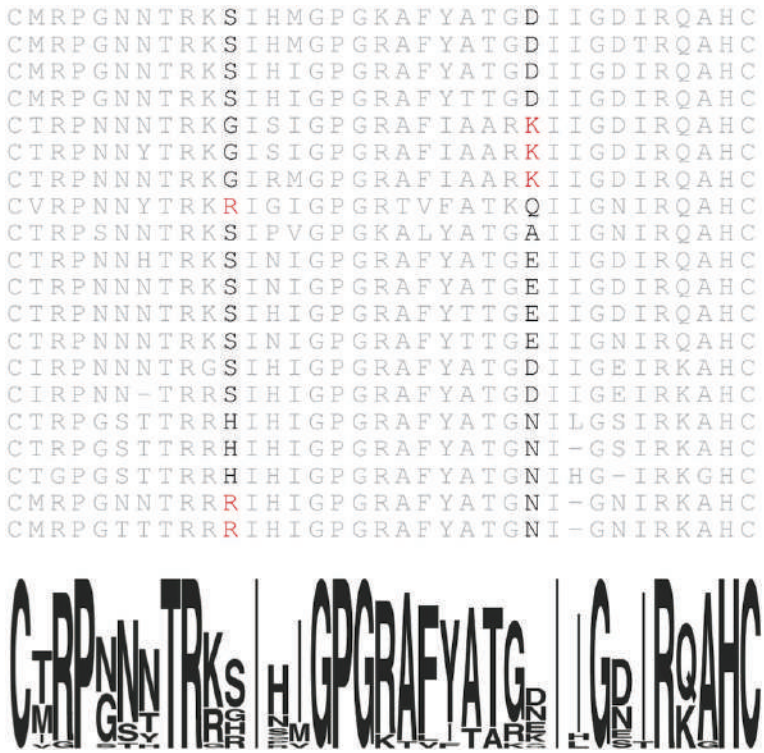


Рис. 10.4 Выравнивание из рис. 10.3 вместе с мотивом логотипа, созданным из этого выравнивания

Другими словами, *формулировка задачи* множественного выравнивания последовательностей, представленная в предыдущей главе, не предлагает адек-

ватного перевода биологической проблемы классификации ВИЧ в алгоритмическую задачу. Поэтому мы должны разработать новую формулировку задачи для выравнивания последовательностей, которая приведет к статистически достоверному анализу белков gp120. Но сначала попросим вас присоединиться к нам в нашей машине времени для еще одного путешествия.

Азартные игры с якудза

Японские преступные синдикаты, называемые якудза, произошли от групп странствующих игроков XVIII века, называемых бакуто. (На самом деле «якудза» – это название проигрышной руки в японской карточной игре.) Одна из самых популярных игр, которую бакуто устраивали в своих импровизированных казино, называется **Чо-Хан**. В этой игре, которая дословно переводится как «чет-нечет», дилер бросает две кости, а игроки делают ставки на то, будет ли сумма костей четной или нечетной.

Хотя игра в Чо-Хан в игорном доме якудза, несомненно, подарит вам веселый вечер, мы можем сыграть в эквивалентную, хотя и менее захватывающую игру под названием «Орел или решка», или «Орлянка», подбрасывая монету и делая ставки на результат. Предположим, что по какой-то странной причине в этой игре больше людей ставит на решку, чем на орла. Тогда мошенник может использовать несимметричную монету, которая с большей вероятностью выпадет орлом, чем решкой. Будем считать, что эта несимметричная монета выпадает с вероятностью 3/4.

ОСТАНОВИТЕСЬ и задумайтесь/ Скажем, вы играете в орлянку 100 раз и монета выпадает орлом 63 раза. Обманывает ли вас дилер? Была ли монета честной или обманной?

Этот вопрос сформулирован неправильно, поскольку любая монета могла произвести любую последовательность бросков. Но можем ли мы определить, какая монета, скорее всего, использовалась?

Запишем вероятности выпадения решки («Т») и орла («Н») для честной монеты (F) как

$$Pr_F(\text{«Н»}) = 1/2 \qquad \Pr_F(\text{«Т»}) = 1/2$$

и вероятности для несимметричной монеты (B) как

$$Pr_B(\text{«Н»}) = 3/4 \qquad \Pr_B(\text{«Т»}) = 1/4.$$

Поскольку броски монеты являются независимыми событиями, вероятность того, что n бросков правильной монеты сгенерируют заданную последовательность $x = x_1\, x_2 \ldots x_n$ с k вхождениями «Н», равна

$$Pr(x|F) = \prod_{i=1}^{n} Pr_F(x_i) = (1/2)^n.$$

С другой стороны, вероятность того, что смещенная монета сгенерирует ту же последовательность, равна

$$Pr(x|B) = \prod_{i=1}^{n} Pr_B(x_i) = (1/4)^n \cdot (3/4)^k = 3^k/4^n.$$

Если $Pr(x|F) > Pr(x|B)$, то дилер, скорее всего, использовал честную монету, а если $Pr(x|F) < Pr(x|B)$, то дилер, скорее всего, использовал необъективную монету. Числа $(1/2)^n$ и $3^k/4^n$ настолько малы для больших n, что для их сравнения мы будем использовать их **логарифмическое отношение** шансов,

$$\log_2\left(\frac{Pr(x|F)}{Pr(x|B)}\right) = \log_2\left(\frac{2^n}{3^k}\right) = n - k \cdot \log_2 3.$$

Упражнение. Покажите, что $Pr(x|F)$ больше, чем $Pr(x|B)$, когда логарифмическое отношение шансов положительно (т. е. когда $k/n < 1/\log_2(3)$), и меньше, чем $Pr(x|B)$, когда логарифмическое отношение шансов отрицательно (т. е. когда $k/n > 1/\log_2(3)$).

Возвращаясь к нашему примеру с $k = 63$ орлами при $n = 100$ бросках, логарифмическое отношение шансов положительно, так как

$$k/n = 0{,}63 < 1/\log_2 3 \approx 0.6309.$$

Из этого следует, что $Pr(x|F) > Pr(x|B)$, поэтому дилер, скорее всего, использовал честную монету, хотя 63 ближе к 75, чем к 50.

Две монеты в рукаве у дилера

В игорных домах бакуто дилер Чо-Хан снимал рубашку во время игры, демонстрируя татуироровку на голом теле, чтобы избежать любых подозрений в манипулировании костями. (Позже эти татуировки станут традицией якудза.) Однако мы предположим ситуацию, что в игре «Орел или решка» нечестный дилер носит рубашку и держит обе монеты в рукаве, тайно меняя их одну на другую, когда ему захочется, во время последовательности бросков. Так как он не хочет, чтобы его поймали на подмене монет, он делает это лишь изредка.

Будем считать, что мошенник меняет монеты с вероятностью 0,1 после каждого броска. Имея последовательность подбрасываний монеты, мы должны определить, когда дилер использовал асимметричную монету, а когда – правильную.

Задача казино: *имея последовательность подбрасываний монеты, определите, когда нечестный дилер использовал правильную монету, а когда – мошенническую.*

Input: последовательность $x = x_1\, x_2 \ldots x_n$ подбрасываний монеты двумя возможными монетами (*F* и *B*).

Output: последовательность $\pi = \pi_1\pi_2 \ldots \pi_n$, где каждое π_i равно либо *F*, либо *B* и указывает, что x_i является результатом подбрасывания честной или мошеннической монеты соответственно.

К сожалению, эта задача плохо сформулирована, поскольку любая монета может генерировать любой результат. Вместо этого нам нужно определить наиболее вероятную последовательность монет, используемых дилером.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы переформулировать задачу казино так, чтобы она имела смысл?

Четко определенная вычислительная задача для нахождения наиболее вероятной последовательности монет, используемых дилером, должна каким-то образом оценивать разные последовательности π как лучшие ответы, чем другие. Один из подходов к угадыванию наиболее вероятной монеты, которую дилер использовал для каждого броска, состоит в том, чтобы сдвинуть окно (длиной $t < n$) вдоль последовательности бросков $x = x_1\, x_2 \ldots x_n$, а затем рассчитать отношение логарифмических шансов для каждого окна. Если логарифмическое отношение шансов окна падает ниже нуля, то дилер, скорее всего, использовал мошенническую монету внутри этого окна; в противном случае дилер, скорее всего, использовал честную монету.

ОСТАНОВИТЕСЬ и задумайтесь. Видите ли вы какие-либо проблемы с этим методом?

Есть две проблемы с движущимся окном. Во-первых, у нас нет очевидного выбора длины окна. Во-вторых, перекрывающиеся окна могут классифицировать один и тот же результат как результат как честных, так и мошеннических монет. Например, если x = «OOOOOPPOOOPPPPP», то окно $x_1 \ldots x_{10}$ = «OOOOOPPOOO» имеет отрицательное логарифмическое отношение шансов, а окно $x = x_6 \ldots x_{15}$ = «PPOOOPPPPP» имеет положительное логарифмическое отношение шансов. Итак, какую монету использовал дилер при подбрасывании $x_6 \ldots x_{10}$ = «PPOOO»?

Поиск CG-островов

В следующем разделе мы усовершенствуем наш метод оценки последовательностей подбрасывания монеты. Решение приведет нас к вычислительной парадигме, которая была успешно применена к широкому кругу задач биоинформатики, включая сопоставление геномов ВИЧ. Однако на данный момент вы, возможно, все еще не верите, как подбрасывание монеты может быть связано с выравниванием последовательностей. Таким образом, мы кратко опишем другую биологическую задачу, которая более четко связана с нашей аналогией подбрасывания монеты.

В начале XX века Феб Левен открыл четыре нуклеотида, из которых состоит ДНК. В то время мало что было известно о ДНК (до работы Уотсона и Крика по двойной спирали оставалось еще полвека). В результате Левен усомнился в том, что ДНК может хранить генетическую информацию, используя всего лишь четырехбуквенный алфавит, и предположил, что ДНК содержит почти равные количества аденина, цитозина, гуанина и тимина.

Столетие спустя мы знаем, что комплементарные нуклеотиды на противоположных цепях ДНК имеют одинаковую частоту из-за спаривания оснований, если не брать во внимание чрезвычайно редкие ошибки спаривания оснований. Однако неверно, что частоты нуклеотидов примерно равны на *одной цепи* ДНК. Например, разные виды имеют сильно различающееся **GC-контент** или процентное содержание цитозиновых и гуаниновых нуклеотидов в геноме. Например, содержание GC в геноме человека составляет примерно 42 %.

После учета аномального содержания GC в геноме человека можно ожидать, что каждый из динуклеотидов CC, CG, GC и GG будет встречаться в геноме человека с частотой $0{,}21 \cdot 0{,}21 = 4{,}41$ %. Однако частота GC в геноме человека составляет всего около 1 %! Этот динуклеотид настолько редок из-за **метилирования**, наиболее распространенной модификации ДНК, которая обычно добавляет метильную группу (CH_3) к нуклеотиду цитозина в динуклеотиде CG. Образовавшийся метилированный цитозин имеет тенденцию к дальнейшему дезаминированию в тимин (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Метилирование ДНК**). В результате метилирования CG является самым редким динуклеотидом во многих геномах.

Тем не менее метилирование часто подавляется вокруг генов в областях, называемых **CG-островами**, где CG появляется относительно часто (рис. 10.5). Если бы вам нужно было секвенировать геном млекопитающего, о котором вы ничего не знали, возможно, первое, что вы могли бы сделать, чтобы найти гены в этом геноме, – это найти CG-острова.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы определили CG-острова в геноме?

	A	C	G	T
A	0.053	0.079	0.127	0.036
C	0.037	0.058	0.058	0.041
G	0.035	0.075	0.081	0.026
T	0.024	0.105	0.115	0.050

	A	C	G	T
A	0.087	0.058	0.084	0.061
C	0.067	0.063	0.017	0.063
G	0.053	0.053	0.063	0.042
T	0.051	0.070	0.084	0.084

Рис. 10.5 Частоты динуклеотидов для набора CG-островов (слева) и не-CG-островов (справа) в геноме человека, рассчитанные для одной цепи X-хромосомы. Частоты CG показаны красным цветом

Наивным подходом к поиску CG-островов в геноме будет сдвигать окно по геному, объявляя окна с более высокими частотами CG потенциальными CG-островками. Недостатки этого метода аналогичны недостаткам использования скользящего окна для определения монеты, которую мошенник, скорее всего, использовал в тот или иной момент времени. Мы не знаем, какой длины должно быть окно, и перекрывающиеся окна могут одновременно классифицировать одну и ту же геномную позицию как принадлежащую CG-острову и как не принадлежащую CG-острову.

Скрытые марковские модели

От подбрасывания монеты к скрытой марковской модели

Наша цель – разработать единую концепцию, моделирующую как мошенника, так и поиск CG-островов в геноме. С этой целью мы будем думать о мошеннике не как о человеке, а как о примитивной машине. Мы не знаем, как устроена эта машина, но мы знаем, что она работает, выполняя последовательность шагов; на каждом этапе она находится в одном из двух скрытых состояний, *F* и *B*, и выдает символ «O» или «P».

После каждого шага машина принимает два решения.

- В какое скрытое состояние я перейду дальше?
- Какой символ я выдам в этом состоянии?

Машина отвечает на первый вопрос, выбирая случайным образом одно из состояний *F* и *B* с вероятностью 0,9 остаться в своем текущем состоянии и с вероятностью 0,1 изменить состояние. Машина отвечает на второй вопрос, выбирая между символами «O» и «P» с вероятностями, зависящими от состояния, в котором она находится. В нашем примере с подбрасыванием монеты вероятности для состояния *F* (0,5 и 0,5) отличаются от вероятностей для состояния *B* (0,75 и 0,25). Наша цель – вывести наиболее вероятную последовательность состояний машины, анализируя последовательность выдаваемых ею символов.

Мы только что превратили дилера в абстрактную машину, называемую **скрытой марковской моделью (HMM)**. Единственная разница между нашей специализированной «машиной для подбрасывания монет» и общей кон-

цепцией HMM заключается в том, что последняя может иметь произвольное количество состояний и произвольное распределение вероятностей, определяющее, в какое состояние перейти и какие символы выдать. В общем, HMM (∑, *States*, *Transition*, *Emission*) определяется набором из четырех объектов:

- алфавита Σ выдаваемых символов;
- набора *States* **скрытых состояний**;
- матрицы *Transition* = ($transition_{l,k}$) **вероятностей перехода** размерностью |*States*|×|*States*|, где $transition_{l,k}$ представляет собой вероятность перехода из состояния *l* в состояние *k*;
- матрицы *Emission* = ($emission_k(b)$) **вероятностей выброса** размерностью |*States*|×|Σ|, где $emission_k(b)$ представляет вероятность выдачи символа *b* из алфавита Σ, когда HMM находится в состоянии *k*.

Для каждого состояния *l*

$$\sum_{\text{all states } k} transition_{l,k} = 1,$$

а также

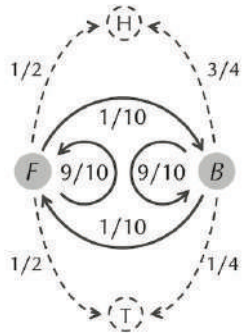
$$\sum_{\text{all symbols } b \text{ from } \Sigma} emission_l(b) = 1.$$

Что такое Σ, *States*, *Transition* и *Emission* для HMM, моделирующей нечестного дилера?

Диаграмма HMM

Как показано на рисунке ниже, HMM можно визуализировать с помощью **диаграммы HMM**, графа, в котором каждое состояние представлено неподвижным узлом. Сплошные направленные ребра соединяют каждую пару узлов, а также каждый узел сам с собой. Каждое такое ребро помечено вероятностью перехода из одного состояния в другое (или пребывания в том же состоянии). Кроме того, на диаграмме HMM есть пунктирные узлы, представляющие каждый возможный символ из алфавита Σ, и пунктирные ребра, соединяющие каждое состояние с каждым пунктирным узлом. Каждое такое ребро помечено вероятностью того, что HMM выдаст этот символ, находясь в заданном состоянии.

Скрытый путь π = $\pi_1 \ldots \pi_n$ в CMM – это последовательность состояний, через которые проходит CMM; такой путь соответствует пути сплошных ребер на диаграмме HMM. На рис. 10.7 представлен пример, в котором мошенник HMM производит последовательность бросков *x* = «РОРОООРОРРО» со скрытым путем π = FFFBBBBBFFFF, т. е. для первых трех бросков и последних трех бросков используется честная монета, а мошенническая монета используется для пяти промежуточных бросков.



Emission

	O	P
F	1/2	1/2
B	3/4	1/4

Transition

	F	B
F	9/10	1/10
B	1/10	9/10

Рис. 10.6 Матрицы вероятности перехода и эмиссии для HMM нечестного дилера, описываемые диаграммой HMM, показанной в центре. Эта HMM имеет два состояния (серые узлы), *F* и *B*. В каждом состоянии HMM может выдать один из двух символов (штриховые узлы) – орел («O») или решку («P») – с вероятностью, показанной вдоль пунктирных ребер. Вероятности перехода показаны сплошными ребрами; мошенник HMM переходит между состояниями *F* и *B* с вероятностью 1/10 и остается в том же состоянии с вероятностью 9/10

i	1	2	3	4	5	6	7	8	9	10	11
x	P	O	P	O	O	O	P	O	P	P	O
π	*F*	*F*	*F*	*B*	*B*	*B*	*B*	*B*	*F*	*F*	*F*
$Pr(\pi_i \to \pi_{i+1})$	$\frac{1}{2}$	$\frac{9}{10}$	$\frac{9}{10}$	$\frac{1}{10}$	$\frac{9}{10}$	$\frac{9}{10}$	$\frac{9}{10}$	$\frac{9}{10}$	$\frac{1}{10}$	$\frac{9}{10}$	$\frac{9}{10}$
$Pr(x_i \mid \pi_i)$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{1}{4}$	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Рис. 10.7 Последовательность выдаваемых символов *x* вместе со скрытым путем π для мошеннического дилера HMM. $Pr(\pi_i \to \pi_{i+1})$ обозначает вероятность $transition_{\pi_i\pi_{i+1}}$ перехода из состояния π_i в π_{i+1}. $Pr(\pi_0 \to \pi_1)$ принимается равным 1/2, чтобы соответствовать предположению, что вначале дилер с одинаковой вероятностью будет использовать честную или мошенническую монету. $Pr(x_i|\pi_i)$ обозначает вероятность того, что дилер выдал символ x_i из состояния π_i, что равно $emission_{\pi_i}(x_i)$

Переформулировка задачи казино

Теперь мы можем исправить неправильно сформулированную задачу казино как поиск наиболее вероятного скрытого пути π для строки *x* символов, выдаваемой HMM. Чтобы решить эту задачу, мы сначала рассмотрим более простую задачу вычисления вероятности $Pr(x, \pi)$ того, что HMM следует по скрытому пути $\pi = \pi_1 \ldots \pi_n$ и выдает строку $x = x_1 \ldots x_n$. Обратите внимание, что

$$\sum_{\text{all strings of emitted symbols } x} \; \sum_{\text{all hidden paths } \pi} Pr(x, \pi) = 1.$$

ОСТАНОВИТЕСЬ и задумайтесь. Чему равно $Pr(x, \pi)$ для x и π на рис. 10.7?

Каждая выдаваемая строка x имеет вероятность $Pr(x)$, что не зависит от скрытого пути, выбранного HMM:

$$Pr(x) = \sum_{\text{all hidden paths } \pi} Pr(x, \pi).$$

Каждый скрытый путь π имеет вероятность $Pr(\pi)$, которая не зависит от строки, выдаваемой HMM:

$$Pr(\pi) = \sum_{\text{all strings of emitted symbols } x} Pr(x, \pi).$$

Событие «HMM следует по скрытому пути π и выдает x» можно рассматривать как комбинацию двух последовательных событий:

- HMM следует по пути π. Вероятность этого события равна $Pr(\pi)$;
- HMM выдает x, учитывая, что HMM следует по пути π. Мы называем вероятность этого события **условной вероятностью** x при заданном π, обозначаемом $Pr(x|\pi)$.

Оба эти события должны произойти, чтобы HMM следовала по пути π и выдавала строку x, что подразумевает, что

$$Pr(x, \pi) = Pr(x|\pi) \cdot Pr(\pi).$$

Чтобы узнать больше об этой формуле, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Условная вероятность**.

Чтобы вычислить $Pr(x, \pi)$, сначала вычислим $Pr(\pi)$. Как показано на рис. 10.7, мы пишем $Pr(\pi_i \rightarrow \pi_{i+1})$ для обозначения вероятности перехода HMM из состояния π_i в π_{i+1}. Для простоты мы предполагаем, что в начале дилер с одинаковой вероятностью будет использовать честную или мошенническую монету, предположение, которое моделируется установкой $Pr(\pi_0 \rightarrow \pi_1) = 1/2$ на рисунке ниже, где π_0 – это «молчащее» **начальное состояние**, которое не выдает никаких символов. Таким образом, вероятность π равна произведению вероятностей его перехода (зеленые элементы на таблице ниже),

$$Pr(\pi) = \prod_{i=1}^{n} Pr(\pi_{i-1} \rightarrow \pi_i) = \prod_{i=1}^{n} transition_{\pi_{i-1}, \pi_i}.$$

Задача определения вероятности скрытого пути: *вычислить вероятность скрытого пути.*

Input: скрытый путь π в HMM (Σ, *States*, *Transition*, *Emission*).
Output: вероятность этого пути, $Pr(\pi)$.

Обратите внимание, что мы уже вычислили $Pr(x|\pi)$ для нечестного дилера HMM, когда скрытый путь дилера состоял только из *B* или *F*, которые мы записали как $Pr(x|B)$ и $Pr(x|F)$ соответственно. Чтобы вычислить $Pr(x|\pi)$ для общей HMM, мы будем писать $Pr(x_i|\pi_i)$, чтобы обозначить вероятность $\text{emission}_{\pi_i}(x_i)$, которая была выдана символом x_i, при условии, что HMM находилась в состоянии π_i (рис. 10.7). В результате для заданного пути π CMM выдает строку *x* с вероятностью, равной произведению вероятностей эмиссий на этом пути,

$$\begin{aligned} Pr(x|\pi) &= \prod_{i=1}^{n} Pr(x_i|\pi_i) \\ &= \prod_{i=1}^{n} emission_{\pi_i}(x_i). \end{aligned}$$

Задача определения вероятности результата при наличии скрытого пути: *вычислить вероятность того, что HMM выдаст строку, имея ее скрытый путь.*

Input: строка $x = x_1 \ldots x_n$, выдаваемая HMM (Σ, *States*, *Transition*, *Emission*) и скрытый путь $\pi = \pi_1 \ldots \pi_n$.
Output: условная вероятность $Pr(x|\pi)$ того, что *x* будет выдано, при условии, что HMM следует скрытому пути π.

Возвращаясь к нашей формуле для $Pr(x, \pi)$, вероятность того, что HMM следует по пути π и выдает строку *x*, может быть записана как произведение вероятностей эмиссии и перехода:

$$\begin{aligned} Pr(x, \pi) &= Pr(x|\pi) \cdot Pr(\pi) \\ &= \prod_{i=1}^{n} Pr(x_i|\pi_i) \cdot Pr(\pi_{i-1} \rightarrow \pi_i) \\ &= \prod_{i=1}^{n} emission_{\pi_i}(x_i) \cdot transition_{\pi_{i-1}, \pi_i}. \end{aligned}$$

ОСТАНОВИТЕСЬ и задумайтесь. Теперь, когда вы узнали о HMM, попробуйте разработать HMM, которая будет моделировать поиск CG-островов в геноме. С какими трудностями вы столкнулись?

Задача декодирования

Граф Витерби

Как мы заявили в предыдущем разделе, как в HMM дилера-мошенника, так и в HMM острова CG мы ищем наиболее вероятный скрытый путь π для HMM, который выдает строку x. Другими словами, мы хотели бы максимизировать $Pr(x, \pi)$ среди всех возможных скрытых путей π.

Задача декодирования: *найти оптимальный скрытый путь в HMM по заданной строке выдаваемых им символов.*

Input: строка $x = x_1 \ldots x_n$, выдаваемый HMM (Σ, *States*, *Transition*, *Emission*).

Output: путь π, максимизирующий вероятность $Pr(x, \pi)$ по всем возможным путям через эту HMM.

В 1967 году Эндрю Витерби использовал вдохновленный HMM аналог манхэттенской сетки для решения проблемы декодирования. Для HMM, выдающей строку из n символов $x = x_1 \ldots x_n$, узлы графа Витерби HMM делятся на $|States|$ строк и n столбцов (рис. 10.8). То есть узел (k, i) представляет состояние k и i-й выдаваемый символ. Каждый узел соединен со всеми узлами в столбце справа от него; ребро, соединяющее $(l, i-1)$ с (k, i), соответствует переходу из состояния l в состояние k (с вероятностью $transition_{l,k}$) и последующей эмиссии символа x_i (с вероятностью эмиссии $k(x_i)$). В результате каждый путь, соединяющий узел в первом столбце графа Витерби с узлом в последнем столбце, соответствует скрытому пути $\pi = \pi_1 \ldots \pi_n$.

Мы присваиваем вес

$$Weight_i(l,k) = transition_{\pi_{i-1},\pi_i} \cdot emission_{\pi_i}(x_i)$$

ребру, соединяющему $(l, i-1)$ с (k, i) в графе Витерби. Кроме того, мы определяем **вес произведения** пути в графе Витерби как произведение весов его ребер. Для пути из крайнего левого столбца в крайний правый столбец графа Витерби, соответствующего скрытому пути π, этот вес произведения равен произведению $n-1$ слагаемых,

$$\prod_{i=2}^{n} transition_{\pi_{i-1},\pi_i} \cdot emission_{\pi_i}(x_i) = \prod_{i=2}^{n} Weight_i(l,k).$$

ОСТАНОВИТЕСЬ и задумайтесь. Чем это выражение отличается от формулы для $Pr(x, \pi)$, которую мы вывели в предыдущем разделе?

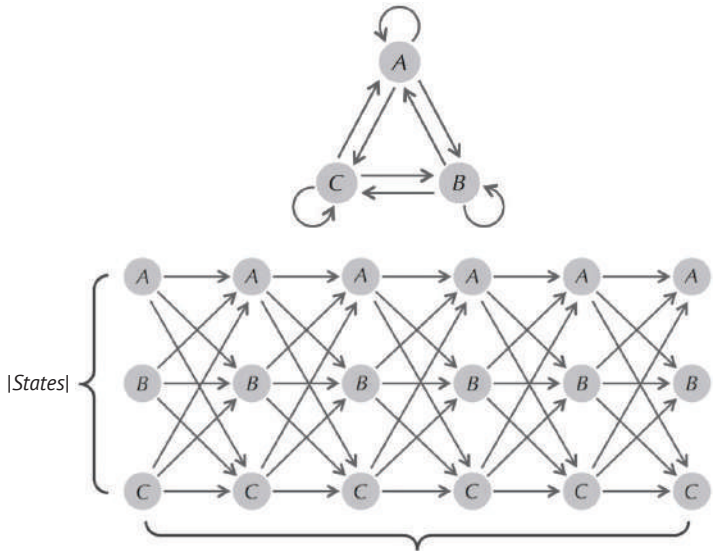


Рис. 10.8 (Вверху) Диаграмма HMM с тремя состояниями (вероятности эмиссии/перехода, а также узлы, соответствующие выдаваемым символам, опущены). (Внизу) Дана строка из n символов $x = x_1 \ldots x_n$, выдаваемая HMM; Манхэттен Витерби представляет собой сетку с $|States|$ строк и n столбцов, в которых каждый узел соединен с каждым узлом в столбце справа от него. Вес ребра, соединяющего $(l, i-1)$ с (k, i), равен $Weight(l, k) = transition_{l,k} \cdot emission_k(x_i)$. В отличие от графов выравнивания, с которыми мы сталкивались ранее, где набор допустимых направлений был ограничен южным, восточным и юго-восточным ребрами, в графе Витерби каждый узел в столбце соединен ребром с каждым узлом в столбце справа от него

Единственная разница между выражением

$$\prod_{i=2}^{n} transition_{\pi_{i-1},\pi_i} \cdot emission_{\pi_i}(x_i) = \prod_{i=1}^{n-1} Weight_i(l,k)$$

и выражением, которое мы получили для $Pr(x, \pi)$,

$$\prod_{i=1}^{n} transition_{\pi_{i-1},\pi_i} \cdot emission_{\pi_i}(x_i),$$

есть единственный фактор $transition_{\pi_0,\pi_1} \cdot emission_{\pi_1}(x_1)$, который соответствует переходу из начального состояния π_0 в π_1 и эмиссии первого символа. Чтобы смоделировать начальное состояние, мы добавим исходный узел-источник *source* к графу Витерби, а затем соединим источник с каждым узлом $(k, 1)$ в первом столбце с ребром $Weight_1(source, k) = transition_{\pi_0,k} \cdot emission_k(x_1)$. Мы также предположим, что у HMM есть еще одно молчащее **конечное состояние**, в которое

HMM входит после того, как она закончила генерировать символы. Чтобы смоделировать конечное состояние, мы добавляем узел-сток *sink* в граф Витерби и соединяем каждый узел в последнем столбце с *sink* ребром веса 1 (рис. 10.9).

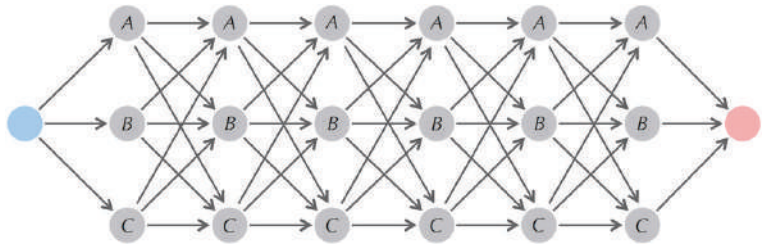


Рис. 10.9 Граф Витерби с дополнительным узлом-источником (синий) и узлом-стоком (красный). Путь наибольшего веса произведения, соединяющий источник со стоком, соответствует оптимальному скрытому пути, решающему Задачу декодирования

Каждый скрытый путь π в HMM теперь соответствует пути от *source* к *sink* в графе Витерби с весом произведения $Pr(x, \pi)$. Таким образом, задача декодирования сводится к поиску пути в графе Витерби с наибольшим весом произведения среди всех путей, соединяющих *source* с *sink*.

Упражнение. Найдите траекторию максимального веса произведения на графе Витерби для нечестного дилера HMM (рис. 10.6), когда x = «HHTT». Выразите свой ответ в виде строки из четырех символов «F» и «B».

Примечание. Можно предположить, что переходы из начального состояния происходят с равной вероятностью.

Алгоритм Витерби

Применим алгоритм динамического программирования для решения проблемы декодирования. Сначала определим $s_{k,i}$ как вес произведения оптимального пути (т. е. пути с максимальным весом произведения) от *source* к узлу (k, i). **Алгоритм Витерби** основан на том факте, что первые $i - 1$ ребер оптимального пути от *source* к (k, i) должны образовывать оптимальный путь от *source* к $(l, i - 1)$ для некоторого (неизвестного) состояния l. Это наблюдение дает следующую рекурсию:

$$\begin{aligned} s_{k,l} &= \max_{\text{all states } l} \{s_{l,i-1} \cdot (\text{weight of edge between nodes } (l, i-1) \text{ and } (k,i))\} \\ &= \max_{\text{all states } l} \{s_{l,i-1} \cdot Weight_i(l,k)\} \\ &= \max_{\text{all states } l} \{s_{l,i-1} \cdot transition_{\pi_{i-1},\pi_i} \cdot emission_{\pi_i}(x_i)\}. \end{aligned}$$

Поскольку *source* связан с каждым узлом в первом столбце графа Витерби,

$$\begin{aligned} s_{k,1} &= s_{source} \cdot (\text{weight of edge between } source \text{ and } (k,1)) \\ &= s_{source} \cdot Weight_0(source, k) \\ &= s_{source} \cdot transition_{source,k} \cdot emission_k(x_1). \end{aligned}$$

Чтобы инициализировать эту рекурсию, мы устанавливаем s_{source} равным 1. Теперь мы можем вычислить максимальный вес произведения по всем путям от *source* к *sink* как

$$s_{sink} = \max_{\text{all states } l} s_{l,n}.$$

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем адаптировать наш алгоритм поиска самого длинного пути в DAG, чтобы найти путь с максимальным весом произведения?

Насколько быстр алгоритм Витерби?

Мы можем интерпретировать задачу декодирования как еще один пример задачи самого длинного пути в задаче DAG из нашей работы по выравниванию последовательностей, поскольку путь π максимизирует вес произведения $\prod_{i=1}^{n} Weight_i(\pi_{i-1}, \pi_i)$, а также максимизирует логарифм этого произведения, который равен $\sum_{i=1}^{n} \log(Weight_i(\pi_{i-1}))$. Таким образом, мы можем заменить веса всех ребер в графе Витерби их логарифмами. Поиск самого длинного пути (т. е. пути, максимизирующего сумму весов ребер) в результирующем графе будет соответствовать пути максимального веса произведения в исходном графе Витерби. По этой причине время выполнения алгоритма Витерби, который вы теперь готовы реализовать, линейно зависит от количества ребер в графе Витерби. Следующее упражнение показывает, что количество этих ребер равно $O(|States|^2 \cdot n)$, где n – количество выданных символов.

Упражнение. Покажите, что число ребер в графе Витерби HMM, выдающем строку длины n, равно $|States|^2 \cdot (n - 1) + 2 \cdot |States|$.

Упражнение. Примените свое решение задачи декодирования, чтобы найти CG-острова в первом миллионе нуклеотидов X-хромосомы человека[1] (данные в формате FASTA). Чтобы

[1] https://github.com/BioinformaticsAlgorithms/BioinformaticsAlgorithms.github.io/blob/main/data/realdatasets/HMM/chrX.txt.

помочь вам спроектировать HMM для этого приложения, вы можете предположить, что переходы от CG-островов к не-CG-островам редки и происходят с вероятностью 0,001 и что переходы от не-CG-островов к CG-островам еще более редки, встречаются с вероятностью 0,0001. Сколько CG-островов вы нашли?

☞ **Загрузить данные 10.1 (X-хромосома человека)**

На практике многие HMM имеют запрещенные переходы между некоторыми состояниями. Для таких переходов мы можем смело удалить соответствующие ребра из диаграммы HMM (рисунок ниже). В результате этой операции получается более разреженный граф Витерби (рисунок ниже), что сокращает время выполнения алгоритма Витерби, поскольку время выполнения алгоритма поиска самого длинного пути в DAG линейно зависит от количества ребер в DAG.

Упражнение. Пусть *Edges* обозначает набор ребер на диаграмме HMM, которые могут иметь некоторые запрещенные переходы. Докажите, что количество ребер в графе Витерби для этой HMM равно $|Edges| \cdot (n - 1) + 2 \cdot |States|$.

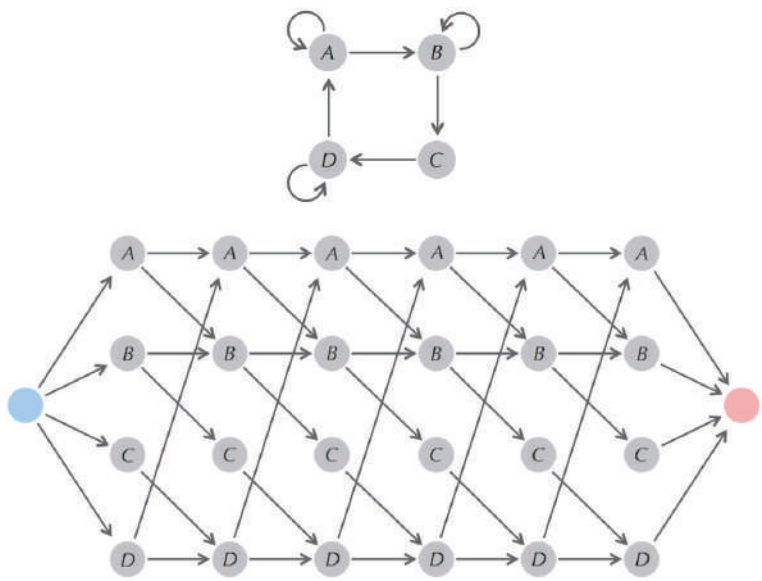


Рис. 10.10 (Вверху) Диаграмма HMM для HMM, которая имеет четыре состояния с некоторыми запрещенными переходами, например из *A* в *D* и из *C* в себя. Ребра, соответствующие запрещенным переходам между состояниями, не включны в диаграмму HMM. (Внизу) Граф Витерби для этой HMM, выдающей строку длины 6

Поиск наиболее вероятного результата HMM

Динамическое программирование позволяет нам ответить на вопросы о HMM, выходящих за пределы наиболее вероятного скрытого пути. Например, мы уже вычислили вероятность $Pr(\pi)$ скрытого пути π. Но как насчет вычисления $Pr(x)$, вероятности того, что HMM выдаст строку x?

ОСТАНОВИТЕСЬ и задумайтесь. Какой исход более вероятен в мошенническом казино: «OOPP» или «OPOP»? Как бы вы нашли наиболее вероятную последовательность из четырех бросков монеты?

Задача вероятности результата: *найти вероятность того, что HMM выдаст заданную строку.*

Input: строка $x = x_1 \ldots x_n$, выдаваемая HMM (Σ, *States*, *Transition*, *Emission*).

Output: вероятность $Pr(x)$ того, что HMM выдает x.

ОСТАНОВИТЕСЬ и задумайтесь. Чтобы решить задачу вероятности исхода, вы можете внести небольшое изменение в повторение Витерби,

$$s_{k,i} = \max_{\text{all states } l}\{s_{l,i-1} \cdot Weight_i(l, k)\}.$$

Что изменилось?

Мы уже заметили, что $Pr(x)$ равно сумме $Pr(x, \pi)$ по всем скрытым путям π. Однако количество путей через граф Витерби экспоненциально зависит от длины выдаваемой строки x, поэтому мы будем использовать динамическое программирование для разработки более быстрого метода вычисления $Pr(x)$.

Обозначим общий вес произведения всех путей от *source* к узлу (k, i) в графе Витерби как $forward_{k,i}$; обратите внимание, что $forward_{sink}$ равен $Pr(x)$. Чтобы вычислить $forward_{k,i}$, мы разделим все пути, соединяющие *source* с (k, i), на подмножества $|States|$, где каждое подмножество содержит те пути, которые проходят через узел $(l, i - 1)$ с весом $forward_{i,i-1}$ до достижения (k, i) между 1 и $|States|$. Следовательно, $forward_{k,i}$ является суммой $|States|$,

$$forward_{k,i} = \sum_{\text{all states } l} forward_{l,i-1} \cdot (\text{weight of edge connecting } (l, i-1) \text{ and } (k,i))$$
$$= \sum_{\text{all states } l} forward_{l,i-1} \cdot Weight_i(l,k).$$

Обратите внимание, что единственная разница между этой рекурсией и рекурсией Витерби

$$s_{k,i} = \max_{\text{all states } l} \{s_{l,i-1} \cdot Weight_i(l,k)\}$$

заключается в том, что максимизация в алгоритме Витерби превратилась в символ суммирования. Теперь мы можем решить задачу вероятности результата, подсчитав $forward_{sink}$, который равен

$$\sum_{\text{all states } k} forward_{k,n}.$$

Теперь, когда мы можем вычислить $Pr(x)$ для выданной строки x, возникает естественный вопрос: найти лучшую такую строку. В примере с нечестным дилером это соответствует нахождению наиболее вероятной последовательности бросков среди всех возможных последовательностей честных и мошеннических монет, которые мог бы использовать дилер.

Задача наиболее вероятного результата: *найдите наиболее вероятную строку, выдаваемую HMM.*

Input: HMM (Σ, *States*, *Transition*, *Emission*) и целое число n.

Output: наиболее вероятная строка $x = x_1 \ldots x_n$, выдаваемая этой HMM, т. е. строка, максимизирующая вероятность $Pr(x)$ того, что HMM будет выдавать x.

Упражнение. Решите задачу наиболее вероятного результата (подсказка: вам может понадобиться построить трехмерную версию Манхэттена Витерби).

Профильные HMM для выравнивания последовательностей

Как HMM связаны с выравниванием последовательностей?

Возможно, вам все еще интересно, какое отношение HMM имеют к нашей первоначальной задаче выравнивания последовательностей с использованием специфичного для столбца счета. Как мы увидим, HMM предлагают элегантное решение этой задачи.

Имея семейство родственных белков, мы можем проверить, принадлежит ли к нему вновь секвенированный, построив попарное выравнивание между вновь секвенированным белком и каждым членом семейства. Если одно из полученных выравниваний превышает некоторый определенный порог, мы можем предположить, что новый белок принадлежит к этому семейству. Однако этот метод может не идентифицировать отдаленно родственные белки, такие как белки gp120, взятые из разных изолятов ВИЧ, поскольку эти белки могут иметь оценки ниже порогового значения. Однако, если последовательность имеет слабое сходство со многими членами семьи, то она, скорее всего, принадлежит семье.

Таким образом, задача состоит в том, чтобы подвергнуть выравниванию новый белок со всеми членами семейства одновременно. Для этого мы должны предположить, что мы уже построили множественное выравнивание семейства белков. К счастью, часто бывает очевидно, что два белка происходят из одного семейства (например, если белки взяты из близкородственных видов). Соответственно, биологи часто начинают с построения выравнивания бесспорно родственных белков, которые обычно легко сопоставить даже с использованием простых методов множественного выравнивания, которые мы обсуждали ранее.

На рис. 10.11 показано выравнивание *Alignment* размерностью 5×10, представляющее гипотетическое семейство белков. Обратите внимание, что шестой и седьмой столбцы этого выравнивания содержат много пробелов и, вероятно, не имеют значимых характеристик семейства. Соответственно, биологи часто игнорируют столбцы, для которых доля пробелов больше или равна **порогу удаления столбца** θ. Удаление колонки приводит к **семени выравнивания** 5×8.

Для данного семени выравнивания *Alignment**, представляющее семейство родственных белков, мы должны построить HMM, которая реалистично моделирует сходства символов в *Alignment**, представленном матрицей профиля *Profile*(*Alignment**), которая добавляет к рис. 10.11 третью картинку, как показано на рис. 10.12. Вместо того, чтобы думать о выравнивании существующего семени выравнивания с заданной строкой *Text* (представляющей новый белок), мы подумаем о вычислении вероятности того, что HMM выдаст *Text*. Если HMM сделана хорошо, то чем больше *Text* будет похож на строки в *Alignment**, тем больше вероятность того, что он будет выдан HMM.

	1	2	3	4	5	6	7	8
Alignment	A	C	D	E	F	A C	A	D	F
	A	F	D	A	–	– –	C	C	F
	A	–	–	E	F	D –	F	D	C
	A	C	A	E	F	– –	A	–	C
	A	D	D	E	F	A A	A	D	F
*Alignment**	A	C	D	E	F		A	D	F
	A	F	D	A	–		C	C	F
	A	–	–	E	F		F	D	C
	A	C	A	E	F		A	–	C
	A	D	D	E	F		A	D	F

Рис. 10.11 Множественное выравнивание *Alignment* 5×10 (вверху) и его семени *Alignment** 5×8 (внизу). Семя выравнивания получается из оригинального выравнивания путем игнорирования плохо сохранившихся столбцов (заштриховано серым цветом); в этом случае мы игнорируем столбцы, для которых доля пробелов больше или равна порогу удаления столбца θ = 0,35. Чтобы лучше проиллюстрировать взаимосвязь между выравниванием и его семенем, мы отделили первые пять столбцов начального выравнивания от его последних трех столбцов и пронумеровали эти столбцы над исходным выравниванием

		1	2	3	4	5	6	7	8
Alignment		A	C	D	E	F	A C	A	D	F
		A	F	D	A	–	– –	C	C	F
		A	–	–	E	F	D –	F	D	C
		A	C	A	E	F	– –	A	–	C
		A	D	D	E	F	A A	A	D	F
*Alignment**		A	C	D	E	F		A	D	F
		A	F	D	A	–		C	C	F
		A	–	–	E	F		F	D	C
		A	C	A	E	F		A	–	C
		A	D	D	E	F		A	D	F
PROFILE(*Alignment**)	A	1	0	1/4	1/5	0	3/5	0	0
	C	0	2/4	0	0	0	1/5	1/4	2/5
	D	0	1/4	3/4	0	0	0	3/4	0
	E	0	0	0	4/5	0	0	0	0
	F	0	1/4	0	0	1	1/5	0	3/5

Рис. 10.12 К рис. 10.11 добавлена матрица профиля множественного выравнивания

Сначала мы создадим простую HMM, которая обрабатывает столбцы *Alignment** как k последовательно связанных состояний, называемых **состояниями совпадения** (рис. 10.12), обозначаемых *Match*(1), … , *Match*(k). Когда HMM переходит в состояние *Match*(i), она выдает символ x_i с вероятностью, равной частоте появления этого символа в i-м столбце *Profile*(*Alignment**). Затем HMM переходит в состояние *Match*(i + 1) с вероятностью перехода, равной 1.

Показатель сходства между *Alignment** и *Text* – это вероятность *Pr*(*Text*), что HMM для *Alignment** выдает *Text*. Этот показатель равен произведению частот в *Profile*(*Alignment**), соответствующих каждому символу *Text*. Например, вероятность того, что HMM на рисунке ниже выдает ADDAFFDF, равна

$$1 \cdot (1/4) \cdot (3/4) \cdot (1/5) \cdot 1 \cdot (1/5) \cdot (3/4) \cdot (3/5) = 0{,}003375.$$

ОСТАНОВИТЕСЬ и задумайтесь. Каковы ограничения HMM (рис. 10.13)?

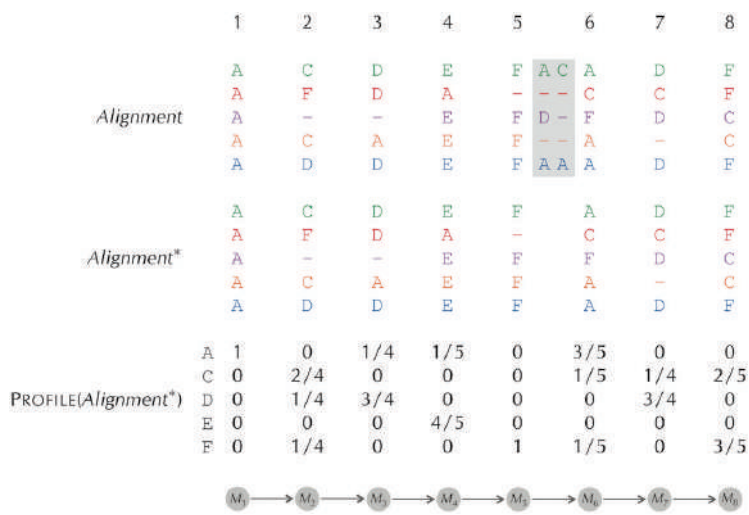


Рис. 10.13 Добавим диаграмму простой HMM. Состояния соответствия *Match*(*i*) сокращенно обозначаются как M_i. У HMM есть только один возможный путь; изначально она находится в состоянии *Match*(1), вероятность перехода из состояния *Match*(*i*) в состояние *Match*(*i* + 1) равна 1 для всех *i*, а все остальные переходы запрещены. Вероятности выбросов равны частотам в профиле, например вероятности выбросов для M_2 равны 0 для A, 2/4 для C, 1/4 для D, 0 для E и 1/4 для F

Предложенная нами HMM действительно считает каждый столбец поразному, и в какой-то степени чем больше *Text* похож на *Alignment**, тем выше его показатель сходства. Однако эта HMM не соответствует духу HMM, потому что имеет только один скрытый путь. Кроме того, она предлагает упрощенное представление о множественном выравнивании, поскольку не учитывает вставки и делеции. Наконец, она может «выровнять» *Text* по отношению к *Alignment** только в том случае, если длина *Text* точно равна количеству столбцов в *Text* (рис. 10.14). Тем не менее мы будем использовать эту ограниченную HMM в качестве основы для более мощной HMM.

	A	C	D	E	F	A	D	F
	A	F	D	A	–	C	C	F
*Alignment**	A	–	–	E	F	F	D	C
	A	C	A	E	F	A	–	C
	A	D	D	E	F	A	D	F
Text	A	D	D	A	F	F	D	F
emission probability	1	1/4	3/4	1/5	1	1/5	3/4	3/5

Рис. 10.14 Выравнивание *Text* = ADDAFFDF относительно исходного выравнивания *Alignment**, представленного в виде простой HMM на рис. 10.13. Эта HMM ограничена, потому что мы не можем выровнять строку длины, отличной от 8. Действительно, нет способа добавить символы пробела в *Text* или добавить символы *Text* «между» столбцами *Alignment**

Создание профильной HMM

Усовершенствованная модель HMM, которую мы предлагаем, называется **профильной HMM**. Имея множественное выравнивание Alignment и пороговое значение удаления столбца θ, используемое для получения исходного выравнивания *Alignment**, мы будем обозначать эту профильную HMM как *HMM*(*Alignment*, θ). Поскольку профильная HMM будет построена из начального выравнивания, мы также будем неофициально называть ее *HMM*(*Alignment**). Имея строку *Text*, которую нужно выровнять по существующему начальному выравниванию, наша цель – найти оптимальный скрытый путь в профильной HMM, решив задачу декодирования для этой HMM и выданной строки *Text*.

Как и в нашем первом пприближении HMM, профильная HMM по-прежнему будет проходить свои состояния в порядке, согласующемся с проходом столбцов *Alignment**, слева направо. Однако для выравнивания строк *Text* различной длины нам потребуется больше состояний в дополнение к k состояниям совпадения.

Во-первых, мы добавляем $k+1$ **состояний вставки**, обозначаемых *Insertion*(0), … , *Insertion*(k) (рис. 10.15). Ввод *Insertion*(i) позволяет профильную HMM выдать дополнительный символ после посещения i-го столбца *Profile*(*Alignment**) и перед входом в (i + 1)-й столбец. Таким образом, мы свяжем *Match*(i) с *Insertion*(i) и *Insertion*(i) с Match(i + 1). Кроме того, чтобы разрешить вставку нескольких символов между столбцами *Profile*(*Alignment**), мы соединим *Insertion*(i) с самой собой.

ОСТАНОВИТЕСЬ и задумайтесь. Можем ли мы использовать приведенную ниже HMM для выравнивания строки *Text* длиной менее 8?

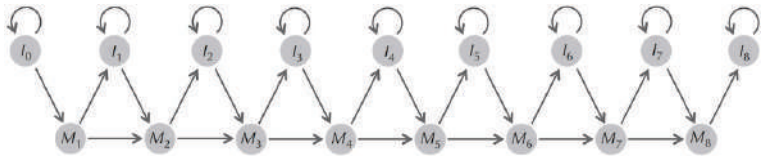


Рис. 10.15 Диаграмма HMM для семян выравнивания, представленного выше, с состояниями совпадения и вставки, сокращенно обозначаемыми как *M* и *I* соответственно. Состояния I_0 и I_8 моделируют вставки символов, происходящие до начала и конца *Alignment** соответственно

После моделирования вставок новых символов в *Profile*(*Alignment**) мы должны также смоделировать «делеции», позволяющие профильной HMM пропускать столбцы *Profile*(*Alignment**). Одним из способов моделирования этих удалений является добавление ребер, соединяющих каждое состояние в профильной HMM с каждым состоянием справа от нее (рис. 10.16).

ОСТАНОВИТЕСЬ и задумайтесь. Вспомните, что время работы алгоритма Витерби пропорционально количеству ребер (с ненулевыми вероятностями перехода) на диаграмме HMM. Сколько ребер будет иметь диаграмма на рисунке ниже? Как мы можем уменьшить количество ребер в диаграмме HMM?

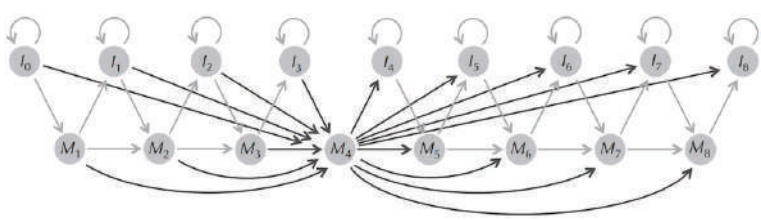


Рис. 10.16 Добавляя ребра, соединяющие каждое состояние в профильной HMM с каждым состоянием справа от него, мы можем пропустить столбцы выравнивания *Alignment* при сравнении его с *Text*. На приведенной выше диаграмме HMM выделены все ребра, ведущие в *Match*(4) и выходящие из него

Вместо того чтобы пропускать состояния, как мы сделали выше, можно уменьшить количество ребер в диаграмме HMM, введя k молчащих **состояний делеции** *Deletion*(1), ..., *Deletion*(k) (рисунок ниже). Например, вместо того, чтобы переходить от *Match*(i – 1) к *Match*(i + 1), мы можем сделать переход *Match*(i – 1) → *Deletion*(i) → *Match*(i + 1). Ввод *Deletion*(i) позволяет HMM пропустить столбец выравнивания без эмиссии символа.

ОСТАНОВИТЕСЬ и задумайтесь. Является ли HMM на рисунке ниже адекватной или мы что-то забыли добавить?

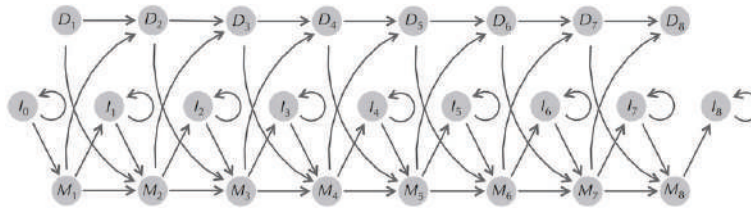


Рис. 10.17 Добавление состояний молчащих делеций (сокращенно D_i) на диаграмму профильной HMM

Теперь мы можем переходить назад и вперед между состояниями совпадения и состояниями вставки, а также между состояниями совпадения и состояниями делеции и обратно, но мы не можем переходить между состояниями вставки и состояниями делеции. Следовательно, диаграмма профильной HMM должна включать ребра, соединяющие *Insertion*(*i*) с *Deletion*(*i* + 1) и соединяющие *Deletion*(*i*) с *Insertion*(*i*) для каждого *i*. В результате профильная HMM может перейти из любого состояния совпадения/вставки в любое другое состояние совпадения/вставки справа от него, отклоняясь от промежуточных состояний делеции. Мы получаем полную профильную HMM-диаграмму, показанную на рисунке ниже, после соединения начального состояния (*S*) с первыми состояниями совпадения/вставки/делеции и соединения конечных состояний совпадения/вставки/делеции с конечным состоянием (*E*).

ОСТАНОВИТЕСЬ и задумайтесь. Рассмотрите следующие вопросы.

- Сколько ребер имеет диаграмма HMM на рисунке ниже? Как это соотносится с диаграммой HMM, представленной ранее, в которой у нас были ребра, перескакивающие через состояния?
- Как выглядит граф Витерби профильной HMM на рисунке ниже? Сколько у него узлов и ребер?

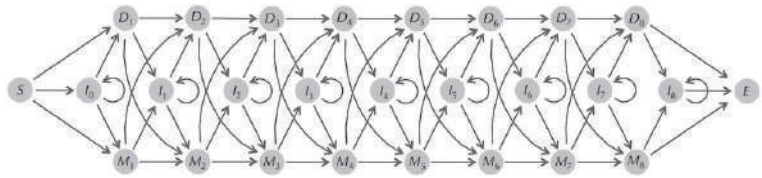


Рис. 10.18 Добавление переходов из состояний вставки в состояния делеции и наоборот завершает диаграмму профильной HMM для матрицы профиля, представленную в этом разделе. Молчащие начальное и конечное состояния показаны буквами *S* и *E* соответственно

Вероятности перехода и эмиссии профильной HMM

На рис. 10.19 мы возвращаемся к множественному выравниванию *Alignment* из предыдущего раздела и представляем каждый из пяти цветных рядов этого выравнивания в виде пути на диаграмме *HMM*(*Alignment* *). Символы в ис-

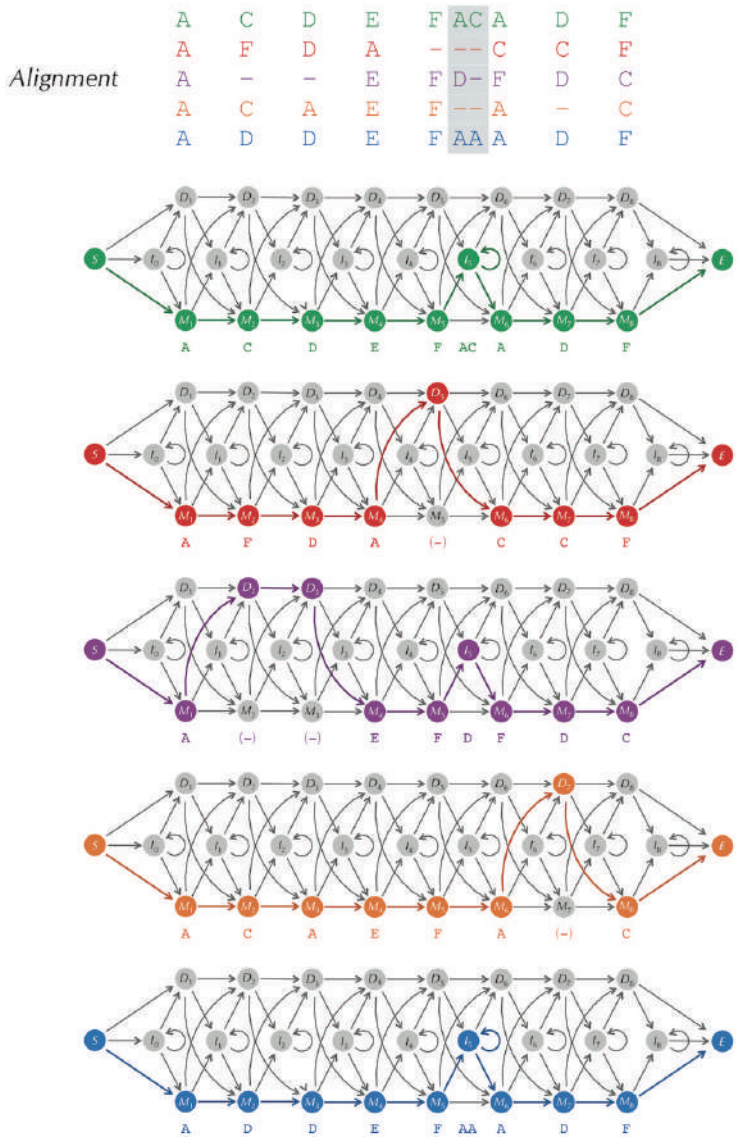


Рис. 10.19 Пять путей через профильную HMM, соответствующих пяти рядам в выравнивании из предыдущего раздела. Символы пробела под диаграммой HMM соответствуют состояниям делеции и показаны в круглых скобках, чтобы указать, что они не эмитируются HMM

ходном выравнивании *Alignment** (незаштрихованные столбцы) соответствуют либо состоянию совпадения (символы без пробелов), либо состоянию делеции (символы пробелов). Что касается символов, отсутствующих в исходном выравнивании (заштрихованные столбцы), символы пробела игнорируются, а символы, не являющиеся пробелами, выводятся из состояний вставки.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы назначили вероятности перехода и эмиссии для профильной HMM для выравнивания на рис. 10.19?

Чтобы назначить вероятность перехода $transition_{l,k}$, мы просто берем частоту переходов из состояния l в состояние k, сделанных по этим окрашенным путям по отношению ко всем путям, которые посетили состояние l. Например, на рис. 10.19 четыре цветных пути посещают *Match*(5). Затем три из этих путей переходят к *Insertion*(5), а один переходит к *Match*(6). Таким образом, мы устанавливаем следующие вероятности перехода при выходе из *Match*(5):

$$transition_{Match(5),\,Insertion(5)} = 3/4$$
$$transition_{Match(5),\,Match(6)} = 1/4$$
$$transition_{Match(5),\,Deletion(6)} = 0.$$

Аналогично можно определить вероятности перехода из начального состояния. Для множественного выравнивания из рис. 10.19 мы вводим *Match*(1) с вероятностью 1; для общей профильной HMM единственными другими состояниями, которые мы могли бы ввести из начального состояния, являются *Insertion*(0) и *Deletion*(1). Полная матрица вероятностей передачи показана на рис. 10.20.

ОСТАНОВИТЕСЬ и задумайтесь. Из-за небольшого количества строк в выравнивании из рис. 10.19 вверху многие вероятности перехода в серых ячейках на рис. 10.20 равны нулю. Каковы возможные негативные последствия этих нулей и как бы вы справились с этими последствиями?

Чтобы присвоить вероятность эмиссии $emission_k(b)$, мы делим количество раз, когда символ b был выдан из состояния k, на общее количество символов, выданных из состояния k. Например, в предоставленной профильной HMM есть три вхождения A, одно вхождение C и одно вхождение D, выдаваемые из состояния *Insertion*(5). (Примечание: эти символы встречаются в заштрихованных столбцах выравнивания, где доля символов пробела превышает порог удаления столбца θ.) Кроме того, есть два вхождения C, одно вхождение D и одно вхождение F, выдаваемые функцией *Match*(2). Таким образом, мы можем вывести следующие вероятности эмиссии для этих двух состояний:

$emission_{Insertion(5)}(A) = 3/5$ $emission_{Match(2)}(A) = 0$
$emission_{Insertion(5)}(C) = 1/5$ $emission_{Match(2)}(C) = 2/4$
$emission_{Insertion(5)}(D) = 1/5$ $emission_{Match(2)}(D) = 1/4$
$emission_{Insertion(5)}(E) = 0$ $emission_{Match(2)}(E) = 0$
$emission_{Insertion(5)}(F) = 0$ $emission_{Match(2)}(F) = ¼.$

	S	I_0	M_1	D_1	I_1	M_2	D_2	I_2	M_3	D_3	I_3	M_4	D_4	I_4	M_5	D_5	I_5	M_6	D_6	I_6	M_7	D_7	I_7	M_8	D_8	I_8	E
S			1																								
I_0																											
M_1						.8	.2																				
D_1																											
I_1																											
M_2									1																		
D_2										1																	
I_2																											
M_3												1															
D_3												1															
I_3																											
M_4															.8	.2											
D_4																											
I_4																											
M_5																	.75	.25									
D_5																		1									
I_5																	.4	.6									
M_6																					.8	.2					
D_6																											
I_6																											
M_7																								1			
D_7																								1			
I_7																											
M_8																											1
D_8																											
I_8																											
E																											

Рис. 10.20 Матрица 27×27 вероятностей перехода для *HMM*(*Alignment*, 0,35), где выравнивание – это множественное выравнивание, показанное выше, а 0,35 – порог делеции столбца. Все значения в пустых ячейках равны нулю. Каждая строка и столбец этой матрицы соответствует одному из 27 узлов на диаграмме HMM для этого множественного выравнивания. Ячейки, заштрихованные серым цветом, соответствуют ребрам на диаграмме HMM; незаштрихованные клетки соответствуют запрещенным переходам

Упражнение. Постройте матрицу вероятности выбросов размерности 27×20 для HMM(*Alignment*, 0,35), полученную из множественного выравнивания *Alignment*, которое мы использовали в качестве примера, воспроизведенного на рис. 10.19 вверху.

Упражнение. Построить профильную HMM для последовательностей HIV (рис. 10.21) с θ = 0,35.

Рис. 10.21 Множественное выравнивание последовательностей HIV

Классификация белков с помощью профильных HMM

Выравнивание белков по профильной HMM

Имея семейство белков, представленное *Alignment*, мы можем теперь вернуться к задаче принятия решения о том, принадлежит ли к семейству вновь секвенированный белок, представленный *Text*. Сначала мы формируем *HMM*(*Alignment*, θ) для некоторого параметра θ. Как показано на рис. 10.22, скрытый путь через *HMM*(*Alignment*, θ) соответствует последовательности состояний совпадения, вставки и делеции для выравнивания *Text* по *Alignment*.

Чтобы найти «лучшее» выравнивание *Text* по *Alignment*, нам просто нужно применить алгоритм Витерби, чтобы найти оптимальный скрытый путь в *HMM*(*Alignment*, θ). Если вес произведения этого оптимального скрытого пути превышает заданный порог, то можно заключить, что *Text* принадлежит к семейству белков, и в этом случае мы дополняем существующее начальное выравнивание дополнительной строкой, соответствующей *Text*. Таким образом, мы можем привлекать все больше и больше отдаленных членов семейства к начальному выравниванию, добавляя эти новые белки к растущему множественному выравниванию и, таким образом, делая результирующщю профильную HMM все более и более подходящей для анализа интересующего семейства белков.

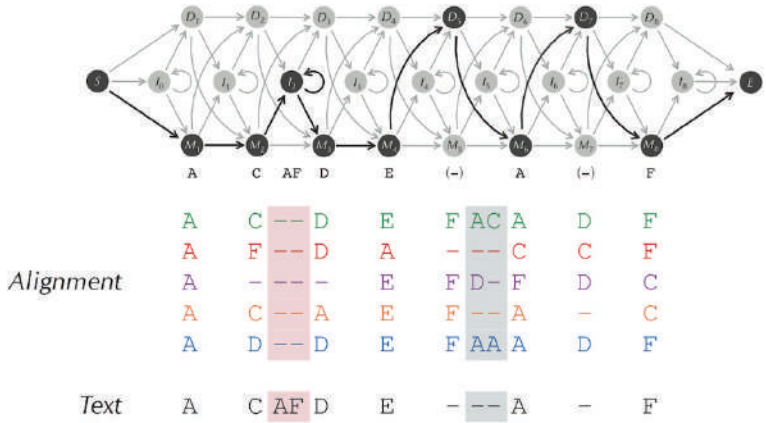


Рис. 10.22 (Вверху) Путь через *HMM*(*Alignment*, 0,35) для множественного выравнивания из предыдущего раздела и сгенерированной строки *Text* = ACAFDEAF. (Внизу) Эмитируемые символы соответствуют выравниванию *Text* по *Alignment*. В частности, первые два символа выдаются из двух состояний совпадения и принадлежат первым двум позициям выравнивания. Следующие два символа эмитируются из состояния вставки и принадлежат отдельным столбцам (отмечены розовым цветом). Символы пробела в седьмом и одиннадцатом столбцах выше соответствуют состояниям делеции; эти символы HMM не выдаются. Символы пробела в серых столбцах не соответствуют никаким состояниям и пропускаются. Незаштрихованные столбцы образуют расширенное выравнивание семян 6×8 для сравнения с недавно секвенированными белками

ОСТАНОВИТЕСЬ и задумайтесь. Если вес произведения для нового белка превышает пороговое значение более чем для одного семейства белков, как бы вы классифицировали этот белок?

Профильные HMM, наконец, помогли нам достичь нашей первоначальной цели – посчитать разные столбцы множественного выравнивания по-разному, в зависимости от частоты символов в каждом столбце. Например, предположим, что седьмой столбец *Alignment** содержит больше вхождений *A*, чем *C*, а девятый столбец *Alignment** содержит больше вхождений *C*, чем *A*. Скрытый путь, проходящий через *Match*(7), получит большее вознаграждение за эмиссию *A*, чем за эмиссию *C*, в то время как скрытый путь, проходящий через *Match*(9), будет вознагражден за эмиссию *C* больше, чем за *A*.

Возвращение псевдосчетов

Возвращаясь к рис. 10.20 из предыдущего раздела, показывающему таблицу вероятностей переходов, видим, что большинство вероятностей переходов

в серых ячейках этого рисунка равны нулю. (То же самое относится и к вероятностям эмиссии.)

Эти нули могут вызвать проблемы; например, путь на рис. 10.22 от выравнивания строки по матрице профиля, воспроизведенный ниже, кажется совершенно разумным для *Text* = ACAFDEAF, и все же $Pr(x, \pi)$ равен нулю, поскольку вероятность перехода от *Match*(2) к *Insertion*(2) для этой профильной HMM равен нулю.

Как и при поиске мотивов, мы будем вводить псевдосчеты, добавляя малое значение σ к элементам в матрице перехода, которые соответствуют ребрам диаграммы HMM (т. е. только серые элементы таблицы из рис. 10.20). Обратите внимание, что белые ячейки в этой таблице, соответствующие запрещенным переходам, не подвержены влиянию псевдосчетов. Затем полученную матрицу необходимо нормализовать, чтобы сумма элементов в каждой строке равнялась 1.

Упражнение. Вычислите нормализованную матрицу для приведенной на рис. 10.20 матрицы после добавления псевдосчета $\sigma = 0{,}01$.

Мы также добавим псевдосчеты в матрицу вероятностей выбросов и нормализуем полученную матрицу. Мы называем профильную HMM, определяемую результирующими нормированными матрицами вероятностей перехода и эмиссии, *HMM*(*Alignment*, θ, σ).

Задача построения профильной HMM с псевдосчетами:
построить профильную HMM с псевдосчетами из множественного выравнивания.

Input: множественное выравнивание *Alignment*, пороговое значение θ и значение псевдосчета σ.

Output: *HMM*(*Alignment*, θ) в виде матриц перехода и эмиссии.

ОСТАНОВИТЕСЬ и задумайтесь. Поскольку диаграмма HMM на рис. 10.22 имеет 25 узлов, не считая начального и конечного состояний, граф Витерби для строки, эмитируемой на этом рисунке, имеет 25 строк. Сколько столбцов имеет этот граф Витерби?

Теперь мы готовы сделать множественное выравнивание строки *Text*, построив для нее граф Витерби (рис. 10.23) и решив задачу декодирования, чтобы найти наиболее вероятный скрытый путь.

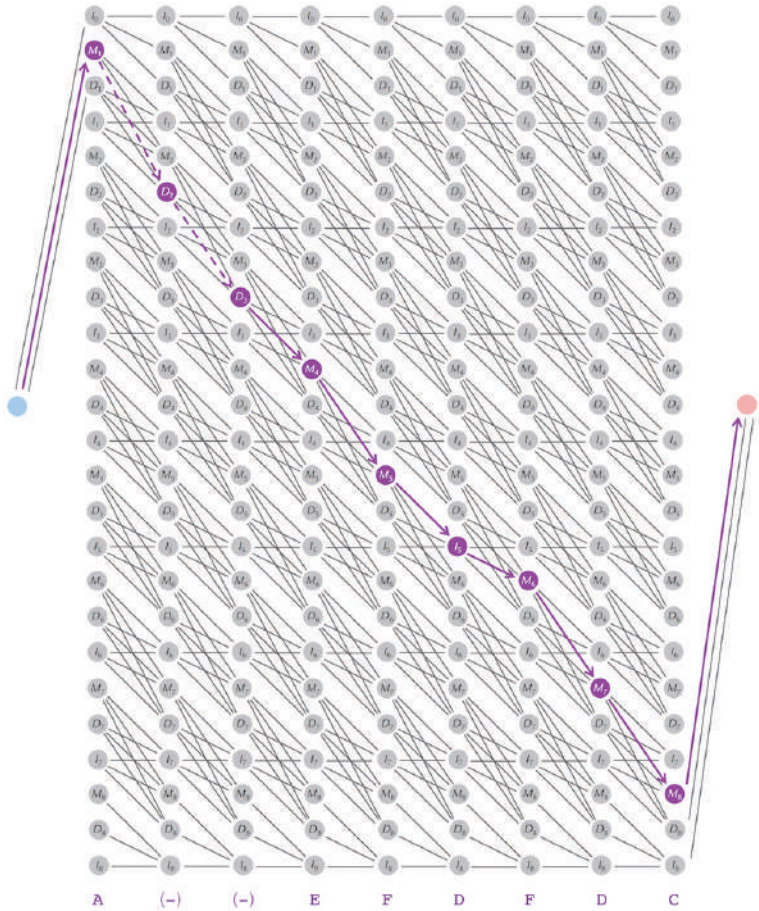


Рис. 10.23 Граф Витерби для *HMM*(*Alignment*, θ) и путь на этом графе (показан фиолетовым цветом), соответствующий скрытому пути для выдаваемой строки **AEFDFDC** из выравнивания, с которым мы работали. Ребра между столбцами соответствуют разрешенным переходам на диаграмме HMM и имеют подразумеваемую правую ориентацию. Ребра, входящие в узлы, соответствующие состояниям делеции, отмечены пунктиром. Эмитируемые символы отображаются под каждым столбцом

ОСТАНОВИТЕСЬ и задумайтесь. Найдите пути через граф Витерби, соответствующие четырем нижним скрытым путям на рис. 10.19. Что получилось?

Проблема с молчащими состояниями

Если вы дошли до этого момента без каких-либо вопросов о рис. 10.23, то мы успешно скрыли от вас, что решение задачи декодирования для HMM с молчащими состояниями не так просто, как может показаться: предложенный нами граф не является графом Витерби! Чтобы понять, почему нет, рассмотрим путь на рис. 10.24, который выдает ту же строку, но проходит на одно состояние молчащей делеции меньше, тем самым уменьшая количество столбцов на один. Но граф Витерби не может меняться в зависимости от скрытого пути π, так как мы ничего заранее не знаем о скрытом пути! Вместо этого количество столб-

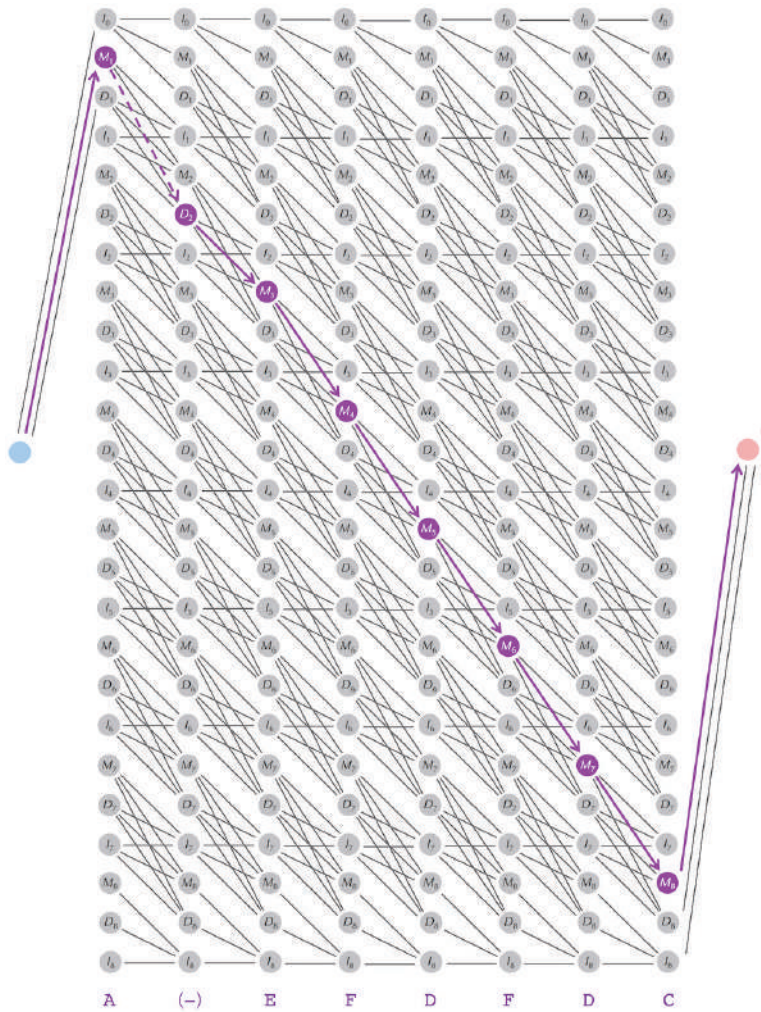


Рис. 10.24 Другой путь через «другой граф Витерби», выдающий ту же строку **AEFDFDC**

цов в графе Витерби должно равняться длине *эмитируемой строки* – условие, которое нарушается на двух предложенных нами рисунках.

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем изменить понятие графа Витерби для HMM с молчащими состояниями?

В более общем смысле алгоритм Витерби не допускает молчащих состояний, кроме начального и конечного. Другими словами, этот алгоритм предполагает, что узел (*k, i*) в графе Витерби описывает событие «HMM выдала символ x_i, когда она была в состоянии *k*». Однако если *k* – молчащее состояние, то роль узла (*k, i*) в графе Витерби определена плохо, так как неясно, как определить вес ребер, входящих в этот узел.

К счастью, мы можем решить эту задачу в случае профильных HMM, определив граф Витерби с помощью |*States*| строк и |*Text*| столбцов (рис. 10.25). Каждый раз, когда HMM переходит в состояние делеции, вместо того, чтобы переходить к следующему столбцу графа Витерби (как на двух представленных ранее неудачных рисунках), мы будем двигаться в пределах одного и того же столбца. Когда HMM переходит в состояние совпадения или вставки, мы перейдем к следующему столбцу. В результате каждый столбец графа Витерби соответствует одному выдаваемому символу, даже если путь может проходить более чем через одно состояние в данном столбце.

Упражнение. Покажите, что вертикальному ребру, соединяющему (*i, l*) с (*i, k*), где *k* – состояние делеции, следует присвоить вес, равный $transition_{l,k}$.

ОСТАНОВИТЕСЬ и задумайтесь. Еще раз взгляните на граф на рис. 10.25. Остались ли с ним какие-нибудь проблемы?

На графе, представленном на рис. 10.25 все еще есть небольшой недостаток. Если HMM переходит из начального состояния в *Deletion*(1), то она будет перемещаться по первому столбцу без эмиссии символа. Поэтому мы преобразуем исходное состояние в столбец молчащих состояний, содержащий исходное состояние и все состояния делеции (рис. 10.26). Таким образом, если HMM входит в состояние *Deletion*(1) из начального состояния, она может перемещаться вниз по состояниям делеции, прежде чем перейти в состояние совпадения или вставки в первом столбце.

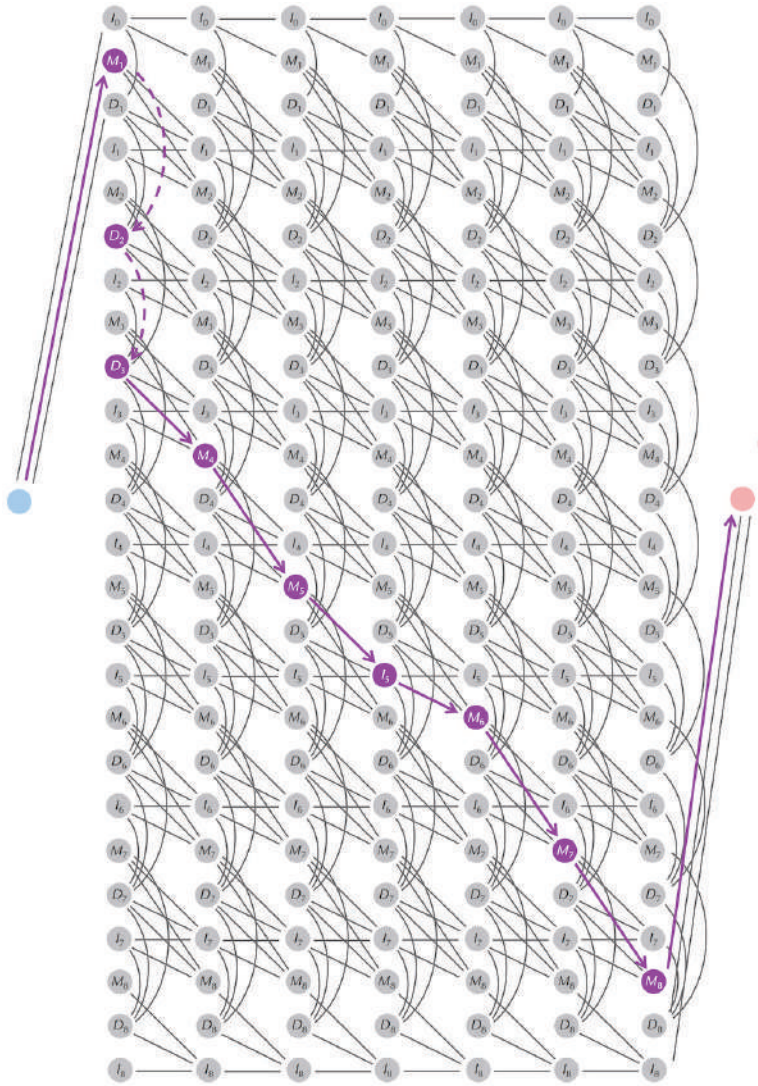


Рис. 10.25 Граф Витерби с |*States*| строками и |*Text*| столбцами для профильной HMM мы рассматривали как возможность создания строки *Text* длиной 7, чтобы ребра, входящие в состояние делеции, шли вниз в пределах одного столбца, а не между столбцами, как на двух ранее предложенных рисунках. Фиолетовый путь соответствует пути через HMM, выдающему **AEFDFDC**

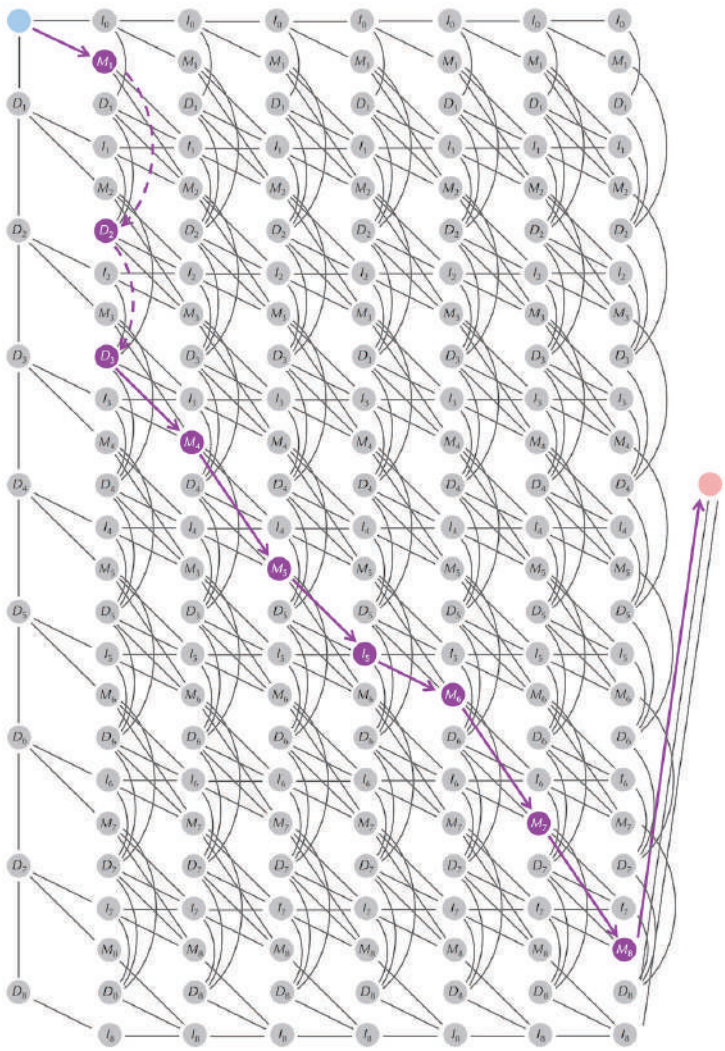


Рис. 10.26 Окончательный граф Витерби для профильной HMM, эмитирующей строку длины 7. Ребра в одном столбце имеют ориентацию вниз; ребра между столбцами имеют правую ориентацию. И снова фиолетовый путь соответствует пути через HMM, выдающему **AEFDFDC**

Теперь вы готовы использовать профильную HMM для выравнивания последовательности с семенем. Единственная оставшаяся ловушка заключается в том, что при вычислении $s_{k,i}$ – или, что то же самое, $\log(s_{k,i})$ – мы должны убедиться, что все входящие счета были вычислены. Поэтому мы предлагаем топологический порядок следования сверху вниз, столбец за столбцом для профильной HMM, показанный на рисунке ниже.

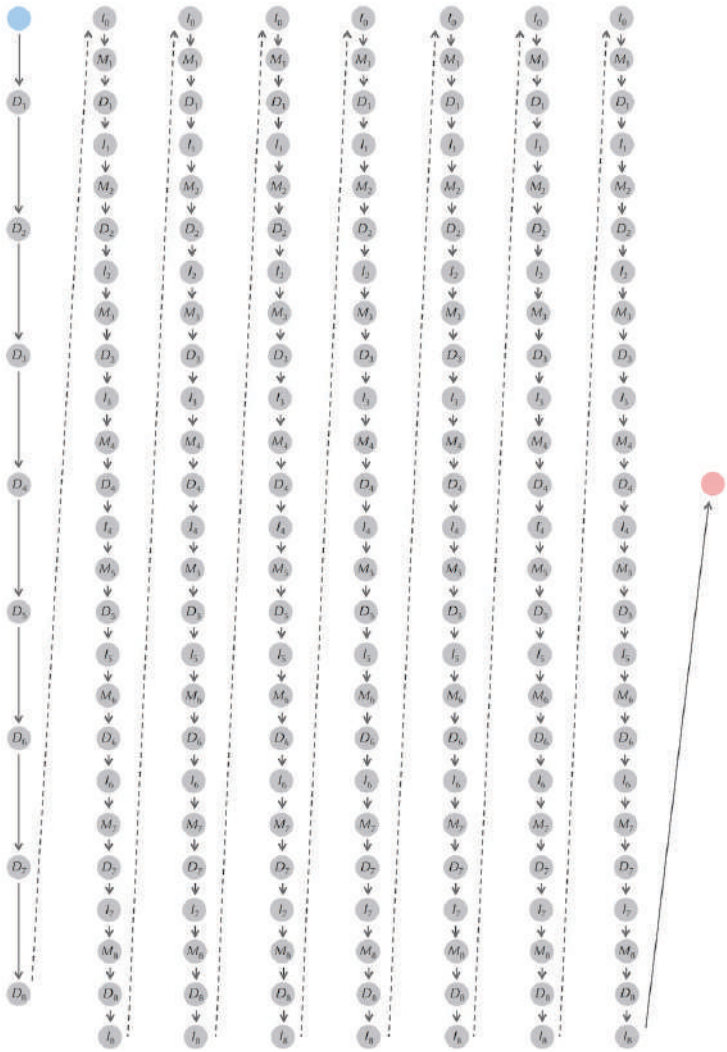


Рис. 10.27 Топологический порядок графа Витерби, который идет сверху вниз, перебирая столбец за столбцом

Задача выравнивания последовательности с профильной HMM: *выровняйте новую последовательность с имеющимся семейством последовательностей, используя профильную HMM.*

Input: множественное выравнивание *Alignment*, пороговое значение θ, значение псевдосчета σ и строка *Text*.

Output: оптимальный скрытый путь, выдающий *Text* в HMM (*Alignment*, θ, σ).

Упражнение. Сделайте выравнивание последовательности с помощью профильной HMM, которую вы создали для множественного выравнивания gp120 (рис. 10.21), вместе с белком gp120, взятым из вируса иммунодефицита обезьян шимпанзе (SIV).

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы построили граф Витерби для произвольной HMM с молчащими состояниями? В каких ситуациях будет невозможно построить граф Витерби CMM с такими состояниями?

Действительно ли профильные HMM так полезны?

Алгоритм Витерби применим для любой HMM, но мы опишем, как он работает для профильных HMM, чтобы отметить один важный момент. Определим $s_{Match(j),i}$ как вероятность наиболее вероятного скрытого пути для префикса $x_1 \ldots x_i$ из x, который заканчивается в состоянии $Match(j)$, и аналогично определим $s_{Insertin(j),i}$ и $s_{Deletion(j),i}$. Поскольку в $Match(j)$ входят только три ребра, рекуррентность Витерби устанавливает, что

$$s_{Match(j),i} = \max \begin{cases} s_{Match(j-1),i-1} \cdot Weight_i(Match(j-1), Match(j)) \\ s_{Insertion(j-1),i-1} \cdot Weight_i(Insertion(j-1), Match(j)) \\ s_{Deletion(j-1),i-1} \cdot Weight_i(Deletion(j-1), Match(j)) \end{cases}.$$

После логарифмирования обеих сторон результирующая рекуррентность очень похожа на стандартное рекуррентное соотношение для глобального попарного выравнивания, поскольку оно представляет собой максимум трех сумм:

$$\log(s_{Match(j),i}) = \max \begin{cases} \log(s_{Match(j-1),i-1}) + \log(Weight_i(Match(j-1), Match(j))) \\ \log(s_{Insertion(j-1),i-1}) + \log(Weight_i(Insertion(j-1), Match(j))) \\ \log(s_{Deletion(j-1),i-1}) + \log(Weight_i(Deletion(j-1), Match(j))) \end{cases}.$$

На рисунке ниже показано, как путь в графе выравнивания типа сетки Манхэттена соответствует пути через профильную HMM. Диагональные, вертикальные и горизонтальные ребра в манхэттенском графе соответствуют состояниям совпадения, вставки и делеции соответственно.

На этом рисунке может показаться, что мы зря потратили ваше время, вводя HMM, поскольку похоже, что профильная HMM каким-то образом эквивалентна попарному выравниванию последовательностей. Однако имейте в виду, что выбор ребер на рисунке ниже основан на различных вероятностях перехода и эмиссии. Выводя отдельные параметры оценки для каждого столбца в матрице выравнивания, профиль HMM позволяет нам уловить тонкие сходства, которые могут остаться незамеченными простыми методами счета, рассмотренным ранее при обсуждении выравнивания последовательностей.

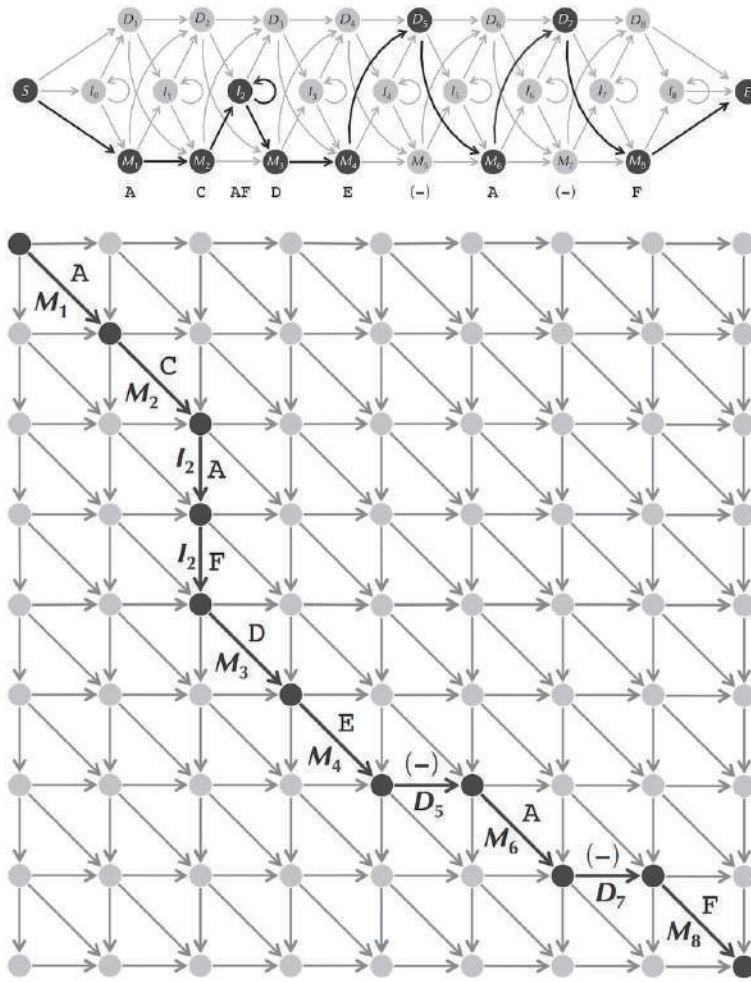


Рис. 10.28 (Вверху) Скрытый путь через профильную HMM, выдающий ACAFDEAF. (Внизу) Путь через манхэттенский граф, соответствующий этому скрытому пути

Обучение параметров HMM

Определение параметров HMM, когда скрытый путь известен

До сих пор наш анализ предполагал, что мы знаем параметры HMM, т. е. вероятности ее перехода и эмиссии. Мы описали наивную – и не обязательно оптимальную – эвристику выбора этих параметров для профильной HMM, но осталось непонятно, как выбирать параметры для произвольной HMM.

Действительно, самой большой сложностью при моделировании биологических проблем с помощью HMM является определение параметров HMM по данным. С точки зрения мошеннического казино представьте, что вы знаете, что дилер использует две монеты для обмана, но вы не знаете смещение (асимметричность) монет или вероятность того, что дилер подменит монеты в любой момент времени. Можете ли вы вывести эти параметры только из последовательности подбрасывания монеты?

ОСТАНОВИТЕСЬ и задумайтесь. Скажем, вы имеете последовательность подбрасывания монеты «OOPOOOPOOPPPO». Каковы ваши наилучшие предположения относительно смещений двух монет и вероятностей переключения с одной монеты на другую? Изменилось бы ваше предположение, если бы вы знали, что скрытый путь равен π = *FFFBBFFFFFFBBB*?

Мы будем называть обе матрицы *Transition* и *Emission* как *Parameters*. Наша цель – найти *Parameters* и π, когда нам дана только строка *x* на выходе. Мы будем работать над достижением этой цели, предполагая, что нам дан *x*, а также либо *Parameters*, либо π, и мы должны вывести оставшийся компонент. Если *x* и *Parameters* известны, то мы можем найти наиболее вероятный скрытый путь π с помощью алгоритма Витерби. Однако мы еще не рассмотрели, как посчитать *Parameters*, если мы знаем *x* и скрытый путь π.

Задача определения параметров HMM: *найти оптимальные параметры, объясняющие выдаваемую строку и скрытый путь HMM.*

Input: строка $x = x_1 \ldots x_n$, выдаваемая HMM с *k*-состоянием с неизвестными вероятностями перехода и эмиссии по известному скрытому пути $\pi = \pi_1 \ldots \pi_n$.

Output: матрица перехода *Transition* и матрица эмиссии *Emission*, которые максимизируют $Pr(x, \pi)$ по всем возможным матрицам перехода и эмиссии.

Если мы знаем и *x*, и π, то можно вычислить эмпирические оценки вероятностей перехода и эмиссии, используя метод, аналогичный тому, который мы

использовали для определения параметров профильных HMM. Если $T_{l,k}$ обозначает число переходов из состояния l в состояние k на скрытом пути π, то мы можем определить вероятность $transition_{l,k}$, вычислив отношение $T_{l,k}$ к общему числу переходов, покидающих состояние l, как

$$transition_{l,k} = \frac{T_{l,k}}{\sum_{\text{all states } j} T_{l,j}}.$$

Аналогичным образом если $E_k(b)$ обозначает количество эмиссий символа b, когда скрытый путь π находится в состоянии k, то мы можем оценить вероятность $emission_k(b)$ как отношение $E_k(b)$ к общему числу выданных символов из состояния k, как

$$emission_k(b) = \frac{E_k(b)}{\sum_{\text{all symbols } c \text{ in the alphabet}} E_k(c)}.$$

Оказывается, что две приведенные выше формулы для вычисления *Transition* и *Emission* приводят к параметрам, решающим задачу определения параметров HMM.

Обучение Витерби

Если мы знаем x и *Parameters*, то можно построить наиболее вероятный путь π, применив алгоритм Витерби для решения задачи декодирования:

$$(x, ?, Parameters) \rightarrow \pi.$$

С другой стороны, если мы знаем x и π, то восстановление *Parameters* сводится к решению задачи определения параметров HMM:

$$(x, \pi, ?) \rightarrow Parameters.$$

ОСТАНОВИТЕСЬ и задумайтесь. Что вам напоминают выражения $(x, \pi, ?) \rightarrow Parameters$ и $(x, ?, Parameters) \rightarrow \pi$?

Задача обучения параметров HMM: *определить параметры HMM, объясняющие выдаваемую строку.*

Input: строка $x = x_1 \ldots x_n$, выдаваемая HMM с неизвестными вероятностями перехода и эмиссии.

Output: матрица перехода *Transition* и матрица эмиссии *Emission*, которые максимизируют $Pr(x, \pi)$ по всем возможным матрицам перехода и эмиссии и по всем скрытым путям π.

К сожалению, задача обучения параметров HMM неразрешима, поэтому вместо этого мы разработаем эвристику, аналогичную алгоритму Ллойда для кластеризации k-средних. В этом алгоритме мы повторили два шага: «от центров к кластерам»:

$$(Data, ?, Centers) \rightarrow HiddenVector,$$

и «от кластеров к центрам»,

$$(Data, HiddenVector, ?) \rightarrow Centers.$$

Что касается определения параметров HMM, мы начинаем с начального случайного предположения для параметров. Затем мы используем алгоритм Витерби, чтобы найти оптимальный скрытый путь π:

$$(x, ?, Parameters) \rightarrow \pi.$$

Как только мы определим π, то поставим под сомнение наш первоначальный выбор параметров и применим наше решение к задаче определения параметров HMM для обновления параметров по x и π:

$$(x, \pi, ?) \rightarrow Parameters'.$$

Затем мы повторяем эти два шага, надеясь, что оценочные параметры все ближе и ближе к параметрам, решающим задачу обучения параметрам HMM:

$$\begin{aligned}(x, ?, Parameters) &\rightarrow (x, \pi, Parameters) \rightarrow (x, \pi, ?)\\ &\rightarrow (x, \pi, Parameters') \rightarrow (x, ?, Parameters')\\ &\rightarrow (x, \pi', Parameters') \rightarrow (x, \pi', ?)\\ &\rightarrow (x, \pi', Parameters'') \rightarrow \ldots\end{aligned}$$

Такой алгоритм определения параметров HMM называется **обучением Витерби**.

ОСТАНОВИТЕСЬ и задумайтесь. Может ли $Pr(x, \pi)$ уменьшаться при обучении Витерби? Когда бы вы решили остановить алгоритм обучения Витерби?

Обратите внимание, что мы не указали, как должно заканчиваться обучение Витерби. На практике существуют различные правила останова для контроля времени его работы. Например, алгоритм может быть остановлен, если количество итераций превышает заданный порог или если $Pr(x, \pi)$ очень мало меняется от одной итерации к другой.

Кроме того, поскольку алгоритм обучения Витерби зависит от начального предположения для *Parameters*, он может застрять в локальном оптимуме. Как

и другие эвристики, он часто запускается много раз, сохраняя наилучший выбор *Parameters*.

Упражнение. Примените обучение Витерби для определения параметров HMM, моделирующей CG-острова, а также для профильной HMM для выравнивания gp120 ВИЧ.

Мягкие решения для определения параметров

Задача мягкого декодирования

В предыдущей главе мы представили «мягкий» алгоритм кластеризации, основанный на более общем алгоритме максимизации ожидания, который смягчил жесткое распределение точек алгоритма Ллойда по кластерам. Аналогичным образом, генерируя единственный оптимальный скрытый путь, алгоритм Витерби дает жесткий ответ «да» или «нет» на вопрос, находилась ли HMM в состоянии k в момент времени i. Но насколько мы уверены, что это именно так?

Возвращаясь к аналогии с мошенническим казино еще раз, предположим, что i-й бросок монеты выпал орлом. Если этот бросок произошел в середине десяти последовательных орлов, то вы должны быть достаточно уверенными, что была использована смещенная монета. Но что, если из десяти бросков, окружающих i-й бросок, шесть выпадают орлом, а четыре – решкой? В этом случае вы должны быть менее уверены в том, что использовалась мошенническая монета.

В случае произвольной HMM мы хотели бы вычислить условную вероятность $Pr(\pi_i = k|x)$ того, что HMM находилась в состоянии k в момент времени i при условии, что она выдал строку x.

Задача мягкого декодирования: *найти вероятность того, что HMM находилась в определенном состоянии в определенный момент, имея выдаваемую ей строку.*

Input: строка $x = x_1 \ldots x_n$, созданная HMM.

Output: условная вероятность $Pr(\pi_i = k|x)$ того, что HMM находилась в состоянии k на шаге i при условии, что она выдал x.

Безусловную вероятность того, что скрытый путь пройдет через состояние k в момент времени i и выдаст x, можно представить в виде суммы:

$$Pr(\pi_i = k, x) = \sum_{\text{all paths } \pi \text{ with } \pi_i = k} Pr(x, \pi).$$

Условная вероятность $Pr(\pi_i = k|x)$ равна пропорции путей, которые проходят через состояние k в момент времени i и выдают x по отношению ко всем путям, выдающим x:

$$Pr(\pi_i = k|x) = \frac{Pr(\pi_i = k, x)}{Pr(x)} = \frac{\sum_{\text{all paths } \pi \text{ with } \pi_i = k} Pr(x, \pi)}{\sum_{\text{all paths } \pi} Pr(x, \pi)}.$$

ОСТАНОВИТЕСЬ и задумайтесь. Если алгоритм Витерби для мошеннического казино выдает путь $\pi = \pi_1, \pi_2, \ldots, \pi_n$ с $\pi_i = B$, является ли более вероятным, что дилер использовал мошенническую монету на шаге i? Возможно ли, что $\pi_i = B$, но $Pr(\pi_i = B|x)$ меньше, чем $Pr(\pi_i = F|x)$?

Алгоритм «вперед–назад»

Заметим, что $Pr(\pi_i = k, x)$ равен сумме весов произведений $Pr(\pi, x)$ всех путей π через граф Витерби для x, проходящих через узел (k, i). Как показано на рисунке ниже, мы можем разбить каждый такой путь на синий подпуть от *source* до (k, i), который мы обозначаем π_{blue}, и (красный) подпуть от (k, i) до стока, который мы обозначаем π_{red}. Запись $Weight(\pi_{blue})$ и $Weight(\pi_{red})$ в качестве соответствующих весов произведений этих подпутей дает рекурсию:

$$\begin{aligned} Pr(\pi_i = k, x) &= \sum_{\text{all paths } \pi \text{ with } \pi_i = k} Pr(x, \pi) \\ &= \sum_{\text{all paths } \pi_{blue}} \sum_{\text{all paths } \pi_{red}} Weight(\pi_{blue}) \cdot Weight(\pi_{red}) \\ &= \sum_{\text{all paths } \pi_{blue}} Weight(\pi_{blue}) \cdot \sum_{\text{all paths } \pi_{red}} Weight(\pi_{red}). \end{aligned}$$

Мы уже вычислили сумму весов произведений всех синих подпутей; это просто $forward_{k,i}$, с которым мы столкнулись при решении задачи вероятности выхода. Теперь мы хотели бы вычислить сумму весов произведений всех красных подпутей, которые обозначаем как $backward_{k,i}$, так что предыдущее уравнение принимает вид:

$$Pr(\pi_i = k, x) = forward_{k,i} \cdot backward_{k,i}.$$

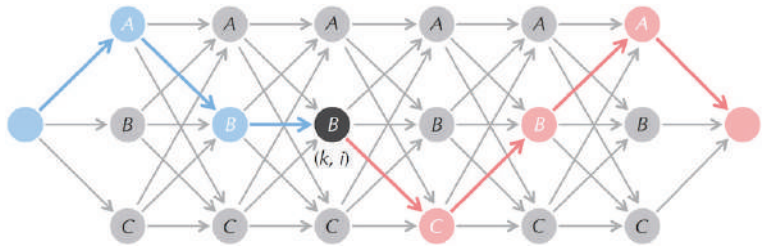


Рис. 10.29 Каждый путь от источника к стоку, проходящий через (черный) узел (*k, i*) в графе Витерби, может быть разделен на два подпути, один от *source* к (*k, i*) (показан синим цветом), а другой от (k , i), к *sink* (показано красным)

Название $backward_{k,i}$ происходит от того факта, что для вычисления этого значения мы можем просто поменять местами направления всех ребер в графе Витерби (см. рис. 10.30) и применить тот же алгоритм динамического программирования, что и для вычисления $forward_{k,i}$. Так как перевернутое ребро, соединяющее $(l, i + 1)$ с (k, i), имеет вес $Weight_i(k, l) = transition_{k,l} \cdot emission_l(x_i + 1)$, мы имеем:

$$backward_{k,i} = \sum_{\text{all states } l} backward_{l,i+1} \cdot Weight_i(k,l).$$

ОСТАНОВИТЕСЬ и подумайте. Как следует инициализировать эту рекурсию?

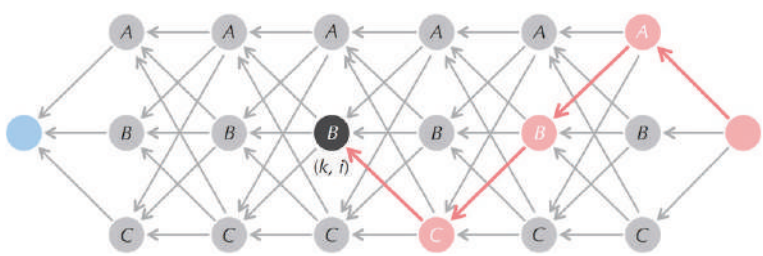


Рис. 10.30 «Обратный граф Витерби», в котором все ребра перевернуты, а путь от *sink* к (*k, i*) выделен красным. Рекуррентность для $backward_{k,i}$ основана на вычислении $backward_{l,i+1}$ для каждого состояния *l*

Полученное в результате приближение динамического программирования для вычисления $Pr(\pi_i = k, x)$ называется алгоритмом «вперед–назад». Сочетание этого алгоритма с нашим решением задачи вероятности исхода для вычисления $Pr(x)$ дает следующее:

$$Pr(\pi_i - k|x) = \frac{Pr(\pi_i - k, x)}{Pr(x)} = \frac{forward_{k,i} \cdot backward_{k,i}}{forward(sink)}.$$

И теперь мы готовы решить задачу мягкого декодирования.

Упражнение. Рассмотрите следующие вопросы.

- Для нечестного дилера HMM вычислите $Pr(\pi_i = k|x)$ для x = «РОРОООРОРРО» и каждого значения i. Как изменится ваш ответ, если x = «ОООООООООО»?
- Примените свое решение задачи мягкого декодирования, чтобы найти CG-острова в первом миллионе нуклеотидов X-хромосомы человека (было предложено загрузить ранее). Чем ваш ответ отличается от решения, данного алгоритмом Витерби?

Мы только что увидели, как вычислить условную вероятность $Pr(\pi_i = k|x)$ того, что HMM проходит через узел (k, i) в графе Витерби при условии, что HMM выдает x. Но как насчет условной вероятности $Pr(\pi_{i+l}, \pi_{i+1} = k|x)$ того, что HMM проходит через ребро, соединяющее (l, i) с $(k, i + 1)$, при условии, что HMM выдает x? Как и в алгоритме «вперед–назад», мы можем разделить каждый путь через рассматриваемое ребро на синий путь от источника к этому ребру и красный путь от этого ребра к стоку (рис. 10.31).

Упражнение. Докажите, что $Pr(\pi_{i+l}, \pi_{i+1} = k|x)$ равно $forward_{l,i} \cdot Weight_i(l, k) \cdot backward_{k,i+1}/forward(sink)$.

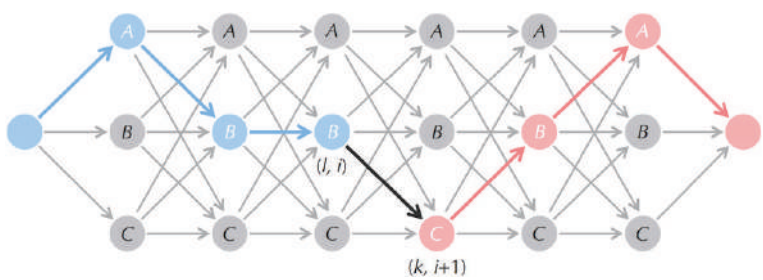


Рис. 10.31 Каждый путь от истока к стоку в графе Витерби, проходящий через (черное) ребро $(l, i) \rightarrow (k, i + 1)$ в графе Витерби, можно разбить на два подпути, один от истока к (l, i) (показан синим цветом) и еще один из $(k, i + 1)$ в сток (показан красным)

Вероятности $Pr(\pi_i = k|x)$ можно поместить в **матрицу ответственности** Π^* размерностью $|States| \times n$, где $\Pi^*_{k,i}$ соответствует узлу графа Витерби и равна $Pr(\pi_i = k|x)$. На рисунке ниже (вверху) показана матрица ответственности Π^* для мошеннического казино.

Вероятности $Pr(\pi_{i+l}, \pi_{i+1} = k|x)$ можно перевести в другую матрицу ответственности Π^{**} размерностью $|States| \times |States| \times (n - 1)$, где $\Pi^{**}_{l,k,i}$ соответствует ребру

в графе Витерби и равно $Pr(\pi_{i+l}, \pi_{i+1} = k|x)$ (рис. 10.32 (внизу)). Для краткости мы используем $\prod$ для общего обозначения матриц Π^* и Π^{**}.

Упражнение. Какова сложность алгоритма вычисления матриц Π^* и Π^{**}?

	P	O	P	O	O	O	P	O	P	P	O
F	0.636	0.593	0.600	0.533	0.515	0.544	0.627	0.633	0.692	0.686	0.609
B	0.364	0.407	0.400	0.467	0.485	0.456	0.373	0.367	0.308	0.314	0.391

	1	2	3	4	5	6	7	8	9	10
FF	0.562	0.548	0.507	0.473	0.478	0.523	0.582	0.608	0.643	0.588
FB	0.074	0.045	0.093	0.059	0.037	0.022	0.045	0.025	0.049	0.098
BF	0.031	0.053	0.025	0.042	0.066	0.104	0.051	0.084	0.043	0.022
BB	0.333	0.354	0.374	0.426	0.418	0.351	0.322	0.282	0.265	0.293

Рис. 10.32 (Вверху) Матрица ответственности Π^*, где x = «POPOOOPOP-PO» и матрицы эмиссии/перехода *Parameters* взяты от мошеннического дилера HMM. $\Pi^*_{k,i}$ равно $Pr(\pi_i = k|x)$. (Внизу) Матрица ответственности Π^{**}, где $\Pi^{**}_{l,k,i} = Pr(\pi_{i+l}, \pi_{i+1} = k|x)$ для тех же выдаваемых матриц строк и эмиссии/перехода

Обучение Баума–Уэлча

Алгоритм максимизации ожидания для счета параметров, называемый **обучением Баума–Уэлча**, состоит из двух этапов. На шаге E он считает профиль ответственности Π с учетом текущих параметров:

$$(x, ?, Parameters) \rightarrow \Pi.$$

Затем на M-этапе пересчитывает параметры из профиля ответственности:

$$(x, \Pi, ?) \rightarrow Parameters.$$

Мы уже реализовали E-шаг алгоритма максимизации ожидания, но остается вопрос, как разработать M-шаг.

Когда мы знаем скрытый путь, ранее определенные оценки для *Parameters*, воспроизведенные ниже, определяют оптимальный выбор для данного скрытого пути π:

$$transition_{l,k} = \frac{T_{l,k}}{\sum_{\text{all states } j} T_{l,j}};$$

$$emission_k(b) = \frac{E_k(b)}{\sum_{\text{all symbols } c \text{ in the alphabet}} E_k(c)}.$$

Здесь $T_{l,k}$ – количество переходов из состояния l в состояние k на скрытом пути π, а $E_k(b)$ – количество раз, когда символ b выдается, когда скрытый путь π находится в состоянии k.

Упражнение. Как бы вы переопределили эти оценочные функции, если скрытый путь неизвестен?

Чтобы увидеть, как определить $transition_{l,k}$ и $emission_k(b)$, когда скрытый путь неизвестен, мы вычислим $T_{l,k}$ и $E_k(b)$ для известного пути π немного другим способом, чтобы сделать переход от жесткого к мягкому выбору более очевидным. Сначала определите следующие бинарные переменные:

$$T_{l,k}^i = \begin{cases} 1, & \text{если } \pi_i = l \text{ и } \pi_{i+1} = k \\ 0 & \text{в противном случае} \end{cases};$$

$$E_k^i(b) = \begin{cases} 1, & \text{если } \pi_i = k \text{ и } x_i = b \\ 0 & \text{в противном случае} \end{cases}.$$

В этих обозначениях формулы для вычисления $T_{l,k}$ и $E_k(b)$ можно переписать как

$$T_{l,k} = \sum_{i=1}^{n-1} T_{l,k}^i \qquad E_k(b) = \sum_{i=1}^{n} E_k^i(b).$$

Когда скрытый путь неизвестен, мы заменим бинарные переменные $T_{l,k}^i$ и $E_k^i(b)$ на новые переменные, которые вычисляются в терминах условных вероятностей того, что скрытый путь пройдет через заданный узел или ребро графа Витерби:

$$\begin{aligned} T_{l,k}^i &= Pr(\pi_i = l, \pi_{i+1} = k | x) \\ &= \Pi_{l,k,i}^{**}; \end{aligned}$$

$$\begin{aligned} E_k^i(b) &= Pr(\pi_i = k | x) \\ &= \Pi_{k,i}^{**}, \text{ если } x_i = b \text{ и } 0 \text{ в противном случае.} \end{aligned}$$

Вооружившись этими вероятностями, вычисленными в предыдущем разделе, мы можем вычислить новые оценки *Parameters*, которые на практике часто работают лучше, чем оценки, полученные методом обучения Витерби:

$$T_{l,k} = \sum_{i=1}^{n-1} \Pi_{l,k,i}^{**} \qquad E_k(b) = \sum_{\text{all } i \text{ such that } x_i = b} \Pi_{k,i}^{*}.$$

Подставив эти значения в приведенные выше формулы для $T_{l,k}$ и $E_k(b)$, получаем обновленные вероятности перехода и эмиссии. Для примера из предыдущего раздела (матрицы П* и П** которого воспроизведены на рис. 10.32) получаем:

$$T_{F,F} = 0.562 + 0.548 + \ldots + 0.588 = 5.512;$$
$$T_{F,B} = 0.074 + 0.045 + \ldots + 0.098 = 0.547;$$
$$T_{B,F} = 0.031 + 0.053 + \ldots + 0.022 = 0.521;$$
$$T_{B,B} = 0.333 + 0.354 + \ldots + 0.293 = 3.418;$$

$$E_F(H) = 0.593 + 0.533 + 0.515 + 0.544 + 0.633 + 0.609 = 3.427;$$
$$E_F(T) = 0.636 + 0.600 + 0.627 + 0.692 + 0.686 = 3.241;$$
$$E_B(H) = 0.407 + 0.467 + 0.485 + 0.456 + 0.367 + 0.391 = 2.573;$$
$$E_B(T) = 0.364 + 0.400 + 0.373 + 0.308 + 0.314 = 1.759.$$

Упражнение. Используйте обучение Баума–Уэлча, чтобы изучить параметры для HMM, моделирующей CG-острова, и для HMM профиля HIV. Сравните эти параметры с параметрами, полученными путем применения обучения Витерби.

Многоликость HMM

Профильные HMM для множественного выравнивания последовательностей и HMM, обнаруживающие CG-острова, – это всего лишь два примера множества приложений HMM в биоинформатике. Кроме того, применение HMM для анализа ВИЧ (HIV) не ограничивается профильными HMM, но также включает анализ устойчивости ВИЧ к противовирусной лекарственной терапии.

В начале главы мы упоминали, что пациентов, инфицированных ВИЧ, лечат комплексом из нескольких препаратов. Эти препараты пытаются подавить репликацию вируса, но ВИЧ часто мутирует в штаммы, устойчивые к лекарствам, и эти штаммы в конечном итоге доминируют в популяции вируса в организме хозяина, что постепенно делает смесь лекарств неэффективной. Вирусы ВИЧ часто секвенируют после того, как лекарственная терапия не удалась, чтобы решить, как изменить состав лекарственного коктейля. Таким образом, понимание путей ВИЧ в устойчивости к лекарствам важно для разработки эффективного коктейля.

Тем не менее моделирование путей устойчивости ВИЧ к лекарствам является сложной задачей. Две полезные для вируса мутации могут синергетически взаимодействовать, в результате чего двойная мутация будет фиксироваться чаще, чем можно было бы предсказать, исходя из частот отдельных замен. Мутации также могут взаимодействовать антагонистически, в результате чего мутанты менее приспособлены, чем мы могли бы предсказать.

В 2007 году Нико Беренвинкель и Матиас Дртон представили основанную на HMM модель эволюции ВИЧ и развития устойчивости к лекарствам. Однако их HMM слишком сложна, чтобы дать ее здесь. Тем не менее мы упомянули о ней, чтобы подчеркнуть силу HMM. Несмотря на то что модели HMM могут показаться простыми машинами, которые подбрасывают монеты и выдают символы, их можно применять для решения сложных задач биоинформатики, начиная от предсказания генов и заканчивая поиском регуляторных мотивов.

Эпилог. Природа – мастер, а не изобретатель

Последовательность аминокислот в белке кодирует его трехмерную структуру, которая часто определяет биологическую функцию белка. Например, **цинковый палец** является элементом трехмерной структуры белков (рис. 10.33). Располагая два цистеина и два гистидина близко друг к другу в аминокислотной последовательности белка цинковых пальцев, белок способен «захватывать» ион цинка и плотно сворачиваться вокруг него. Цинковые пальцы настолько полезны, что их можно найти в тысячах белков человека. Кроме того, белки с цинковыми пальцами используются не только для связывания цинка, поскольку многие из этих белков связываются с другими металлами и даже с неметаллами.

В настоящее время известно более 100 000 экспериментально определенных белковых структур, но многие из них представляют собой очень похожие структуры или имеют сегменты с очень похожей структурой. **Белковый домен** представляет собой консервативную часть белка, которая может функционировать независимо от остальной его части. Домены различаются по длине, но средняя длина домена составляет примерно 100 аминокислот (домены с цинковыми пальцами имеют длину всего 20–30 аминокислот). Многие белки состоят из нескольких доменов, и один и тот же домен может появляться (с вариациями) во многих белках.

Лауреат Нобелевской премии Франсуа Жакоб в 1977 году сказал: «Природа – мастер, а не изобретатель». В соответствии с этим принципом природа использует домены в качестве строительных блоков, перетасовывая их в различные конфигурации для создания **многодоменных** белков. Большинство доменов когда-то существовали как независимые белки; например, многие домены, принадлежащие многодоменным белкам человека, могут быть обнаружены как однодоменные белки у бактерий. Многодоменные белки возникают естественным образом, когда рекомбинация генома создает новую кодирующую белок последовательность, содержащую части кодирующих последовательностей двух разных генов. Объединение двух доменов в один белок часто обеспечивает эволюционное преимущество, например, когда оба домена являются ферментами, и в этом случае для клетки может быть полезно обеспечить фиксированное соотношение активности ферментов один к одному.

Поскольку белки часто состоят из нескольких доменов с различной структурой и функциями, биологи обычно анализируют отдельные домены, а не целые

белки, чтобы понять эволюционные отношения. Поскольку сходство последовательностей между доменами со сходной структурой может быть крайне низким, классификация доменов по структурным семействам может быть затруднена. **База данных Pfam**, которая содержит более 10 000 полученных с помощью HMM множественных выравниваний семейств белковых доменов, может использоваться для анализа новых белковых последовательностей.

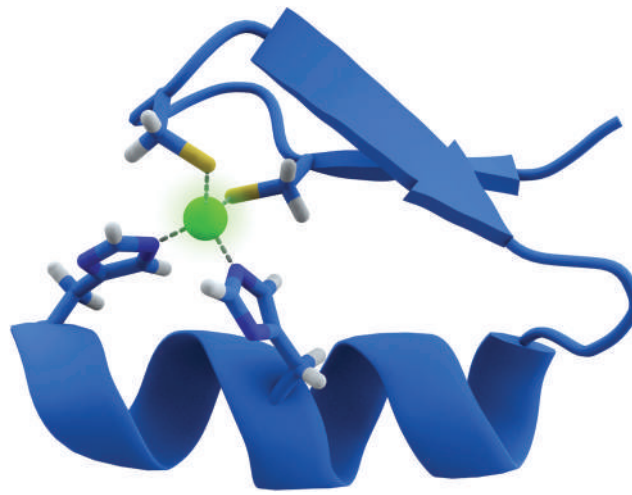


Рис. 10.33 Ион цинка (показан зеленым) удерживается на месте двумя остатками гистидина и двумя остатками цистеина внутри цинкового пальца. Предоставлено Томасом Сплеттстоессером

Заключительная задача. Используя Pfam HMM для gp120[1] (созданного из начального выравнивания всего 24 белков gp120), постройте выравнивания всех известных белков gp120 и определите «наиболее отличающуюся» последовательность gp120.

Сопутствующие материалы

Эффект Красной Королевы

Эффект Красной Королевы (Также используется название «Эффект Черной Королевы». – *Прим. ред.*) – это гипотеза о том, что эволюция необходима не только для того, чтобы дать организмам преимущество в фиксированной среде, но и для того, чтобы помочь им выжить в ответ на изменение окружающей сре-

[1] http://pfam.xfam.org/family/gp120.

ды. Его название происходит от заявления, которое Красная Королева сделала Алисе в «Зазеркалье» Льюиса Кэрролла:

> *А здесь, как видите, чтобы удержаться на одном и том же месте, приходится бежать изо всех сил.*

Эффект Красной Королевы часто наблюдается в отношениях хищник–жертва. Например, адаптация может помочь волкам бежать немного быстрее, а северные олени, в свою очередь, должны эволюционировать, чтобы выжить. В результате кажется, что волки и олени бегают с одинаковой скоростью, причем самые медленные волки голодают, а самых медленных оленей съедают.

Гликозилирование

Клетки имеют на своей поверхности плотное покрытие из полисахаридов, называемых **гликанами**. Гликаны часто представляют собой посттрансляционные модификации гликопротеинов, которые модулируют взаимодействия с другими клетками в многоклеточном организме или между клеткой и другим организмом (например, между клетками человека и вирусом). Например, заражение гриппом начинается с взаимодействия между белками на поверхности вируса и гликанами на поверхности клетки-хозяина.

Гликаны построены из строительных блоков семейства **моносахаридов**. Каждый моносахарид может быть связан с другими моносахаридами с образованием сложных древовидных структур (рис. 10.34).

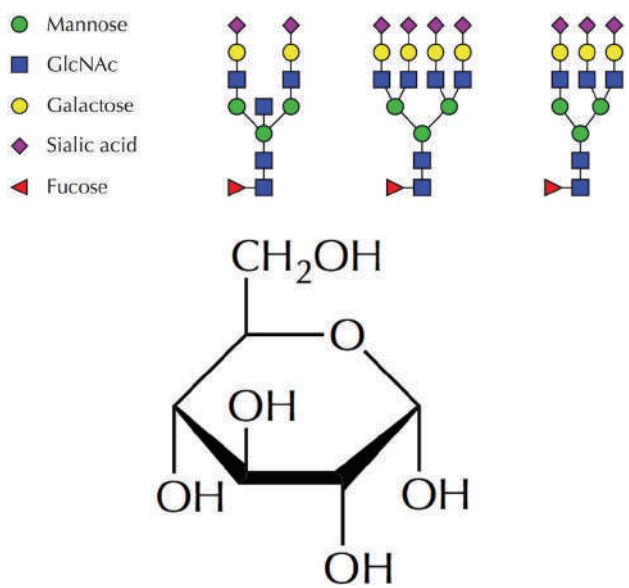


Рис. 10.34 (Вверху) Пять типов моносахаридов вместе с тремя примерами того, как эти моносахариды собираются в гликаны у человека. (Внизу) Химическая формула моносахарида галактозы

Метилирование ДНК

Метилирование ДНК приводит к добавлению метильной группы (CH_3) к цитозиновому или гуаниновому нуклеотиду (рис. 10.35), что часто изменяет экспрессию близлежащих генов. Гены, которые приобретают высокую концентрацию метилированных остатков в своих восходящих областях, имеют подавленную экспрессию. Метилирование ДНК жизненно важно для развития, и как гиперметилирование, так и гипометилирование ДНК связаны с различными видами рака.

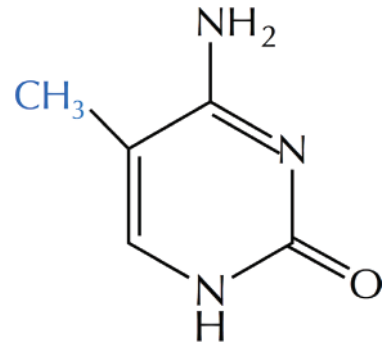


Рис. 10.35 Метилирование ДНК цитозинового нуклеотидного основания, метильная группа показана синим цветом

Метилирование ДНК играет важную роль в процессе **дифференциации клеток**, при котором эмбриональные стволовые клетки становятся специализированными тканями. Это изменение часто является постоянным, предотвращая превращение клетки в стволовую клетку или преобразование в другой тип клеток. Метилирование наследуется во время клеточного деления, но обычно удаляется во время образования зиготы.

Условная вероятность

Вернемся к игре Чо-Хан и проанализируем сумму двух стандартных шестигранных костей. Пусть A – событие, когда s нечетное, а B – событие, когда s больше 10. Вероятность A равна 1/2, потому что половина из 36 возможных результатов броска двух игральных костей дает нечетную сумму. Вероятность B равна 3/36, потому что есть три исхода (5 + 6, 6 + 5 и 6 + 6), для которых $s > 10$.

ОСТАНОВИТЕСЬ и задумайтесь. Если мы скажем вам, что s больше 10 (но вы не видите игральные кости), будет ли s с большей вероятностью четным или нечетным?

Условная вероятность события A при данном событии B, обозначаемая $Pr(A|B)$, представляет собой вероятность того, что событие A произойдет при условии, что произошло событие B. Для примера с игральными костями, поскольку B соответствует двум броскам с s нечетным (6 + 5 и 5 + 6) и одному броску с четным s (6 + 6), $Pr(A|B) = 2/3$. Обратите внимание, что $Pr(A|B)$ полностью отличается от вероятности $Pr(A|B)$ того, что произойдут оба события A и B, которая равна 2/36 (A и B происходят вместе только для сумм 6 + 5 и 5 + 6).

В более общем смысле условная вероятность $Pr(A|B)$ часто определяется по следующей формуле:

$$Pr(A|B) = \frac{Pr(A,B)}{Pr(B)}.$$

Чтобы проверить свои знания об условной вероятности, рассмотрите следующую задачу под названием **Парадокс Монти Холла**, которая первоначально была опубликована в журнале «Американский статистик» профессором Калифорнийского университета Стивеном Селвином в 1975 году и на протяжении многих лет ставила в тупик многих начинающих математиков: (Задача формулируется как описание игры, основанной на американском телешоу «Let's Make a Deal», и названа в честь ведущего этой передачи. – *Прим. ред.*)

Предположим, вы участвуете в игровом шоу и вам предоставляется выбор из трех дверей: за одной дверью стоит автомобиль; за другой – козлы. Вы выбираете дверь, скажем, № 1, и ведущий, который знает, что находится за дверью, открывает другую дверь, скажем, № 3, за которой – козел. Затем он говорит вам: «Не желаете ли вы изменить свой выбор и выбрать дверь № 2?» Увеличатся ли ваши шансы выиграть автомобиль, если вы примете предложение ведущего и измените свой выбор? (Эту задачу называют «парадоксом», потому что интуитивный ответ не нее является неверным. Для стратегии выигрыша важно следующее: если вы меняете выбор двери после действий ведущего, то вы выигрываете, если изначально выбрали проигрышную дверь. Это произойдет с вероятностью 2⁄3, так как изначально выбрать проигрышную дверь можно двумя способами из трех. – *Прим. ред.*)

Библиографические примечания

Алгоритм декодирования был разработан Viterbi, 1967[1]. Первые алгоритмы счета параметров CMM были разработаны Baum et al., 1970[2]. Churchill, 1989[3], Krogh et al., 1994[4], и Baldi et al., 1994[5], первыми применили HMM в вычислительной биологии. Bateman et al., 2002[6], описали применение выравнивания профилей HMM для разработки базы данных семейств белковых доменов под названием **Pfam**. De Jong et al.,1992[7], открыли правило 11/25. Beerenwinkel and Drton, 2007[8], разработали HMM для анализа устойчивости к ВИЧ.

[1] https://ieeexplore.ieee.org/document/1054010.
[2] https://www.semanticscholar.org/paper/A-Maximization-Technique-Occurring-in-the-Analysis-Baum-Petrie/3092a4929bdb3d6a8fe53f162586b7431b5ff8a4.
[3] https://link.springer.com/article/10.1007/BF02458837.
[4] https://www.ncbi.nlm.nih.gov/pubmed/8107089.
[5] https://www.ncbi.nlm.nih.gov/pubmed/8302831.
[6] https://www.ncbi.nlm.nih.gov/pubmed/11752314.
[7] https://www.ncbi.nlm.nih.gov/pubmed/1404617.
[8] https://www.ncbi.nlm.nih.gov/pubmed/16569743.

Глава 11

Является ли T. rex всего лишь гигантской курицей?

Палеонтология встречается с информатикой

Выросший в Монтане в 1950-х годах, Джек Хорнер был застенчивым и замкнутым. Он так медленно продвигался в чтении и математике, что другие дети называли его глупым. Однако его школьный проект о динозаврах получил высшие награды на местной научной ярмарке и был замечен профессором Университета Монтаны, который помог Джеку поступить в университет.

Тем не менее успеваемость Хорнера и в колледже не улучшилась; он был неуспевающим студентом два семестра подряд и был отчислен. Спустя годы Хорнер узнал, что страдает **дислексией**, нарушением развития, которое часто характеризуется трудностями с распознаванием прочитанного и математических символов, несмотря на его интеллект выше среднего уровня.

К счастью для Хорнера, он в конце концов нашел свое призвание. Его мобилизовали во время войны во Вьетнаме, затем он работал водителем грузовика, а потом устроился техником в Принстонский Музей естественной истории, где быстро завоевал репутацию блестящего исследователя. Впоследствии он стал самым известным палеонтологом в мире, вдохновил одного из главных героев бестселлера «Парк Юрского периода» и даже давал советы Стивену Спилбергу по экранизации его известного фильма.

Хорнер смог добиться успеха, несмотря на дислексию, отчасти потому, что палеонтология традиционно не требовала беглости мысли, как математика. Однако собственный ученик Хорнера показал, что даже палеонтология не может обойтись без вычислений. В 2000 году Хорнер исследовал свое любимое кладбище динозавров в Монтане и обнаружил окаменелость кости ноги *Tyrannosaurus rex* возрастом 68 млн лет. Три года спустя он передал небольшой кусочек этой окаменелости своей ученице Мэри Швейцер, которая растворила его в деминерализирующей ванне для изучения состава, но оставила там образец слишком надолго (помните Александра Флеминга?). Когда она вернулась, все, что осталось, было волокнистой субстанцией. Затем Швейцер отправил этот материал масс-спектрометристу (Джон Асара) в надежде обнаружить пептиды тираннозавра рекса, чудом уцелевшие внутри кости.

В 2007 году, проанализировав тысячи спектров, Асара и Швейцер опубликовали статью в журнале Science, в которой объявили об открытии пептидов *T. rex*, которые очень похожи на куриные пептиды. Их результат стал первым молекулярным доказательством противоречивой гипотезы о том, что птицы произошли от динозавров.

Тот факт, что белки могут избежать разрушения миллионы лет, был настолько удивительным, что привел ко многим грандиозным заявлениям. Палеонтолог Ханс Ларссон предположил, что в настоящее время «динозавры вошли в область молекулярной биологии и палеонтологии». *The Guardian* прогнозирует, что «однажды ученые смогут подражать Парку Юрского периода, клонировав динозавра». Сам Хорнер даже опубликовал книгу под названием «Как создать динозавра», в которой подробно описал свой план воссоздания динозавра путем генетической модификации генома курицы.

Тем не менее некоторые ученые оставались скептичными. В то время как предыдущие исследования динозавров не требовали больших вычислений, анализ *T. rex* Асары был основан на алгоритме со сложной статистикой. В 2008 году журнал *Science* опубликовал опровержения, в которых утверждалось, что Асаре и Швейцеру не удалось доказать, что некоторые из их пептидов не являются просто статистическими артефактами. Но как узнать, кто прав? В этой главе мы исследуем пептиды *T. rex*, углубившись в некоторые алгоритмы анализа спектров.

Какие белки присутствуют в этом образце?

Только четверо ученых когда-либо были удостоены двух Нобелевских премий. Один из них – Фредерик Сенгер, о сборке первого генома которого в 1977 году мы упоминали в предыдущей главе. Тем не менее Сенгер уже получил свою первую Нобелевскую премию два десятилетия назад за определение последовательности 52 аминокислот, составляющих инсулин, белок, необходимый для переработки глюкозы в крови. Подобно тому, как ученые секвенируют геномы, Сенгер разбил несколько молекул инсулина на короткие пептиды, упорядочил эти пептиды, а затем собрал их в аминокислотную последовательность инсулина (рис. 11.1).

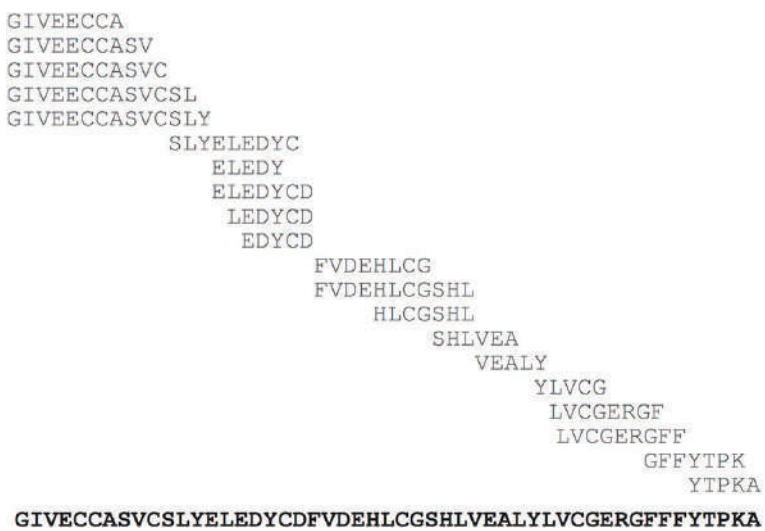


Рис. 11.1 Сборка пептидов, которую Фредерик Сенгер использовал для определения аминокислотной последовательности инсулина

В 1950-х секвенирование белков было очень сложным процессом, а секвенирование ДНК вообще невозможно. Сегодня генерировать миллионы ридов для секвенирования ДНК стало, по существу, тривиально, но секвенирование белков остается сложной задачей. По этой причине большинство белков обнаруживают, сначала секвенируя геном, а затем предсказывая гены, которые

кодирует этот геном (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Предсказание генов**). Транслируя нуклеотидную последовательность каждого гена, кодирующего белок, в аминокислотную последовательность, биологи получают предполагаемый **протеом** вида, т. е. набор всех его белков.

Однако разные клетки организма экспрессируют разные белки. Например, клетки мозга экспрессируют белки, дающие начало нейропептидам, тогда как другие клетки этого не делают. Важной задачей изучения белков, или **протеомики**, является выявление того, какие именно *специфические* белки присутствуют в каждой биологической ткани в различных условиях и как эти белки взаимодействуют между собой.

Например, предположим, мы изучаем рибосому курицы, сложную молекулярную машину, состоящую из множества белков. Знание протеома курицы не говорит нам, какие именно белки составляют рибосомный комплекс. Вместо этого мы можем изолировать рибосому, разбить ее на части и определить, какие белки в ней содержатся. На практике простого подтверждения того, что в образце присутствует пептид длиной 10 аминокислот из известного куриного белка, обычно достаточно, чтобы подтвердить присутствие этого белка в образце. Процесс подтверждения того, что пептид из известного протеома присутствует в образце, называется **идентификацией пептида**. Но как мы можем сформировать протеом *T. rex*?

Хотя идентификация пептидов доминирует в современных исследованиях протеомики, протеомы многих видов, включая вымершие виды, такие как *T. rex*, остаются неизвестными. В этом случае биологи полагаются на **секвенирование пептидов** *de novo* или вывод аминокислотной последовательности пептида, не полагаясь на протеом; с этого мы и начнем.

Расшифровка идеального спектра

Возможно, вы испытываете *дежа вю*, так как мы уже обсуждали секвенирование *циклических* пептидов ранее. Итак, сначала напомним вам основы масс-спектрометрии, уделив особое внимание ее применению для линейного секвенирования пептидов.

Имея большое количество идентичных копий пептида в образце, который обычно содержит миллионы клеток, масс-спектрометр разбивает каждую копию на два меньших фрагмента, причем разные копии одного и того же пептида могут разбиваться по-разному. Например, одна копия REDCA может разделиться на RE и DCA, а другая – на RED и CA. Фрагменты RE и RED называются **префиксами** REDCA, а DCA и CA – **суффиксами** REDCA. На рисунке ниже показаны массы аминокислот в целых числах.

G	A	S	P	V	T	C	I	L	N	D	K	Q	E	M	H	F	R	Y	W
57	71	87	97	99	101	103	113	113	114	115	128	128	129	131	137	147	156	163	186

Рис. 11.2 Таблица целочисленных масс 20 стандартных аминокислот (воспроизведена из раздела по антибиотикам)

Алгоритмический вопрос, который мы сначала задаем, аналогичен тому, что мы задавали о циклических антибиотиках, но теперь применим к линейным пептидам: если мы взвесим каждый префикс и суффикс неизвестного пептида, сможем ли мы реконструировать пептид? Для заданной аминокислотной цепи *Peptide* ее **идеальный спектр**, обозначаемый *IdealSpectrum*(*Peptide*), представляет собой набор целых масс всех ее префиксов и суффиксов (рис. 11.3). Обратите внимание, что идеальный спектр может иметь повторяющиеся массы; например, *IdealSpectrum*(GPG) = {0, 57, 57, 154, 154, 211}. Мы говорим, что аминокислотная строка *Peptide* объясняет набор целых чисел *Spectrum*, если *IdealSpectrum*(*Peptide*) = *Spectrum*.

Задача расшифровки идеального спектра: реконструировать пептид по его идеальному спектру.

Input: набор целых чисел *Spectrum*.

Output: пептид, представляющий собой аминокислотную цепь, которая объясняет *Spectrum*.

Fragment	""	R	RE	RED	REDC	REDCA	EDCA	DCA	CA	A
Mass	0	156	285	400	503	574	418	289	174	71

Рис. 11.3 Массы префиксов и суффиксов REDCA составляют *IdealSpectrum*(REDCA) = {0, 71, 156, 174, 285, 289, 400, 418, 503, 574}

Мы хотели бы разделить массы в спектре на производные от префиксных и суффиксных пептидов, но неясно, как это сделать. Вместо этого заметьте, что если две массы «отстоят друг от друга на одну массу аминокислоты», то вполне вероятно, что они соответствуют двум префиксам или двум суффиксам, которые отличаются одной аминокислотой. Например, мы не знали бы, что массы 400 и 503 соответствуют префиксам RED и REDC (рис. 11.3). Но мы могли бы предположить, что, поскольку разница между этими массами составляет 103 (масса C), эти массы соответствуют префиксам или суффиксам, различающимся в единственном вхождении C.

Эта идея мотивирует основанный на графах метод решения задачи декодирования идеального спектра. Представим массы в спектре как последовательность *Spectrum* целых чисел $s_1, \ldots, s_m$ в порядке возрастания, где s_1 равно нулю, а s_m – общая масса (неизвестного) пептида. Мы определяем размеченный граф *Graph*(*Spectrum*), формируя узел для каждого элемента *Spectrum*, затем соединяя узлы s_i и s_j направленным ребром, помеченным аминокислотой a, если разность $s_j - s_i$ равна массе a (рис. 11.4). Как и предполагалось при секвенировании антибиотиков, мы не различаем аминокислоты, имеющие одинаковые целочисленные массы (т. е. пары K/Q и I/L).

Упражнение. Докажите, что для любого выбора *Spectrum* граф *Graph*(*Spectrum*) является DAG.

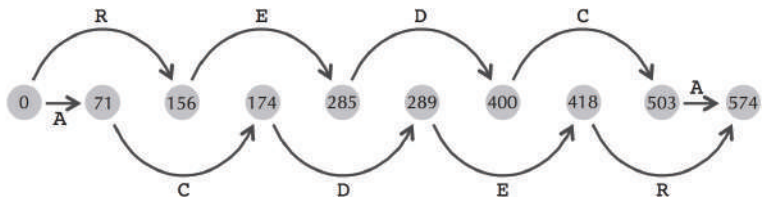


Рис. 11.4 Граф DAG *Graph*(*IdealSpectrum*(REDCA))

Обратите внимание, что *Graph*(*IdealSpectrum*(REDCA)), воспроизведенный ниже, состоит из двух путей, соединяющих *source* = 0 и *sink* = s_m. Объединение аминокислот по этим путям дает REDCA и его реверсный ACDER, оба из которых представляют собой решения задачи декодирования идеального спектра. Метод секвенирования, основанный на написании пути от источника к стоку в *Graph*(*Spectrum*), описывается следующим псевдокодом.

```
DecodingIdealSpectrum(Spectrum)
    построить Graph(Spectrum)
    найти путь Path от source до sink в Graph(Spectrum)
    return строка аминокислоты, написанная метками Path
```

Упражнение. Расшифруйте идеальный спектр {0, 57, 114, 128, 215, 229, 316, 330, 387, 444}.

Если вы попытались выполнить это упражнение, то, скорее всего, вы нашли путь, соответствующий пептиду с неправильным спектром (показан ниже). Это правда, что каждый пептид, объясняющий *Spectrum*, соответствует пути от источника к стоку в *Graph*(*Spectrum*). Однако не каждый путь от источника к стоку на этом графе соответствует пептиду, объясняющему спектр; рассмотрим, например, путь, обозначающий GGDTN на рисунке ниже. По этой причине мы должны переписать приведенный выше ошибочный псевдокод следующим образом.

```
DecodingIdealSpectrum(Spectrum)
    построить граф Graph(Spectrum)
    for каждого пути Path от source до sink в графе Graph(Spectrum)
        Peptide ← строка аминокислоты, написанная краевыми метками Path
        if IdealSpectrum(Peptide) = Spectrum
            return Peptide
```

Упражнение. Какие массы в приведенном ниже спектре пептид GGDTN объяснить не может?

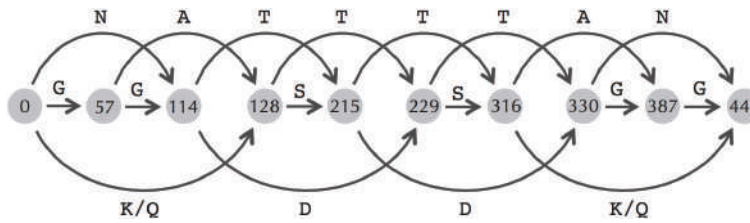


Рис. 11.5 Граф DAG *Graph*(*Spectrum*) для спектра = {0, 57, 114, 128, 215, 229, 316, 330, 387, 444}. Только восемь из 32 путей от *source* до *sink* в этом графе соответствуют пептидам, объясняющим спектр

Хотя **DecodingIdealSpectrum** решает задачу декодирования идеального спектра, исследование всех путей в DAG может занять много времени, поскольку количество таких путей может быть экспоненциальным по отношению к количеству масс в спектре (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Поиск всех путей в графе**).

От идеального спектра к реальному

Мы уже знаем из нашего анализа антибиотиков, что реалии секвенирования пептидов более суровы, чем реконструкция пептида по идеальному спектру. После разделения каждой копии пептида на два меньших фрагмента масс-спектрометр ионизирует их, в результате чего образуются электрически заряженные ионы. Он измеряет отношение массы к заряду каждого ионизированного фрагмента, а также его интенсивность, или количество ионизированных фрагментов для соответствующего отношения массы к заряду (пептиды могут часто разрываться по некоторым связям и почти никогда по другим связям). В результате спектр представляется в виде набора пиков на диаграмме, где координата *x* пика представляет его отношение массы к заряду, а его высота представляет его интенсивность.

Современные масс-спектрометры имеют ограничения на диапазон отношений массы к заряду, которые они могут обнаружить, что затрудняет анализ целых белков с помощью масс-спектрометрии. В результате белки обычно анализируют, сначала разбивая их на более короткие пептиды с помощью ферментов, называемых **протеазами**. Самая популярная протеаза, используемая в протеомике, и та, что использовалась в исследовании *T. rex*, называется **трипсин**. Эта протеаза обычно расщепляет белок после аминокислот R и K и дает пептиды средней длины 14.

На рисунке ниже показан один из масс-спектров *T. rex* (далее именуемый *DinosaurSpectrum*) вместе с двумя его предполагаемыми интерпретациями, ATKIVDCFMTY и GLVGAPGLRGLPGK. Как только мы выведем пептид, сгенерировавший данный спектр, мы можем аннотировать спектр, установив соответствие между массами спектральных пиков и массами префиксов/суффиксов пептида. Чтобы соответствовать стандартной терминологии масс-спектрометрии, пик, аннотированный как префикс длины i, помечен как b_i, а пик, аннотированный как суффикс длины i, помечен как y_i.

ОСТАНОВИТЕСЬ и задумайтесь. Какая из двух интерпретаций *DinosaurSpectrum* на рисунке ниже, по вашему мнению, объясняет его лучше?

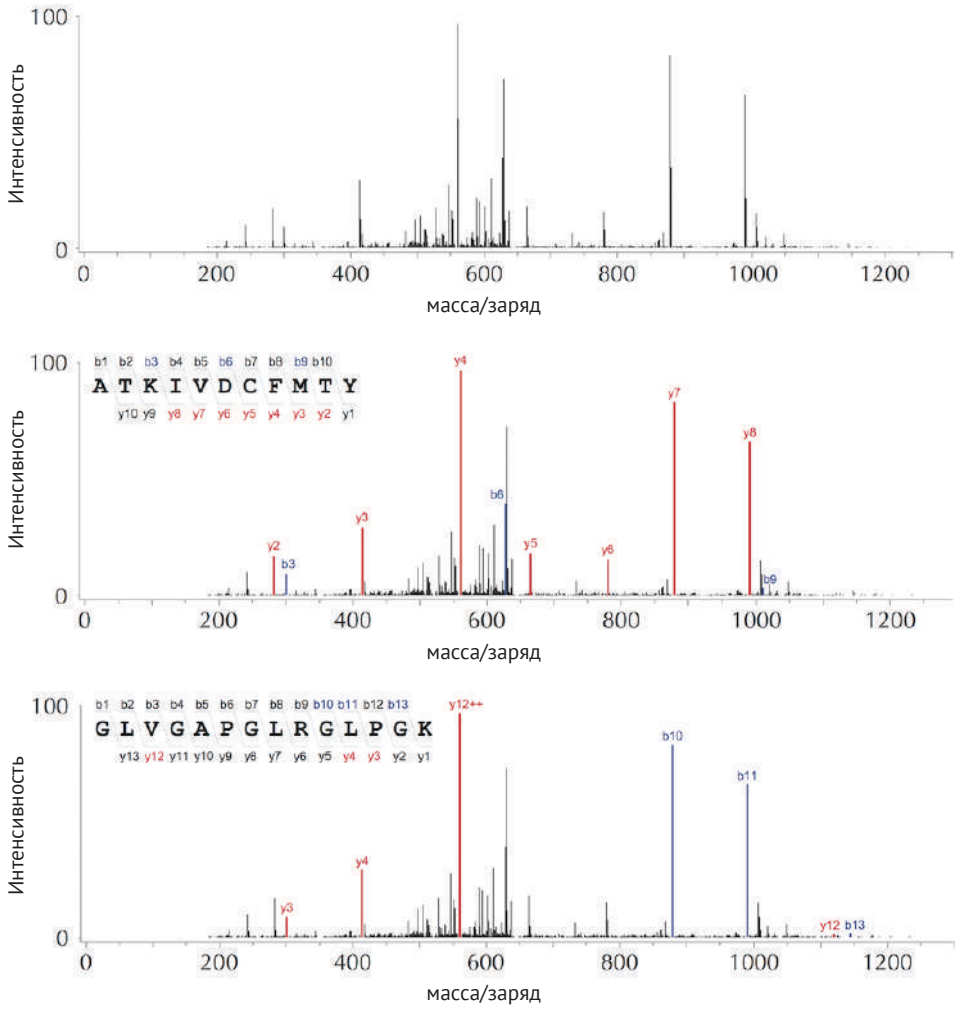


Рис. 11.6 (Вверху) Спектр *DinosaurSpectrum*. (В центре) Тот же спектр, аннотированный ATKIVDCFMTY. (Внизу) Тот же спектр, аннотированный GLVGAPGLRGLPGK. Пик, соответствующий префиксному пептиду длины *i*, обозначен как b_i, а пик, соответствующий суффиксному пептиду длины *i*, обозначен как y_i. Например, пик, обозначенный как b_{10}, соответствует GLVGAPGLRGL, а пик, обозначенный как y_3, соответствует PGK. Большинство аннотированных пиков имеют заряд +1, но некоторые (например, пик, обозначенный y_{12}++) имеют заряд +2. Заряд иона-фрагмента, представленный данным пиком в спектре, заранее неизвестен, но его часто можно предположить после того, как был идентифицирован пептид, сгенерировавший спектр. Только шесть пиков в *DinosaurSpectrum* аннотированы GLVGAPGLRGLPGK; пики b_{10}, b_{11} и b_{13} аннотированы префиксными пептидами, а пики y_3, y_4 и y_{12} – суффиксными пептидами

Секвенировать пептид по его реальному спектру сложнее, чем может показаться. Масс-спектры часто имеют «зашумленные» пики, вносящие вклад в ложные массы, которые могут иметь более высокую интенсивность, чем пики, соответствующие истинным префиксам и суффиксам. Поскольку некоторые пептидные связи почти никогда не рвутся, интенсивности при разных отношениях массы к заряду могут различаться на порядки. В результате спектр может не иметь пиков, соответствующих истинным префиксам и суффиксам. Например, в аннотированном спектре, соответствующем второму пептиду-кандидату из рис. 1.6 (внизу), в *DinosaurSpectrum* нет пиков, аннотированных как b_5 или y_9. По этой причине, хотя мы и игнорировали интенсивность при секвенировании антибиотиков, в этой главе мы будем относиться к ней более серьезно.

Упражнение. REDCA – это один из возможных пептидов, объясняющий некоторые, но не все массы в спектре, показанном на рисунке ниже, который имеет ложные и отсутствующие массы. Можете ли вы найти другой пептид, объясняющий большее количество масс в этом спектре?

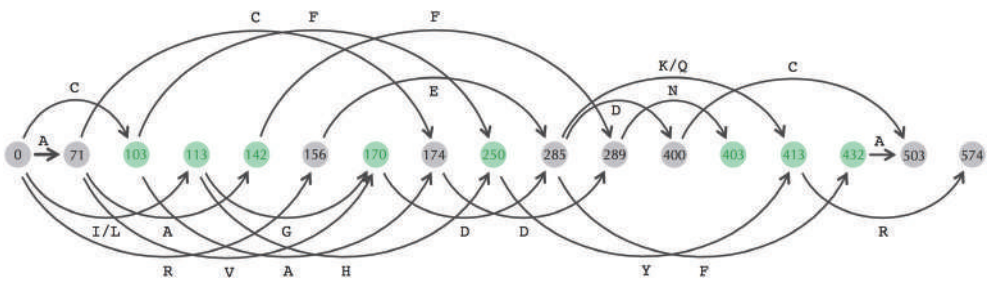


Рис. 11.7 Граф DAG *Graph*(*Spectrum*), построенный из *Spectrum* = {0, 71, **103**, **113**, **142**, 156, **170**, 174, **250**, 285, 289, 400, **403**, **413**, **432**, 503, 574} с одной недостающей массой (418) и восемью ложными массами (показаны зеленым цветом) по сравнению с идеальным спектром REDCA

Упражнение. Перечислите все пептиды, обозначенные путями от *source* до *sink*, для DAG *Graph*(*Spectrum*), показанного на рисунке выше, и подсчитайте, сколько масс в спектре объясняет каждый из идентифицированных пептидов. Объясняет ли какой-либо из этих пептидов спектр лучше, чем REDCA?

Проблема ложных и отсутствующих масс – лишь одна из многих сложностей в масс-спектрометрии. Когда масс-спектрометр разрушает пептид, небольшие части полученных фрагментов могут быть потеряны, что снижает массу фрагмента. Например, при разбиении REDCA на RE и DCA RE может потерять

молекулу воды (H_2O) с массой 1 + 1 + 16 = 18, а DCA может потерять молекулу аммиака (NH_3) с массой 1 + 1 + 1 + 14 = 17. Соответствующие целые массы полученных фрагментов будут равны *Mass*(RE) − 18 и *Mass*(DCA) − 17.

Из-за многих практических сложностей масс-спектрометрии нам нужно будет сделать некоторые упрощающие предположения, чтобы перейти к вычислительной задаче, моделирующей секвенирование пептидов. Вместо того чтобы пытаться объяснить большое разнообразие различных паттернов фрагментации при реконструкции пептида, мы будем рассматривать их как шум. Мы также будем считать, что все пики соответствуют заряду +1 и что спектры дискретизированы (т. е. все массы являются целыми числами).

Секвенирование пептидов

Определение пептидов по спектрам

Рассмотрим воображаемый мир, в котором пептиды состоят всего из двух аминокислот, X и Z, имеющих массы соответственно 4 и 5. Например, для пептида XZZXX префиксы имеют массы 4, 9, 14, 18 и 22, а суффиксы – массы 22, 18, 13, 8, 4.

Теперь рассмотрим гипотетический спектр

{0, 0, 0, 3, 8, 7, 2, 1, 100, 0, 1, 4, 3, 500, 2, 1, 3, 9, 1, 2, 2, 0},

возникающий из этого пептида, где *i*-й элемент этого вектора соответствует интенсивности, обнаруженной при массе *i*. Префиксы XZZXX аннотируют пики с интенсивностью 3, 100, 500, 9 и 0, тогда как суффиксы аннотируют пики с интенсивностью 3, 1, 3, 9 и 0. Наша цель – разработать метод для определения пептидов по спектрам в надежде, что мы сможем найти пептид, который генерирует спектр, просто найдя пептид с максимальным уровнем в этом спектре.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы определили пептид по спектру?

Одним из методов оценки является **сложение интенсивностей**, или сумма интенсивностей, всех пиков, аннотированных пептидом. Например, мы оценили бы пептид XZZXX в приведенном выше спектре как сумму интенсивностей всех пиков, обозначенных XZZXX, т. е. 3 + 100 + 500 + 9 + 0 + 0 + 8 + 0 + 2 + 1. Однако сложение интенсивности не работает на практике, потому что интенсивность пиков сильно различается. В результате самые высокие пики в спектре (100 и 500 в нашем примере) могут доминировать в сумме. Поскольку самые высокие пики могут представлять собой шум, а пики более низкой интенсивности часто представляют правильные пептиды префикса/суффикса, сложение интенсивности на практике не является хорошей функцией оценки.

Другой подход к счету, называемый **сложение общих пиков**, или **SPC**, просто подсчитывает количество «высоких» пиков, аннотированных пептидом, т. е. аннотированных пиков с интенсивностью, превышающей заранее определенный порог. (Два пика считаются общими, если их массы находятся в пределах ε (по умолчанию используется 0,02 Да для спектров с высоким разрешением). – *Прим. ред.*) Принимая пороговое значение интенсивности 5 в нашем текущем примере, SPC равно 4, поскольку префиксы XZZXX аннотируют высокие пики с интенсивностью 100, 500 и 9, а суффиксы аннотируют высокий пик с интенсивностью 8. Для первого пептида-кандидата *DinosaurSpectrum* (рис. 11.8) SPC равно 10, тогда как второй пептид-кандидат имеет SPC, равное 6.

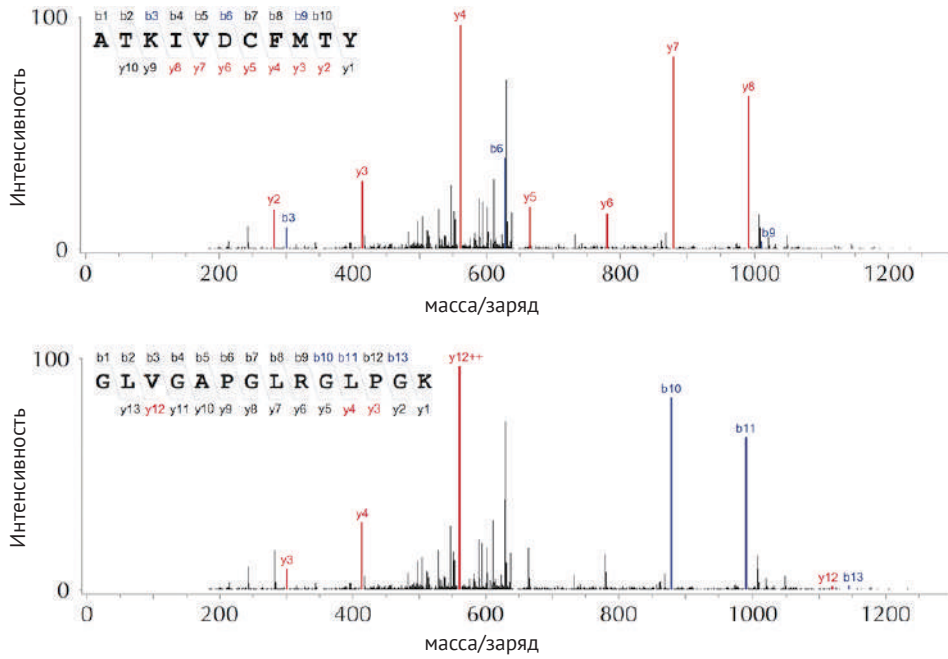


Рис. 11.8 Сложение общих пиков

Хотя SPC на практике работает лучше, чем сложение интенсивностей, он все еще далек от идеала; лучший метод будет учитывать интенсивность пиков, не позволяя самым высоким пикам доминировать в сумме. Для достижения этой цели мы преобразуем пептиды и спектры в векторы, а затем определим оценочную функцию, которая представляет собой скалярное произведение этих векторов.

Для начала нам дана аминокислотная строка $Peptide = a_1 \ldots a_n$ длины n, и мы будем представлять массы ее префикса, используя бинарный **пептидный вектор** *Peptide*' с координатами $Mass(Peptide)$. Этот вектор содержит 1 в каждой из n **координат префикса**

$$Mass(a_1), Mass(a_1a_2), \ldots, Mass(a_1a_2 \ldots a_n),$$

который содержит 0 в каждой из оставшихся координат шума. Пептид XZZXX, префиксные массы которого равны 4, 9, 14, 18 и 22, соответствует пептидному вектору (0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1) длиной 22 (рисунок ниже).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
пептидный вектор	0	0	0	**1**	0	0	0	0	**1**	0	0	0	0	**1**	0	0	0	**1**	0	0	0	**1**
спектральный вектор	0	0	0	**4**	-2	-3	-1	-7	**6**	**5**	3	2	1	**9**	3	-8	0	3	1	2	1	0

Рис. 11.9 Пептидный вектор XZZXX и гипотетический спектральный вектор, созданный этим пептидом (при условии, что X и Z имеют массы 4 и 5 соответственно). Координаты префикса пептидного вектора выделены жирным шрифтом. Амплитуды, превышающие порог 3 в спектральном векторе, показаны цветом. Эти записи, выделенные жирным шрифтом, соответствуют трем координатам префикса (синие) и одной координате шума (красные). Координаты префикса соответствуют единицам в пептидном векторе, тогда как координаты шума соответствуют нулям

Задача преобразования пептида в пептидный вектор:
преобразуйте пептид в пептидный вектор.

Input: строка аминокислоты *Peptide*.

Output: пептидный вектор *Peptide*'.

☞ **Загрузить данные 11.1**

Поскольку пептидный вектор однозначно определяет пептид, от которого он произошел, мы будем использовать термины «пептидный вектор» и «пептид» как синонимы.

Задача преобразование пептидного вектора в пептид:
преобразуйте пептидный вектор в пептид.

Input: бинарный вектор *P*.

Output: пептид, пептидный вектор которого равен *P* (если такой пептид существует).

☞ **Загрузить данные 11.2**

Где находятся суффиксные пептиды?

Вам может быть интересно, почему пептидные векторы моделируют только префиксные пептиды, поскольку и префиксные, и суффиксные пептиды вно-

сят вклад в спектральные аннотации. Не имеет ли больше смысла определять пептидный вектор XZZXX как (0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1 , 0, 0, 0, 1), чтобы отразить как его префиксные массы (4, 9, 14, 18 и 22), так и суффиксные массы (4, 8, 13, 18, 22)?

Действительно, любой пик массы *s* в спектре может быть интерпретирован либо как префиксная масса, либо как суффиксная масса неизвестного пептида *Peptide*, сгенерировавшего спектр. Более того, его **двойной пик** с массой, равной *Mass*(*Peptide*) – *s*, можно интерпретировать либо как массу суффикса, либо как массу префикса одного и того же пептида.

Чтобы справиться с этой неопределенностью, масс-спектрометристы преобразуют спектр *Spectrum* в **спектральный вектор** *Spectrum'*, который объединяет информацию об интенсивности каждого пика и его близнеца в одно значение, называемое **амплитудой**, в координате, представляющей массу гипотетического префиксного пептида для этих близнецов. Почему? Потому что алгоритмы, интерпретирующие объединенный спектр, становятся проще, чем алгоритмы, пытающиеся учесть обоих близнецов (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Задача антисимметричного пути**). Еще больше усложняет ситуацию то, что амплитуды спектральных векторов учитывают также интенсивности фрагментов ионов с различным зарядом и интенсивности фрагментов ионов с потерей молекул воды и аммиака.

На рис. 11.10 представлен спектральный вектор, построенный из *Dinosaur-Spectrum*, и показано, что амплитуды в спектральном векторе могут быть отрицательными. Отрицательные амплитуды обычно соответствуют позициям в спектрах без пиков или с пиками низкой интенсивности. Соответствие между интенсивностями в спектре и амплитудами в его спектральном векторе сложное, но в целом амплитуда при массе *i* отражает *вероятность* того, что (неизвестный) пептид, породивший спектр, имеет префикс с массой *i* (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Преобразование спектров в спектральные векторы**).

После преобразования пептида *Peptide* в пептидный вектор $Peptide' = (p_1, \ldots, p_m)$ и спектра *Spectrum* в спектральный вектор $Spectrum' = (s_1, \ldots, s_m)$ той же длины мы определяем $Score(Peptide, Spectrum) = Score(Peptide', Spectrum')$ как скалярное произведение *Peptide'* и *Spectrum'*,

$$Score(Peptide, Spectrum) = p_1 \cdot s_1 + \ldots + p_m \cdot s_m.$$

Обратите внимание, что *Score*(*Peptide*, *Spectrum*) – это просто «сумма амплитуд», сумма амплитуд в *Spectrum'*, которые «аннотируются» *Peptide'*. Однако эта сумма не страдает от ограничений оценки интенсивности, потому что мы преобразовали интенсивности в спектре в амплитуды в его спектральном векторе. В результате пики высокой интенсивности в спектре вносят вклад в сумму, но не доминируют в ней. Сумма пептидного вектора и спектрального вектора для нашего предыдущего примера (воспроизведенного ниже) составляет 4 + 6 + 9 + 3 + 0 = 22.

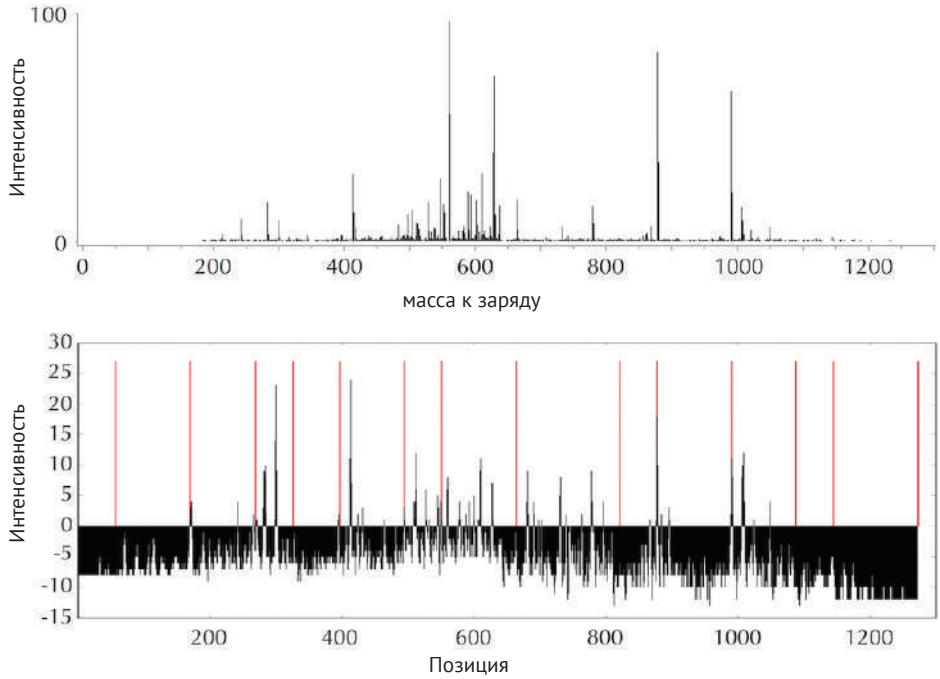


Рис. 11.10 (Вверху) *DinosaurSpectrum*, представленный ранее. (Внизу) Спектральный вектор *DinosaurSpectrum*. Положения единиц в пептидном векторе GLVGAPGLRGLPGK (по координатам 57, 170, 269, 326, 397, 494, 551, 664, 820, 877, 990, 1087, 1144 и 1272) показаны красными линиями. Амплитуды спектрального вектора в этих координатах равны –8, +1, –4, –6, –6, +3, +1, –4, –8, +18, +11, –10, –7 и 0 соответственно

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы найти пептидный вектор, который имеет счет более 22 по сравнению с этим спектральным вектором (рис. 11.9)?

Оценка пептидного вектора и спектрального вектора для нашего примера *T. rex*, *Score*(GLVGAPGLRGLPGK, *DinosaurSpectrum*), составляет –8 + 1 – 4 – 6 – 6 + 3 + 1 – 4 – 8 + 18 + 11 – 10 – 7 + 0 = –19. Поскольку большинство амплитуд отрицательно, тот факт, что *Score*(GLVGAPGLRGLPGK, *DinosaurSpectrum*) является отрицательным, не обязательно означает, что приведенная выше спектральная интерпретация неверна.

В оставшейся части этой главы мы будем работать со спектральными векторами вместо спектров. Наша цель состоит в том, чтоб, имея спектральный вектор *Spectrum′*, найти пептид *Peptide*, максимизирующий *Score*(*Peptide′*, *Spectrum′*). Поскольку масса пептида и исходная масса спектра, который он генерирует, должны быть одинаковыми, пептидный вектор должен иметь ту же

длину, что и рассматриваемый спектральный вектор. Поэтому мы определим разницу между пептидным вектором и спектральным вектором разной длины как $-\infty$.

Задача секвенирования пептидов: *по заданному спектральному вектору найти пептид с максимальным счетом в этом спектре.*

Input: спектральный вектор *Spectrum′*.

Output: строка аминокислот *Peptide*, которая максимизирует *Score*(*Peptide′*, *Spectrum′*) среди всех возможных строк аминокислот.

Алгоритм секвенирования пептидов

Для заданного спектрального вектора $Spectrum' = (s_1, \ldots, s_m)$ мы построим DAG на $m + 1$ узлах, пронумерованных целыми числами от 0 (*source*) до m (*sink*), а затем соединим узел i с узлом j направленным ребром, если $j - i$ равно массе аминокислоты (рисунок ниже). Далее мы присвоим вес s_i узлу i (при $1 \le i \le m$) и нулевой вес узлу 0.

ОСТАНОВИТЕСЬ и задумайтесь. Как этот DAG соотносится с графом DAG *Graph*(*Spectrum*), который мы построили для декодирования идеального спектра?

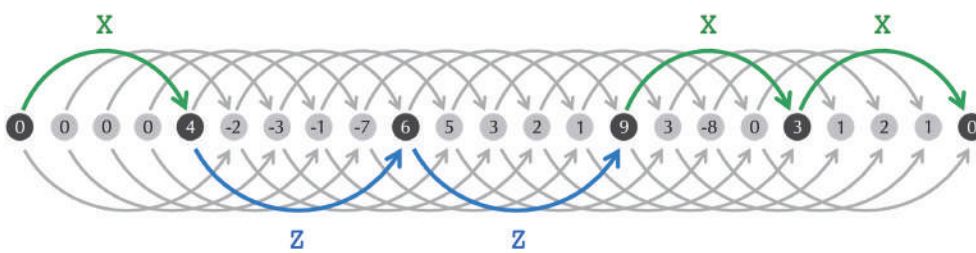


Рис. 11.11 Граф DAG, взвешенный по узлам, для спектрального вектора длины $m = 22$ и алфавита аминокислот {**X**, **Z**} с соответствующими массами 4 и 5. Путь от 0 до m, представляющий пептид **XZZXX** (соответствует пептидному вектору (0, 0, 0, **1**, 0, 0, 0, 0, **1**, 0, 0, 0, 0, **1**, 0, 0, 0, **1**, 0, 0, 0, **1**) со счетом 0 + 4 + 6 + 9 + 3 + 0 выделены. Метки узлов DAG представляют спектральные векторные амплитуды

Любой путь, соединяющий *source* с *sink* в этом DAG, соответствует аминокислотной цепи *Peptide*, а общий вес узлов на этом пути равен *Score*(*Peptide′*, *Spectrum′*). Таким образом, мы свели задачу секвенирования пептидов к задаче поиска пути максимального веса от источника к стоку в DAG со взвешиванием узлов.

ОСТАНОВИТЕСЬ и задумайтесь. Когда мы впервые обсуждали динамическое программирование в контексте выравнивания последовательностей, мы разработали алгоритм для поиска пути максимального веса в DAG со *взвешиванием ребер*. Как мы можем изменить этот алгоритм, чтобы найти путь максимального веса в графе DAG с *невзвешенными узлами*?

Упражнение. Примените свой алгоритм для задачи определения последовательности пептидов к *DinosaurSpectrum*, чтобы найти пептид с наивысшим счетом.

Применяя алгоритм, решающий задачу секвенирования пептидов к *DinosaurSpectrum′*, мы находим пептид ATKIVDCFMTY со счетом 96 (соответствующий первому пептиду-кандидату *T. rex* на рис. 11.12). Однако Asara предложил другой пептид, GLVGAPGLRGLPGK со счетом –19, который мы назовем *DinosaurPeptide* (второй пептид-кандидат *T. rex*). Этот пептид имеет гораздо более низкий счет, чем ATKIVDCFMTY; на самом деле миллиарды пептидов превосходят *DinosaurPeptide*!

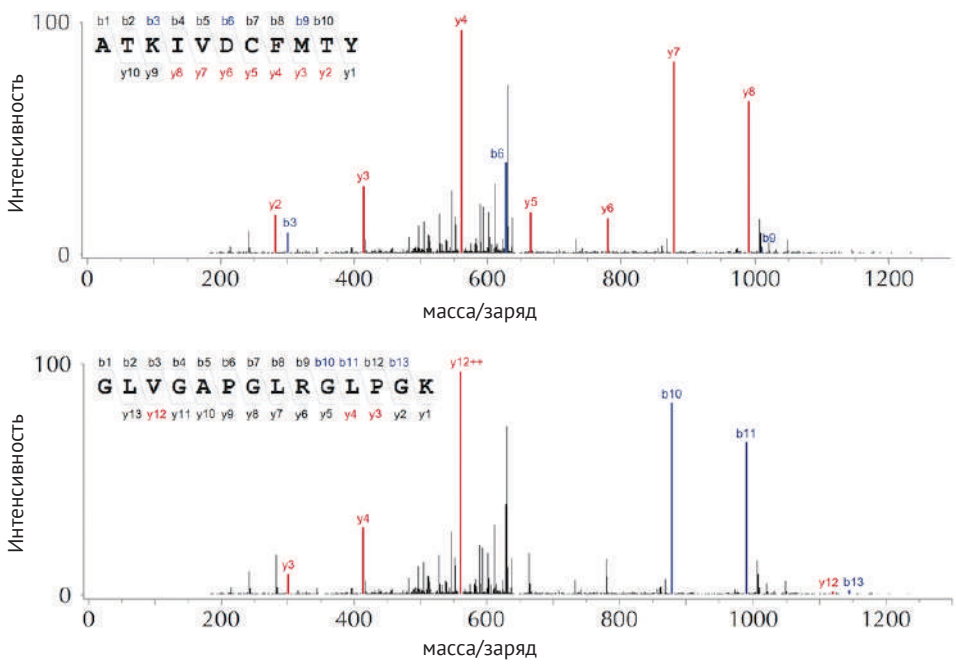


Рис. 11.12 Пептиды-кандидаты *T. rex* (повторение рис. 11.8)

ОСТАНОВИТЕСЬ и задумайтесь. Как вы думаете, почему Asara предложила DinosaurPeptide вместо более результативного ATKIVDCFMTY?

Идентификация пептидов

Задача идентификации пептидов

Если вы следили за нашими усилиями по секвенированию пептидов-антибиотиков, то вы согласитесь с тем, что нам следует опасаться поспешных выводов о том, что пептид с наивысшим уровнем для *DinosaurSpectrum* должен был сгенерировать этот спектр.

Несмотря на многочисленные попытки, исследователи до сих пор не разработали оценочную функцию, которая надежно присваивала бы наивысший балл биологически правильному пептиду, т. е. пептиду, генерирующему спектр. К счастью, хотя правильный пептид часто не достигает наивысшего счета среди *всех* пептидов, он обычно набирает самый высокий уровень среди всех пептидов, *ограниченных* протеомом вида. В результате мы можем перейти от секвенирования пептидов к идентификации пептидов, ограничив наш поиск пептидами, присутствующими в протеоме, которые мы объединяем в *Proteome* из одной цепи аминокислот.

Упражнение. Как соотносится количество всех пептидов длины 10 (которые мы должны исследовать при секвенировании пептидов) с количеством пептидов длины 10 в протеоме человека? (Примечание: в человеческом белке примерно 20 000 кодирующих белок генов, а средняя длина человеческого белка составляет примерно 400 аминокислот.)

Задача идентификации пептида: *найти пептид из протеома с максимальным количеством уровнем в спектре.*

Input: спектральный вектор *Spectrum′* и аминокислотная строка *Proteome*.

Output: аминокислотная строка *Peptide*, которая максимизирует *Score*(*Peptide′*, *Spectrum′*) среди всех подстрок *Proteome*.

☞ **Загрузить данные 11.3**

ОСТАНОВИТЕСЬ и задумайтесь. На практике входными данными для задачи идентификации пептидов является набор белков, а не одна цепь протеома. Каковы потенциальные ловушки при объединении всех белков по сравнению с анализом каждого белка по отдельности?

Идентификация пептидов в неизвестном протеоме T. rex

Вам может быть интересно, почему мы вернулись к идентификации пептидов, ведь мы не знаем протеом *T. rex*. Поэтому может показаться, что мы не можем применить алгоритм для задачи идентификации пептидов к спектрам, полученным из окаменелостей тираннозавра.

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем создать базу данных белков для поиска пептидов *T. rex*?

Примерно 90 % белков, из которых состоят кости животных, составляет **коллаген**. Кости динозавров, несомненно, содержали коллаген, и маловероятно, что другие белки могли сохраниться в течение миллионов лет. Поскольку аминокислотные последовательности коллагенов консервативны и часто совпадают у разных видов, Асара пришел к выводу, что любые белки, сохранившиеся в окаменелостях тираннозавра, вероятно, будут похожи на коллагены современных видов.

В качестве проверки Асара сравнил спектры *T. rex* со всей базой данных **UniProt**, содержащей белки сотен видов и в общей сложности почти 200 млн аминокислот. Он также включил некоторые мутировавшие версии коллагенов современных видов, чтобы смоделировать возможные различия между этими коллагенами и коллагенами тираннозавра (мы назовем полученную базу данных белков **UniProt+**). Оказалось, что большинство пептидов с высокими оценками, идентифицированных в этой базе данных, были куриными коллагенами, что подтверждает гипотезу о том, что птицы произошли от динозавров.

На самом деле *DinosaurPeptide* – это всего лишь одна мутация, не считая пептида куриного коллагена. Но как мы можем проверить, является ли этот пептид правильной интерпретацией *DinosaurSpectrum*?

Поиск совпадений пептидов со спектром

Подобно алгоритмам секвенирования пептидов, алгоритмы идентификации пептидов могут выдавать ошибочный пептид, особенно если показатель пептида с наивысшим счетом, обнаруженным в протеоме, намного ниже, чем его счет среди всех пептидов. По этой причине биологи обычно устанавливают

порог уровня и обращают внимание на решение задачи идентификации пептидов только в том случае, если его показатель по крайней мере равен порогу.

Имея набор спектральных векторов *SpectralVectors*, аминокислотную строку *Proteome* и пороговое значение уровня *threshold*, мы решим задачу идентификации пептида для каждого вектора *Spectrum′* в *SpectralVectors* и идентифицируем пептид *Peptide*, имеющий максимальную величину для этого спектрального вектора среди всех пептидов в *Proteome* (связи разорваны произвольно). Если *Score*(*Peptide′*, *Spectrum′*) больше или равен порогу *threshold*, то мы делаем вывод, что пептид присутствует в образце, и называем пару (*Peptide, Spectrum′*) **пептид-спектр совпадением (PSM)**. Результирующий набор этих PSM для *SpectralVectors* обозначается как $PSM_{threshold}$(*Proteome*, *SpectralVectors*).

Задача поиска PSM: *идентифицируйте все пептид-спектр-совпадения, превышающие пороговое значение для набора спектров и протеома.*

Input: набор спектральных векторов *SpectralVectors*, аминокислотная строка *Proteome* и целочисленный *threshold*.

Output: набор $PSM_{threshold}$(*Proteome*, *SpectralVectors*).

Следующий псевдокод решает задачу поиска PSM, используя алгоритм, который вы только что реализовали для решения задачи идентификации пептидов, который мы назвали **PeptideIdentification**.

```
PSMSearch(SpectralVectors, Proteome, threshold)
  PSMSet ← пустой набор
  for каждого вектора Spectrum′ в SpectralVectors
    Peptide ← PeptideIdentification(Spectrum′, Proteome)
    if Score(Peptide, Spectrum′) ⩾ threshold
      добавить PSM (Peptide, Spectrum′) к PSMSet
  return PSMSet
```

DinosaurPeptide оказался пептидом с наивысшим показателем среди всех пептидов в базе данных UniProt+. Но означают ли миллиарды пептидов, не встречающихся в этой базе данных, которые превосходят *DinosaurPeptide*, что база данных, сформированная Асарой, является неполной и что *DinosaurSpectrum* возник из другого пептида?

Реальность такова, что пептид с наивысшим показателем в протеоме обычно уступает миллиардам пептидов, не принадлежащих к протеому. Однако этот факт не означает, что **PSMSearch** идентифицировал неправильный пептид, потому что общее количество пептидов с одинаковой массой может измеряться триллионами или даже квадриллионами. Другими словами, миллиарды

пептидов, которые превосходят *DinosaurPeptide*, представляют собой небольшую долю всех пептидов, имеющих такую же массу, как этот пептид. Таким образом, нам необходимо дополнить **PSMSearch** величиной статистической значимости идентифицированных им PSM.

ОСТАНОВИТЕСЬ и задумайтесь. Предположим, что мы сравниваем 1000 спектров из образца ткани курицы с протеомом курицы и идентифицируем 100 PSM, чьи показатели превышают пороговый уровень. Как бы вы оценили процент ошибочных PSM среди этих 100 PSM?

Идентификация пептидов и теорема о бесконечных обезьянах

Частота ложных открытий

Чтобы посчитать количество ложных PSM в $PSM_{threshold}(Proteome, SpectralVectors)$, мы построим **протеом-приманку** *DecoyProteome*, случайно сгенерированную аминокислотную строку, имеющую ту же длину, что и *Proteome* (с вероятностью генерации любой аминокислоты в каждой позиции, равной 1/20). Затем мы решим задачу поиска PSM для *DecoyProteome* вместо *Proteome* для того же порогового уровня.

Нас не интересуют PSM, идентифицированные в случайно сгенерированной протеоме-приманке, которые являются не чем иным, как биологически нерелевантными артефактами. Дело в том, что количество этих PSM даст приблизительное количество ошибочных PSM, выявленных в нашем биологически релевантном поиске по сравнению с реальным протеомом. Поэтому мы определим **частоту ложных открытий (FDR)** поиска PSM как отношение количества ложных PSM к количеству PSM, идентифицированных по отношению к реальному протеому,

$$|\text{PSM}_{threshold}(DecoyProteome, SpectralVectors)| / |\text{PSM}_{\text{threshold}}(Proteome, SpectralVectors)|.$$

Например, если поиск по *Proteome* дает 100 PSM, а поиск по *DecoyProteome* дает всего пять PSM, то FDR будет равен 5 %, и мы придем к выводу, что примерно 95 % идентифицированных PSM, вероятно, идентифицированы правильно. С другой стороны, если бы поиск *DecoyProteome* дал около 100 PSM, тогда FDR был бы близок к 1, и нам было бы трудно аргументировать, что любые пептиды, идентифицированные в нашем поиске по *Proteome*, биологически релевантны.

Упражнение. Определите FDR при поиске всех спектров *T. rex* в базе данных UniProt+ с *threshold* = 80.

ОСТАНОВИТЕСЬ и задумайтесь. Если FDR окажется очень высоким для данного значения порога, сможем ли мы найти надежные PSM?

Даже если FDR высок, мы не должны делать вывод, что наш набор спектральных данных бесполезен или что мы ищем не в той базе данных белков. Возможно, мы просто выбрали неправильный порог счета для анализа наших данных, поскольку FDR может сильно варьироваться в зависимости от выбора этого порога (рис. 11.13).

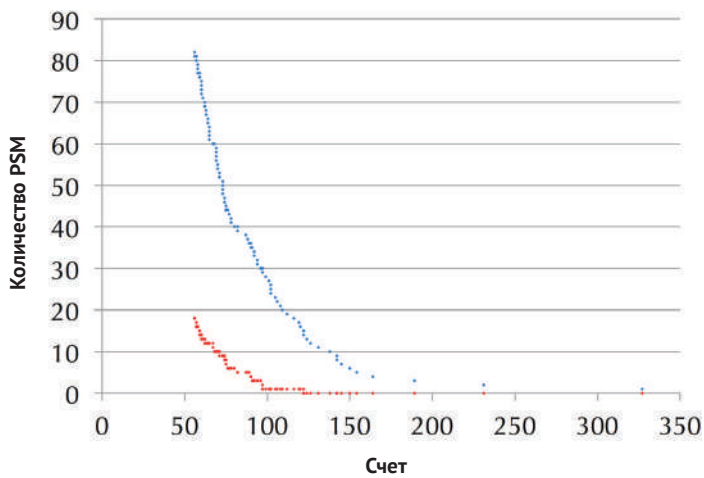


Рис. 11.13 Количество PSM, идентифицированных путем поиска *DinosaurSpectrum* по аминокислотной строке, полученной из базы данных UniProt+ (синий цвет), и протеоме-приманке той же длины (красный цвет) в зависимости от порога счета *threshold*

Для набора спектральных данных *T. rex* мы находим 27 PSM в аминокислотной цепи *Proteome*, полученной из UniProt+, и только один PSM в *DecoyProteome* со счетом, по крайней мере, равным *threshold* = 100 (FDR = 3,7 %). К сожалению, мы не можем автоматически заключить, что обнаружили две дюжины пептидов динозавров, потому что многие из этих PSM соответствуют обычным лабораторным загрязнителям. Наша цель – выяснить, действительно ли оставшиеся несколько, включая *DinosaurPeptide*, соответствуют пептидам *T. rex*.

В частности, FDR помогает нам анализировать весь набор идентифицированных PSM для всех спектров *T. rex*, но как насчет статистической значимости отдельного PSM? В частности, можем ли мы определить, является ли PSM

(*DinosaurPeptide, DinosaurSpectrum*′), который мы будем называть *DinosaurPSM*, статистически значимым? Чтобы ответить на этот вопрос, сначала нужно количественно определить, что мы подразумеваем под «статистически значимым».

Представьте, что мы заперли вас в комнате с обезьяной и пишущей машинкой. Обезьяна быстро устает от вашей компании и начинает стучать по пишущей машинке, создавая цепи символов (предположим, что обезьяне особенно нравится пробел). Когда вы начинаете терять рассудок, вы проверяете каждую новую строку, сгенерированную обезьяной, чтобы убедиться, что некоторые из них правильно написаны английскими словами (рис. 11.4). В конце концов, согласно теореме о бесконечных обезьянах, обезьяна напечатает Гамлета (см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Теорема о бесконечных обезьянах**).

Вы были бы в шоке, если бы обезьяна тут же напечатала «Быть или не быть». Но были бы вы так же удивлены, если бы обезьяна, набрав миллион строк, напечатала дюжину английских слов?

Рис. 11.14 В поисках Уильяма Шекспира

Для заданного набора строк *Dictionary* мы определяем *E*(*Dictionary, n*) как ожидаемое количество вхождений строк из *Dictionary* в случайно сгенерированную строку длины *n*, где вероятности генерации каждой буквы в каждой позиции строки одинаковы. Пусть *EnglishDictionary* представляет собой множество всех английских слов. Если окажется, что после ввода *n* символов обезьяна набирает значительно больше английских слов, чем *E*(*EnglishDictionary, n*), то у нас есть все основания полагать, что обезьяна умеет писать! С другой стороны, если обезьяна набирает примерно то же количество слов, что и *E*(*EnglishDictionary, n*), то эта обезьяна, скорее всего, не является реинкарнацией Шекспира.

Задача «Обезьяна и пишущая машинка»: *найдите ожидаемое количество строк из словаря, встречающихся в случайно сгенерированном тексте.*

Input: набор строк *Dictionary* и целое число *n*.

Output: *E*(*EnglishDictionary, n*).

ОСТАНОВИТЕСЬ и задумайтесь. Какое отношение обезьяна и пишущая машинка имеют к масс-спектрометрии?

Упражнение. Каково ожидаемое количество раз, когда строка SHAKESPEARE встречается в случайно сгенерированной строке на английском языке (без пробелов) длиной 200 млн?

Статистическая значимость пептид-спектр-совпадений

Теперь представьте, что вместо обезьяны, печатающей слова, у нас есть алгоритм, генерирующий набор всех пептидов, набравших по крайней мере пороговое значение *threshold* по отношению к спектральному вектору *Spectrum′*. В дальнейшем мы будем называть этот набор пептидов с высокими показателями **спектральным словарем**, обозначаемым

$$Dictionary_{threshold}(Spectrum').$$

Для PSM (*Peptide, Spectrum′*) мы будем использовать термин **PSM-словарь**, обозначаемый *Dictionary*(*Peptide, Spectrum′*), для обозначения спектрального словаря.

$$Dictionary_{Score(Peptide,Spectrum')}(Spectrum').$$

Для *DinosaurPSM* используется PSM-словарь $Dictionary'_{-19}(DinosaurSpectrum')$.

Вместо проверки того, какие слова, сгенерированные обезьяной, встречаются в английском словаре, мы сопоставим пептиды из спектрального словаря с протеомом. Если мы находим совпадения, то должны решить, представляют ли эти совпадения биологически достоверные PSM или просто статистические артефакты. Для принятия этого решения рассмотрим

$$E(Dictionary_{threshold}(Spectrum', n)),$$

ожидаемое количество пептидов в протеоме-приманке длины n, которое будет встречаться в *Dictionarythreshold*(*Spectrum′*). Если это число больше 1, то нет ничего удивительного в обнаружении пептида, который имеет пороговое значение *threshold* пр сравнению со *Spectrum′*. Поэтому мы сформулировали наш тест статистической значимости как частный случай задачи «Обезьяна и пишущая машинка».

Задача определения ожидаемого количества пептидов с высоким значением: *найти ожидаемое количество пептидов с высоким значением по заданному спектру в протеоме-ловушке.*

Input: спектральный вектор *Spectrum*, целочисленный *threshold* и *n*.
Output: $E(Dictionary_{threshold}(Spectrum', n))$.

Чтобы решить эту задачу, мы начнем со спектрального словаря, состоящего из одной строки аминокислот *Peptide*, которую мы попытаемся сопоставить со случайно сгенерированной строкой *DecoyProteome* длины *n*. Поскольку *DecoyProteome* был сгенерирован случайным образом, вероятность того, что *Peptide* соответствует строке, начинающейся в данной позиции *DecoyProteome*, составляет $1/20^{|Peptide|}$. Мы называем это выражение **вероятностью** *Peptide*. Следовательно, ожидаемое количество раз, когда *Peptide* встречается в *DecoyProteome*, равно

$$\frac{n-|Peptide|+1}{20^{|Peptide|}} \approx n \cdot \frac{1}{20^{|Peptide|}}.$$

Далее предположим, что набор словарей пептидов *Dictionary* содержит несколько строк аминокислот произвольной длины. Используя приведенное выше приближение, ожидаемое количество совпадений между строками в *Dictionary* и *DecoyProteome* можно приблизительно представить как

$$E(Dictionary, n) \approx n \cdot \left(\sum_{\text{each peptide } Peptide \text{ in } Dictionary} \frac{1}{20^{|Peptide|}} \right).$$

Мы будем ссылаться на сумму внутри круглых скобок выше как **вероятность** *Dictionary*, обозначаемую *Pr*(*Dictionary*), так что предыдущее приближение может быть записано как

$$E(Dictionary, n) \approx n \cdot Pr(Dictionary).$$

Таким образом, мы свели статистический анализ *PSM*(*Peptide, Spectrum'*) к вычислению *Pr*(*Dictionary*(*Peptide, Spectrum'*)), вероятности словаря PSM.

ОСТАНОВИТЕСЬ и задумайтесь. Может ли *Pr*(*Dictionary*(*Peptide, Spectrum'*)) быть больше 1?

Вам может быть интересно, почему мы использовали вероятностное обозначение *Pr*(*Dictionary*). Чтобы узнать, почему *Pr*(*Dictionary*(*Peptide, Spectrum'*)) действительно является вероятностью (и, следовательно, не может превышать 1), см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Вероятностное пространство пептидов в спектральном словаре**.

Задача вероятности спектрального словаря: *найти вероятность спектрального словаря для заданного спектра и порога счета.*

Input: спектральный вектор *Spectrum* и целочисленный *threshold*.

Output: вероятность $Dictionary_{threshold}(Spectrum')$.

Кажется, мы наконец-то готовы проверить статистическую значимость *DinosaurPSM*. Для начала нам нужно создать PSM-словарь *Dictionary*(*DinosaurPSM*), а затем вычислить $n \cdot Pr(Dictionary(DinosaurPSM))$, где n – длина строки, образованной путем объединения базы данных UniProt+. Если это значение мало (скажем, 0,001), то мы можем утверждать, что *DinosaurPeptide* является пептидом *T. rex*, а не статистическим артефактом.

К сожалению, *Dictionary*(*DinosaurPSM*) содержит более 200 млрд пептидов, и его создание заняло бы очень много времени. Можем ли мы каким-то образом вычислить вероятность этого словаря, не создавая его?

Спектральные словари

Сначала мы вычислим количество пептидов в спектральном словаре, поскольку эта более простая задача даст представление о том, как вычислить вероятность спектрального словаря.

Задача размера спектрального словаря: *определить размер спектрального словаря для заданного спектрального вектора и порогового уровня.*

Input: спектральный вектор *Spectrum*′ и целочисленный пороговый уровень *threshold*.

Output: количество пептидов в $Dictionary_{threshold}(Spectrum')$.

Мы будем использовать динамическое программирование для решения задачи размера спектрального словаря. Для заданного спектрального вектора $Spectrum' = (s_1, \ldots, s_m)$ мы определяем его ***i*-префикс** (для i между 1 и m) как $Spectrum_i = (s_1, \ldots, s_m)$ и вводим переменную $Size(i, t)$ как количество пептидов. Пептид массы i такой, что $Score(Peptide, Spectrum'_i)$ равен t. Например, рассмотрим спектральный вектор $Spectrum' = (4, -3, -2, 3, 3, -4, 5, -3, -1, -1, 3, 4, 1, 3)$ длины 14 и мини-алфавит аминокислот, состоящий из аминокислот X и Z с массами 4 и 5 соответственно. Имеется только три пептида с массой 13 (XXZ, XZX и ZXX); первые два пептида имеют значение 1 по сравнению со $Spectrum'_{13}$, а третий имеет значение 3. Таким образом, $Size(13, 1) = 2$, $Size(13, 3) = 1$ и $Size(13, t) = 0$ для всех значения t, отличных от 1 и 3.

Ключом к установлению рекуррентного соотношения для вычисления *Size*(*i, t*) является понимание того, что набор пептидов, влияющих на *Size*(*i, t*), может быть разделен на 20 подмножеств в зависимости от их конечной аминокислоты *a*. Каждый пептид, оканчивающийся на определенную аминокислоту *a*, дает более короткий пептид с массой $i - |a|$ и счетом $t - s_i$, если мы удалим *a* из пептида (здесь $|a|$ обозначает массу *a*). Таким образом, как показано на рис. 11.15,

$$Size(i,t) = \sum_{\text{all amino acids } a} Size(i - |a|, t - s_i).$$

Поскольку существует единственный «пустой» пептид нулевой длины, мы инициализируем *Size*(0, 0) = 1. Мы также определяем *Size*(0, t) = 0 для всех возможных значений *t* и устанавливаем *Size*(*i, t*) = 0 для отрицательных значений *i*. Используя приведенную выше рекуррентность, мы можем вычислить размер спектрального словаря $Spectrum_i = (s_1, \ldots, s_m)$ как

$$|Dictionary_{threshold}(\overrightarrow{Spectrum})| = \sum_{t \geq threshold} Size(m,t).$$

ОСТАНОВИТЕСЬ и задумайтесь. Приведенная выше формула требует вычисления *Size*(*m, t*) для всех значений *t*, превышающих *threshold*. Имея спектральный вектор *Spectrum′*, можете ли вы найти значение *T* такое, что *Size*(*m, t*) равен нулю при $t > T$?

Упражнение. Вычислите |*Dictionary*(*DinosaurPSM*)|.

Обратите внимание, что уравнение *вероятности* словаря

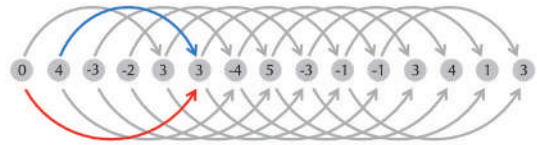
$$Pr(Dictionary) = \sum_{\text{each peptide } Peptide \text{ in } Dictionary} \frac{1}{20^{|Peptide|}}$$

похоже на уравнение размера словаря,

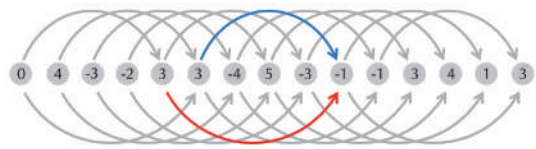
$$|Dictionary| = \sum_{\text{each peptide } Peptide \text{ in } Dictionary} 1.$$

Это сходство предполагает, что мы можем получить рекуррентность для вероятности словаря, используя аргументы, аналогичные тем, которые используются для определения размера словаря.

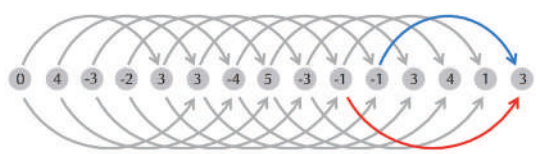
Определим *Pr*(*i, t*) как сумму вероятностей всех пептидов с массой *i*, для которых *Score*(*Peptide′*, $Spectrum'_i$) равен *t*. Набор пептидов, влияющих на *Pr*(*i, t*), можно разделить на 20 подмножеств в зависимости от их конечной амино-



t\i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	**1**	**0**	0	0	0	0									
1	0	0	0	0	0	0									
2	0	0	0	0	0	0									
3	0	0	0	0	1	**1**									
4	0	0	0	0	0	0									
5	0	0	0	0	0	0									



t\i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	0	0	0	0	0	0	1	0					
1	0	0	0	0	0	0	0	0	0	0					
2	0	0	0	0	0	0	0	0	0	**2**					
3	0	0	0	0	**1**	**1**	0	0	0	0					
4	0	0	0	0	0	0	0	0	0	0					
5	0	0	0	0	0	0	0	0	0	0					



t\i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	**2**	**1**	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0	2	0
4	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**

Рис. 11.15 Вычисление *Size*(*i, t*) для мини-алфавита, состоящего из аминокислот X и Z с соответствующими массами 4 и 5 и спектрального вектора (4, –3, –2, 3, 3, –4, 5, – 3, –1, –1, 3, 4, 1, 3). Выделенные жирным шрифтом черные записи в матрицах динамического программирования вычисляются путем суммирования синих и красных записей по формуле *Size*(*i, t*) = *Size*(i – 4, t – s_i) + *Size*(i – 5, t – s_i). Например, в матрице внизу *Size*(14, 5) = *Size*(14 – 4, 5 – 3) + *Size*(14 – 5, 5 – 3) = *Size*(10, 2) + *Size*(9, 2)

кислоты. Каждый пептид *Peptide*, оканчивающийся на определенную аминокислоту, приводит к более короткому пептиду $Peptide_a$, если удалить a; $Peptide_a$ имеет массу $i - |a|$ и значение $t - s_i$. Так как вероятность *Peptide* в 20 раз меньше вероятности $Peptide_a$, вклад пептида в $Pr(i, t)$ в 20 раз меньше, чем вклад пептида в $Pr(i - |a|, t - s_i)$. Следовательно, $Pr(i, t)$ можно вычислить как

$$Pr(i,t) = \sum_{\text{all amino acids } a} \frac{1}{20} \cdot Pr(i - |a|, t - s_i),$$

что отличается от рекуррентности для вычисления $Size(i, t)$ только наличием множителя 1/20.

Теперь мы можем вычислить вероятность спектрального словаря как

$$Pr(Dictionary_{threshold}(\overrightarrow{Spectrum})) = \sum_{t \geq threshold} Pr(m,t).$$

В частности, мы находим, что *Dictionary*(*DinosaurPSM*) состоит из 219 136 251 374 пептидов и имеет вероятность 0,00018. Поэтому мы готовы проверить статистическую значимость *DinosaurPSM*, найденного при поиске, в базе данных UniProt+ длиной n = 194 613 142 (включающей 546 799 белков).

Наша цель состоит в том, чтобы вычислить $n \cdot Pr(Dictionary(DinosaurPSM))$, поскольку это приблизительно соответствует количеству пептидов из *Dictionary*(*DinosaurPSM*), которое мы ожидаем найти в протеоме-приманке длины n. Поскольку $Pr(Dictionary(DinosaurPSM)) = 0{,}00018$, мы имеем, что

$$n \cdot Pr(Dictionary(DinosaurPSM)) = 35{,}311.$$

Поэтому мы ожидаем найти десятки тысяч пептидов со значением не ниже *DinosaurPeptide* (по сравнению с *DinosaurSpectrum′*) в базе данных приманок, и поэтому нет ничего удивительного в обнаружении *DinosaurPSM* при поиске в базе данных UniProt+! Следовательно, мы заключаем, что *DinosaurPeptide* является статистическим артефактом, а не реальным пептидом *T. rex*. А как насчет других пептидов тираннозавра?

Упражнение. Вычислить вероятности спектральных словарей для всех других PSM *T. rex*, о которых сообщил Асара. Являются ли эти PSM статистически значимыми?

Пептиды T. rex: постороннее загрязнение или древнее сокровище?

Загадка гемоглобина

Получив критику в отношении статистических оснований своих заявлений, Асара признал некоторые проблемы со своим анализом, отозвал *DinosaurPeptide* как объяснение *DinosaurSpectrum*, изменил некоторые из ранее предложенных им пептидов *T. rex* и опубликовал все 31 372 спектра ископаемых остатков *T. rex*. Впоследствии другие ученые повторно проанализировали все спектры и подтвердили, что, хотя некоторые из первоначально зарегистрированных PSM *T. rex* сомнительны, другие являются статистически надежными (рис. 11.16).

ID	Пептид	Белок	Вероятность	$n \cdot$ Вероятность
P1	GL**V**GAPGLRGLPGK	Collagen α1t2	$1.8 \cdot 10^{-4}$	36 000
P2	GVVGLP$_{oh}$GQR	Collagen α1t1	$7.6 \cdot 10^{-8}$	16
P3	GVQGPP$_{oh}$GPQGPR	Collagen α1t1	$7.9 \cdot 10^{-11}$	$1.6 \cdot 10^{-2}$
P4	GATGAP$_{oh}$GIAGAP$_{oh}$GFP$_{oh}$GAR	Collagen α1t1	$3.2 \cdot 10^{-12}$	$6.4 \cdot 10^{-4}$
P5	GLPGESGAVGPAGPIGSR	Collagen α2t1	$9.9 \cdot 10^{-14}$	$2.0 \cdot 10^{-5}$
P6	GSAGPP$_{oh}$GATGFP$_{oh}$GAAGR	Collagen α1t1	$3.2 \cdot 10^{-14}$	$6.4 \cdot 10^{-6}$
P7	GAPGPQGPSGAP$_{oh}$GP**K**	Collagen α1t1	$7.0 \cdot 10^{-16}$	$1.4 \cdot 10^{-7}$
P8	VNVADCGAEALAR	Hemoglobin β	$7.8 \cdot 10^{-17}$	$1.6 \cdot 10^{-8}$

Рис. 11.16 Семь потенциальных пептидов коллагена *T. rex* (P1 – P7), о которых сообщил Асара, а также пептид гемоглобина (P8). В последнем столбце показаны вероятности словарей PSM, образованных этими пептидами. Красные символы обозначают мутировавшие аминокислоты по сравнению с пептидами в базе данных UniProt. Аминокислота P_{oh} – это гидроксипролин, модифицированная форма пролина, распространенная в коллагенах

Однако публикация Асары спектров *T. rex* вызвала больше вопросов, чем ответов. В этих спектрах Мэтью Фитцгиббон и Мартин Макинтош определили дополнительный спектр (рис. 11.17), который идеально соответствует гемоглобину страуса, таким образом добавив еще один пептид *T. rex* к семи пептидам коллагена из текста выше. PSM гемоглобина, который был пропущен Асарой, на порядок более статистически значим, чем любой ранее описанный коллагеновый пептид *T. rex*!

Было бы шокирующим, если бы пептид гемоглобина действительно принадлежал *T. rex*, потому что гемоглобины гораздо менее консервативны, чем коллагены. Например, гемоглобин человека с бета-цепью имеет длину 146 аминокислот и имеет 27, 38 и 45 различий аминокислот с мышами, кенгуру и курицей соответственно. Кроме того, интактные пептиды гемоглобина никогда не обнаруживались в гораздо более молодых и широко доступных окаменелостях,

таких как кости вымерших пещерных медведей. Эти окаменелости настолько распространены в европейских пещерах, что во время Первой мировой войны их использовали в качестве источника фосфатов для производства пороха.

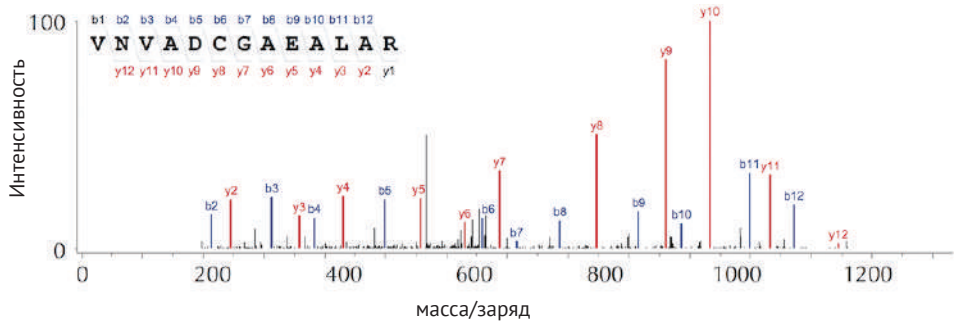


Рис. 11.17 Высококачественный спектр *T. rex*, соответствующий пептиду гемоглобина страуса VNVADCGGAEAIAR. Почти все возможные префиксы и суффиксы представлены пиками высокой интенсивности; фактически применение секвенирования *de novo* к этому спектру приводит к тому же самому пептиду

Поскольку Асара проанализировал образцы страусов до анализа образца тираннозавра, Фитцгиббон и Макинтош утверждали, что пептид гемоглобина может указывать на загрязненный образец в виде **переноса** или ошибочную идентификацию пептидов от предыдущих экспериментов, оставшихся внутри масс-спектрометра после предыдущего эксперимента. Загрязнение является обыденным фактом в каждой протеомной лаборатории, и масс-спектрометристы никогда не удивляются, обнаружив человеческий кератин в своих образцах: воздух в любой комнате обычно содержит миллионы крошечных частиц человеческой кожи.

Если пептид гемоглобина является переносным, то весь образец *T. rex* был загрязнен, что означает, что все другие пептиды *T. rex* следует выбросить. Однако Асара утверждал, что в его эксперименте не было загрязнения и что гемоглобин страуса должен быть пептидом тираннозавра, расширяя класс белков, которые могут сохраняться в течение 68 млн лет, помимо коллагенов.

Тем не менее если окаменелость *T. rex* Хорнера действительно является сокровищницей древних белков и мы признаем, что пептид гемоглобина принадлежит тираннозавру, то почему мы должны ограничивать наши поиски коллагеновыми пептидами и их мутантными вариантами? Почему бы не провести поиск по всем известным белкам всех позвоночных? Конечно, мы должны использовать критерии, аналогичные тем, которые использовал Асара, например допустить наличие одной мутации. Если мы будем следовать этому критерию, то нужно дополнить ранее идентифицированный набор пептидов удивительно разнообразным набором пептидов страуса, курицы, мыши и человека; некоторые из этих пептидов показаны на рисунке ниже.

ID	Пептид	Белок	Вероятность	$n \cdot$ Вероятность
P9	EDCLSG**A**KPK	ATG7 (Chicken)	$3.2 \cdot 10^{-12}$	$6.4 \cdot 10^{-4}$
P10	ENAGEDPGLAR	DCD (Human)	$2.7 \cdot 10^{-12}$	$5.4 \cdot 10^{-4}$
P11	**E**GVDAGAAGDPER	TTL11 (Mouse)	$1.2 \cdot 10^{-12}$	$2.4 \cdot 10^{-2}$
P12	S**W**IHVALVTGGNK	CBR1 (Human)	$1.2 \cdot 10^{-12}$	$2.4 \cdot 10^{-4}$
P13	SSN**V**LSGSTLR	MAMD1 (Human)	$5.9 \cdot 10^{-13}$	$1.8 \cdot 10^{-4}$
P14	DEVTPA**Y**VVVAR	AEPM (Mouse)	$1.9 \cdot 10^{-13}$	$3.8 \cdot 10^{-5}$
P15	**R**NVADCGAEALAR	HBB (Ostrich)	$3.5 \cdot 10^{-15}$	$7.0 \cdot 10^{-7}$

Рис. 11.18 Выравнивание спектров *T. rex* с белками позвоночных в базе данных UniProt (с точностью до одной мутации) выявляет разнообразный набор пептидов. Красные символы обозначают мутировавшие аминокислоты. Обратите внимание на присутствие другого пептида гемоглобина страуса (P15), который немного тяжелее (на 57 Да), чем ранее описанный пептид гемоглобина из предыдущего рисунка. Это изменение массы может представлять собой либо мутацию V в R (как показано выше), либо модификацию аминокислоты

В свете этих новых данных по пептидам заявление Асары о поиске молекулярных доказательств связи между птицами и динозаврами становится еще слабее (подробнее о дебатах вокруг утверждения о том, что птицы произошли от динозавров, см. **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: Действительно ли наземные динозавры являются предками птиц?**). Если бы мы попытались отбросить пептиды на предыдущем рисунке как статистические артефакты, то, возможно, пришлось бы также отбросить и первоначально идентифицированные пептиды *T. rex*.

Споры о ДНК динозавров

Статья «Пептиды *T. rex*» продолжает устаревать, но спорам о ней не видно конца. Тем не менее это была не первая статья, в которой сообщалось об извлечении генетического материала у динозавров. В 1994 году Скотт Вудворд объявил, что секвенировал ДНК кости динозавра возрастом 80 млн лет. Самым яростным критиком его открытия была – хотите верьте, хотите нет – Мэри Швейцер, которая доказала, что Вудворд секвенировал всего лишь загрязнившую образец человеческую ДНК.

Мораль истории заключается в том, что, хотя мы часто представляем научные открытия как ясные и неопровержимые, реальность такова, что некоторые интересные направления современной науки часто не соответствуют этому идеалу. В некотором смысле академическое поле битвы – это часть того, что в первую очередь привлекает молодых людей к карьере ученого. Но мы также не можем не задаться вопросом, получили бы мы окончательный ответ на вопрос, действительно ли окаменелость Хорнера содержала пептиды динозавров, если бы она первоначально была предоставлена десяткам независимых исследователей, которые, несомненно, обнаружили бы шокирующее присутствие гемоглобина в образцах тираннозавра. Соответственно, в своей критике статьи Вудворда о ДНК динозавров Швейцер писал, что «настоящий

прогресс в [палеонтологии] наступит только тогда, когда будет продемонстрировано, что эти исследования могут быть воспроизведены в независимых лабораториях».

Эпилог. От немодифицированных к модифицированным пептидам. (Часть 1)

Посттрансляционные модификации

Алгоритм **PSMSearch** может идентифицировать пептид только в том случае, если он встречается в протеоме без мутаций. Тем не менее некоторые из пептидов в эксперименте с динозаврами мутировали.

ОСТАНОВИТЕСЬ и задумайтесь. Как мы можем использовать **PSMSearch** для поиска мутировавших пептидов (рис. 11.16)?

Чтобы найти пептид с наивысшим значением, содержащий до k мутаций, соответствующих спектральному вектору, мы могли бы сгенерировать все мутировавшие варианты всех пептидов в протеоме, соединить их в строку аминокислот *MutatedProteome*, а затем запустить **PSMSearch** на *MutatedProteome*. К сожалению, количество мутировавших пептидов будет настолько велико, что это может сделать **PSMSearch** практически неприменимым, даже если мы допустим не более одной мутации на пептид.

ОСТАНОВИТЕСЬ и задумайтесь. Сколько мутированных пептидов не более чем с k мутациями существует для данного пептида длины n?

В дополнение к поиску мутировавших пептидов нам также потребуется искать посттрансляционные модификации, которые изменяют аминокислоты после трансляции белка с РНК. Фактически большинство белков модифицируется после трансляции, и были обнаружены сотни типов модификаций. Например, ферментативная активность многих белков регулируется добавлениями или делециями фосфатной группы у определенной аминокислоты (рис. 11.19). Этот процесс, называемый **фосфорилированием**, обратим; **протеинкиназы** добавляют фосфатные группы, тогда как **протеинфосфатазы** удаляют их.

На самом деле, вы, возможно, уже заметили, что большинство пептидов-кандидатов *T. rex* имеет модификацию, превращающую пролин (масса 97) в гидроксипролин (масса 113). Гидроксипролин является основным компонентом коллагена, который важен для стабильности коллагена и составляет примерно 4 % всех аминокислот в организме человека.

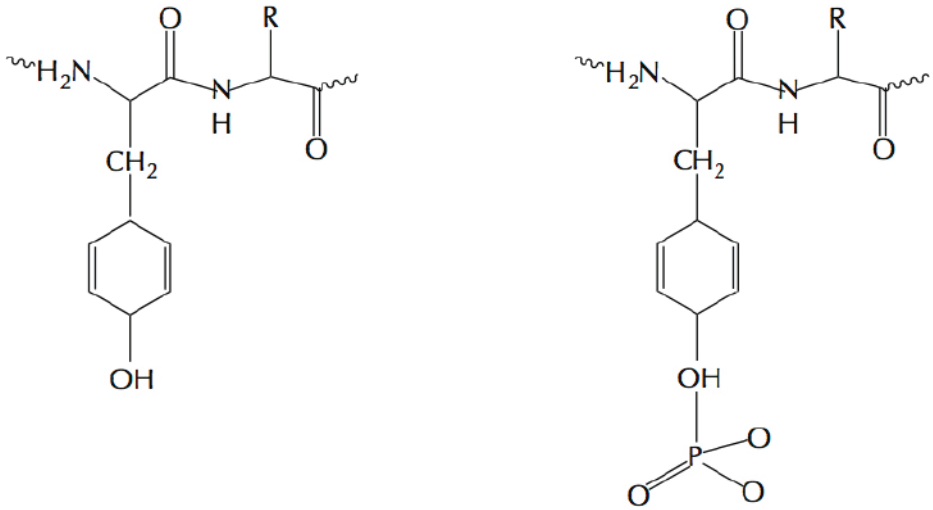


Рис. 11.19 Тирозин (слева) и его посттрансляционная модификация в фосфорилированный тирозин (справа)

Существуют также важные, но редкие посттрансляционные модификации, такие как **дифтамид**. Эта модификация гистидина появляется только в одном белке (**фактор-2 элонгации синтеза белка**), но она универсальна для всех эукариот! Исследователи показали, что дифтамид является мишенью для нескольких токсинов, выделяемых различными патогенными бактериями, в связи с чем возникает вопрос, почему все эукариоты сохраняют эту модификацию, если она делает их такими уязвимыми для патогенов – она должна выполнять какую-то важную, но до сих пор неизвестную функцию в нормальной физиологии.

Поиск модификаций как задача выравнивания

Модификация массы d, присоединенная к аминокислоте, приводит к добавлению d к массе этой аминокислоты. Например, $d = 80$ для фосфорилированных аминокислот (серин, треонин и тирозин), $d = 16$ для модификации пролина в гидроксипролин и $d = 1$ для модификации лизина в **аллизин**. Если d положительная, то полученный модифицированный пептид имеет пептидный вектор, который отличается от исходного пептидного вектора *Peptide*′ вставкой блока из d нулей перед i-м вхождением 1 в *Peptide*′. В более редком случае, когда d отрицательна, модифицированный пептид соответствует удалении блока из $|d|$ нулей из *Peptide*′ (рис. 11.20).

Мы будем использовать термин **блок-индел** для обозначения добавления или делеции блока последовательных нулей из двоичного вектора. Таким образом, применение k модификаций к цепи аминокислот *Peptide* соответствует применению k вставок блока к его пептидному вектору *Peptide*′. Мы определяем $Variants_k(Peptide)$ как набор всех модифицированных вариантов *Peptide* с количеством модификаций до k.

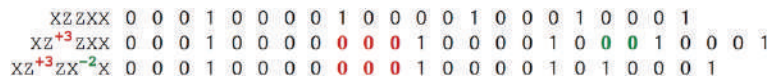


Рис. 11.20 Преобразование пептида XZZXX в пептид $XZ^{+3}ZXX$ соответствует вставке блока из трех нулей (показаны красным) перед вторым появлением 1 в пептидном векторе XZZXX. Преобразование пептида $XZ^{+3}ZXX$ в пептид $XZ^{+3}ZX^{-2}X$ соответствует удалению блока из двух нулей (показаны зеленым) перед четвертым вхождением 1 в пептидный вектор $XZ^{+3}ZXX$

Имея пептид *Peptide* и спектральный вектор *Spectrum′*, наша цель – найти модифицированный пептид от $Variants_k(Peptide)$ с максимальным значением по сравнению со *Spectrum′*.

Задача спектрального выравнивания: *для данного пептида и спектрального вектора найти модифицированный вариант этого пептида, который максимизирует показатель пептидного спектра среди всех вариантов пептида с модификациями до k.*

Input: аминокислотная строка *Peptide*, спектральный вектор *Spectrum′* и целое число *k*.

Output: пептид с максимальным счетом по сравнению со *Spectrum′* среди всех пептидов в $Variants_k(Peptide)$.

Метод грубой силы к задаче спектрального выравнивания будет оценивать каждый пептид в $Variants_k(Peptide)$ по сравнению со спектром. Нам нужно решить эту задачу более эффективно, потому что наша более амбициозная цель состоит в том, чтобы решить следующую задачу, которая потребует многократного применения алгоритма, решающего задачу спектрального выравнивания.

Задача поиска модификации: *по заданному спектру и протеому найти пептид с максимальным значением по этому спектру среди всех модифицированных пептидов в протеоме, содержащих до k модификаций.*

Input: спектральный вектор *Spectrum′*, аминокислотная строка *Proteome* и целое число *k*.

Output: пептид *Peptide*, который максимизирует *Score*(*Peptide′*, *Spectrum′*) среди всех модифицированных вариантов пептидов из *Proteome* с модификациями до *k*.

Подобно модификациям, мутации также можно рассматривать как блочные вставки; например, мутацию V (целая масса 99) в R (целая масса 156) в пептиде

P9 (рис. 11.8) можно рассматривать как вставку блока с $d = 156 - 99 = 57$. Таким образом, мы можем преобразовать поиск модификации в «задачу поиска мутаций», просто заменив слово «модификация» на «мутация» в постановке задачи. В этой новой задаче мутация массы d, примененная к аминокислоте, соответствует разнице между массами этой аминокислоты и другой. Например, мутациям валина (целая масса 99) соответствуют модификации с целыми массами –42, –28, –12, –2, 2, 4, 14, 15, 16, 29, 30, 32, 38, 48, 57, 64 и 87.

Построение сетки Манхэттена для спектрального выравнивания

Задача нахождения пептидного вектора с наивысшей значением, имеющего до k вставных блоков, напоминает задачи выравнивания последовательностей. Это понимание предполагает, что мы должны сформулировать задачу спектрального выравнивания как пример самого длинного пути в задаче DAG.

ОСТАНОВИТЕСЬ и задумайтесь. Как бы вы построили DAG для решения этой задачи?

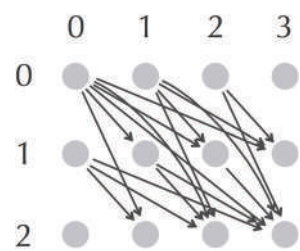


Рис. 11.21 Граф *Southeast*(2,3). Каждый узел графа связан с каждым узлом, лежащим к югу и востоку от него, за исключением узлов в той же строке и столбце

Рассмотрим строку аминокислот $Peptide = a_1 \ldots a_n$ массы m и его модифицированный вариант $Peptide^{mod} = a_1 \ldots a'_n$ массы $m + D$. Постройте манхэттенскую сетку размерностью $(m + 1) \times (m + D + 1)$, в которой каждый узел (i, j) соединен с каждым узлом (i', j') для $0 \le i < i' \le m$ и $0 \le j < j' \le m + D$ (рис. 11.21). Назовем этот граф $Southeast(m, m + D)$, а узлы $(0, 0)$ и $(m, m + D)$ *source* и *sink* соответственно.

Определим путь $Path(Peptide, Peptide^{mod})$ в $Southeast(m, m + D)$, состоящий из n ребер:

$$\begin{aligned}(0,0) &\rightarrow (Mass(a_1), Mass(a'_1)) \\ &\rightarrow (Mass(a_1a_2), Mass(a'_1a'_2)) \\ &\rightarrow \ldots \\ &\rightarrow (Mass(a_1 \ldots a_n), Mass(a'_1 \ldots a'_n)) \\ &= (m, m + D).\end{aligned}$$

Например, рассмотрим аминокислотный мини-алфавит, содержащий X, Y и Z с соответствующими массами 2, 3 и 4. Синий путь на рисунке ниже указывает путь (XYYZX, $\mathbf{XY^{+2}YZ^{-3}X}$):

$$(0,0) \rightarrow (2,2) \rightarrow (5,7) \rightarrow (8,10) \rightarrow (12,11) \rightarrow (14,13).$$

За исключением начального узла (0, 0), каждый узел (*i, j*) на этом пути указывает, что *i*-й элемент *Peptide*′ = (0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1) и *j*-й элемент $Peptide'^{mod}$ = (0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1), оба равны 1.

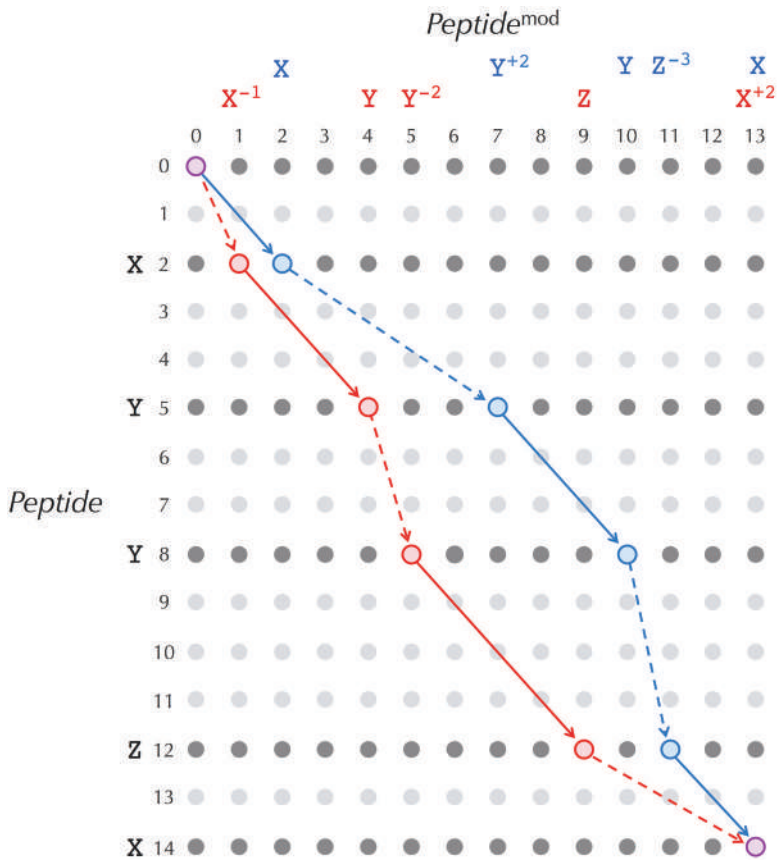


Рис. 11.22 Два пути в *Southeast*(14,13), образованные модификациями $\mathtt{XY^{+2}YZ^{-3}X}$ и $\mathtt{X^{-1}YY^{-2}ZX^{+2}}$, где X, Y и Z имеют массы 2, 3 и 4 соответственно. Диагональные ребра сплошные, а недиагональные ребра – пунктирные. Индексы более темных строк на графе соответствуют входам 1 в пептидном векторе XYYZX. Более темные узлы на этом графе образуют граф PSM

Ребро, соединяющее (*i*, *j*) с (*i*′, *j*′) в *Southeast*(*m*, *m* + *D*), называется **диагональным**, если *i*′ – *i* = *j*′ – *j*, и **недиагональным** в противном случае. Обратите внимание, что если аминокислота $Peptide^{mod}$ не модифицирована, то ребро, соответствующее этой аминокислоте в *Path*(*Peptide*, $Peptide^{mod}$), является диагональным. Аминокислота *a* с модификацией *d* в $Peptide^{mod}$ соответствует недиагональному ребру на этом пути, соединяющему некоторый узел (*i, j*) с узлом (*i* + |*a*|, *j* + |*a*| + *d*).

Теперь мы готовы решить задачу спектрального выравнивания для цепи аминокислот $Peptide = a_1 \dots a_n$ массы m и спектрального вектора $Spectrum = s_1, \dots, s_m + D$. Мы знаем, что модифицированный вариант *Peptide*, решающий эту задачу, должен иметь массу $m + D$. Таким образом, мы можем представить все модифицированные пептиды массы $m + D$ в $Variants_k(Peptide)$ как пути в $Southeast(m, m + D)$ от *source* до *sink* с не более чем k недиагональными ребрами (рис. 11.22). Мы называем эти модифицированные пептиды $Variants_k(Peptide, Spectrum')$.

ОСТАНОВИТЕСЬ и задумайтесь. Соответствует ли каждый путь от источника к стоку в $Southeast(m, m + D)$ модифицированному пептиду-кандидату массой $m + D$?

Хотя каждый пептид в $Variants_k(Peptide, Spectrum')$ соответствует пути от истока к стоку в $Southeast(m, m + D)$, многие пути в этом графе не соответствуют таким модифицированным пептидам. Действительно, поскольку *Peptide* фиксирован, любой путь, соответствующий модифицированному варианту *Peptide*, будет проходить только через строки с индексами

$$0, Mass(a_1), Mass(a_1a_2), \dots, Mass(a_1 \dots a_n) = m,$$

показанными в виде строк с более темными узлами в $Southeast(m, m + D)$ (рис. 11.22).

Таким образом, узлы в других строках $Southeast(m, m + D)$ можно безопасно удалить, в результате чего получится **граф PSM**, обозначенный $PSMGraph(Peptide, Spectrum')$ и показанный на рис. 11.23. Обратите внимание, что $n + 1$ строк узлов в графе PSM имеют индексы i, равные

$$0, Mass(a_1), Mass(a_1a_2), \dots, Mass(a_1 \dots a_n) = m,$$

а не индексы $0, 1, \dots, n$.

В PSM-графе все ребра, входящие в строку с индексом $i = Mass(a_1 \dots a_t)$, берут начало в строке с индексом $Mass(a_1 \dots a_{t-1})$. Поэтому мы определяем $Diff(i)$ как массу аминокислоты a_t. Для пептида XYYZX $Diff(2) = Mass(X) = 2$, $Diff(5) = Mass(Y) = 3$, $Diff(8) = Mass(Y) = 3$, $Diff(12) = Mass(Z) = 4$, and $Diff(14) = Mass(X) = 2$.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы присвоить вес узлам $PSMGraph(Peptide, Spectrum')$ так, чтобы общий вес пути от *source* до *sink*, соответствующий модифицированному пептиду $Peptide^{mod}$, был равен $Score(Peptide^{mod}, Spectrum')$?

Для заданного спектрального вектора $Spectrum' = (s_1, \dots, s_{m+D})$ мы присваиваем вес s_j каждой вершине (i, j) в столбце j графа PSM. При таком назначении весов общий вес узлов на пути, соответствующем $Peptide^{mod}$, равен $Score(Peptide^{mod},$

Spectrum′). Таким образом, решение задачи спектрального выравнивания эквивалентно поиску пути в PSM-графе с максимальным общим весом узлов среди всех путей не более чем с k недиагональными ребрами.

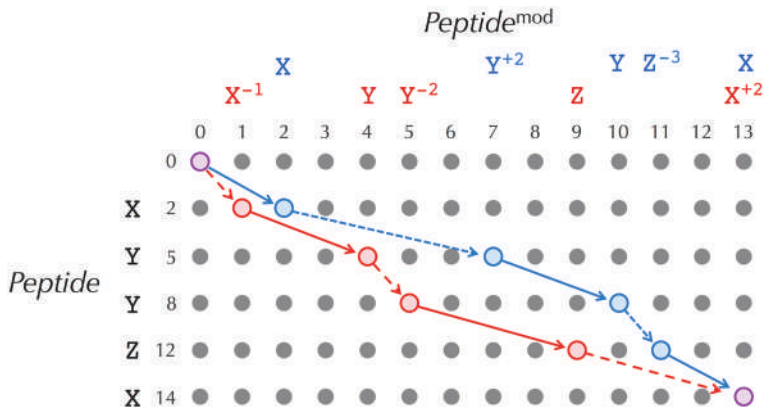


Рис. 11.23 Граф PSM, полученный удалением светлых узлов из графа *Southeast*$(m, m + D)$

Эпилог. От немодифицированных к модифицированным пептидам (Часть 2)

Алгоритм спектрального выравнивания

Мы уже знаем, как найти самый длинный путь в DAG, взвешенной по узлам. Однако неясно, как найти самый длинный путь в DAG при дополнительном ограничении, состоящем в том, что этот путь имеет не более k недиагональных ребер.

В качестве обходного пути мы преобразуем двумерный граф PSM из предыдущего в трехмерный **граф спектрального выравнивания**, состоящий из $k + 1$ слоев (рисунок ниже), где набор узлов в каждом слое совпадает с набором узлов графа PSM. Этот граф будет иметь узлы (i, j, t), где $0 \le i \le m$, $0 \le j \le m + D$ и $0 \le t \le k$. Каждый такой узел наследует вес узла (i, j) в PSM-графе (т. е. амплитуду s_j из $Spectrum' = (s_1, \ldots, s_{m+D})$).

Что касается ребер, то каждый из $k + 1$ слоев графа спектрального выравнивания наследует все диагональные ребра из графа PSM, т. е. каждое диагональное ребро от (i, j) до $(i + x, j + x)$ в графе PSM будет соответствовать $k + 1$ ребрам от (i, j, t) до $(i + x, j + x, t)$ для $0 \le t \le k$.

ОСТАНОВИТЕСЬ и задумайтесь. Слои построенного графа теперь разъединены. Как мы должны их соединить?

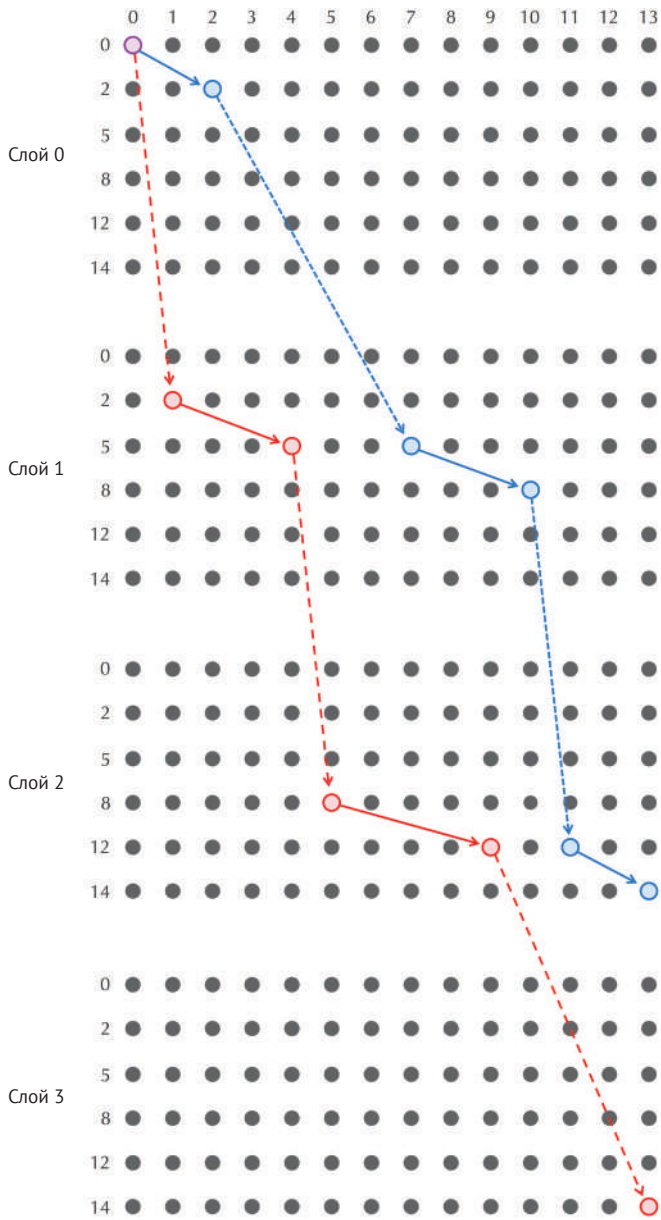


Рис. 11.24 Граф PSM, представленный ранее, преобразован в граф спектрального выравнивания с четырьмя слоями (k = 3). Синие и красные пути на графе PSM соответствуют показанным синим и красным путям, которые представляют соответствующие модифицированные варианты XY + 2YZ – 3X и X – 1YY – 2ZX + 2 пептида XYYZX. Синий путь заканчивается на слое 2, поскольку он соответствует пептиду с двумя модификациями, тогда как красный путь заканчивается на слое 3, поскольку он соответствует пептиду с тремя модификациями

Для каждого недиагонального ребра, соединяющего (i, j) с (i', j') в графе PSM, мы создадим k ребер, соединяющих последовательные слои в графе спектрального выравнивания, соединив (i, j, t) с $(i', j', t + 1)$ для всех t между 0 и $k - 1$. Каждый путь в графе спектрального выравнивания от (0, 0, 0) до $(m, m + D, t)$ соответствует модифицированной версии пептида с t-модификациями (рис. 11.24). Нулевой слой этого графа будет хранить количество пептидов без модификаций, первый слой будет хранить количество пептидов с одной модификацией и т. д.

Чтобы решить задачу спектрального выравнивания, мы определим $Score(i, j, t)$ как максимальную сумму всех путей, соединяющих узел (0, 0, 0) с узлом (i, j, t) в графе спектрального выравнивания. Обратите внимание, что эта сумма равна весу s_j, присвоенному узлу (i, j, t), плюс максимум из значений всех предшественников узла (i, j, t). Один из этих предшественников $(i - Diff(i), j - Diff(i), t)$ находится в том же слое (t), а j других предшественников $((i - Diff(i), j', t - 1)$ при $j' < j)$ расположены в предыдущем слое $(t - 1)$. Это рассуждение приводит к следующему рекурсивному соотношению для вычисления $Score(i, j, t)$:

$$Score(i, j, t) = s_j + \max_{j' < j} \begin{cases} Score(i - Diff(i), j - Diff(i), t) \\ Score(i - Diff(i), j', t - 1) \end{cases}.$$

Таким образом, максимальная сумма значений всех пептидов не более чем с k модификациями является максимальным значением $Score(m, m + D, t)$, когда t находится в диапазоне от 0 до k. Чтобы инициализировать рекурсию, мы предполагаем, что $Score(0, 0, 0) = 0$ и что $Score(0, 0, t) = -\infty$ для всех $1 \le t \le k$.

Хотя эта рекурсия вычисляет значение модифицированного пептида, решающего задачу спектрального выравнивания, нам также необходимо реконструировать этот пептид. Для достижения этой цели нам потребуется реализовать метод возврата, аналогичный описанному в разделе о выравнивании последовательности, который мы оставляем вам в качестве упражнения.

Упражнение. Каково время работы алгоритма спектрального выравнивания?

Заключительная задача. Помимо окаменелости тираннозавра, Асара также проанализировал окаменелость мастодонта возрастом 200 000 лет. Вымирание слоноподобных мастодонтов 10 000 лет назад было вызвано сочетанием изменения климата и охоты людей, вооруженных каменным оружием. В отличие от динозавров, неудивительно, что ученые регулярно идентифицируют пептиды недавно вымерших видов, таких как мастодонты или пещерные медведи.

Проанализируйте пептиды коллагена мастодонта, описанные в статье Асара 2007 года, и решите, какие из них образуют статистически значимые PSM. Можете ли вы определить другие статистически значимые пеп-

тиды мастодонта, которые пропущены в этой статье? Можете ли вы найти неколлагеновые пептиды (особенно пептиды гемоглобина), соответствующие спектрам мастодонтов? Можете ли вы определить количество различных типов посттрансляционных модификаций пептидов мастодонта, решив задачу поиска модификации?

☞ **Загрузить данные 11.4**

Сопутствующие материалы

Предсказание генов

Чтобы предсказать расщепленные гены, исследователи часто пытаются распознать расположение сигналов сплайсинга в экзон-интронных соединениях. Простой пример: динуклеотиды AG и GT по обе стороны от экзона высококонсервативны (рис. 11.25). Чтобы повысить точность этого метода (известного как **статистическое предсказание генов**), исследователи ищут геномные особенности, часто появляющиеся в экзонах и редко в интронах.

Попытки повысить точность методов статистического предсказания генов привели к методам предсказания генов, **основанных на сходстве**, которые базируются на наблюдении, что недавно секвенированный ген часто похож на известный ген другого вида. Например, 99 % генов мыши имеют аналоги у человека.

Однако мы не можем просто искать подобную последовательность в геноме мыши по известным генам человека, поскольку экзонная последовательность и разделение гена на экзоны у разных видов могут быть разными. Чтобы решить эту задачу, методы, основанные на сходстве, иногда ищут набор предполагаемых экзонов в геноме мыши, конкатенация которых соответствует известному человеческому белку.

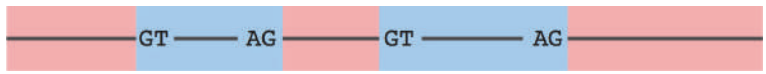


Рис. 11.25 Расщепленный ген с экзонами (красный), разделенными интронами (синий). Интроны обычно начинаются с GT и заканчиваются на AG

Однако не все гены являются расщепленными генами. Например, бактерии вообще не имеют расщепленных генов, что упрощает предсказание бактериальных генов. Такие гены начинаются со стартового кодона, кодирующего метионин (обычно ATG, но иногда также с GTG или TTG), и заканчиваются стоп-кодоном (TAA, TAG или TGA).

Мы можем представить геном длины n как последовательность $n/3$ кодонов. Стоп-кодоны разбивают эту последовательность на сегменты между каждой

парой последовательных стоп-кодонов. Суффиксы этих сегментов, которые начинаются с первого стартового кодона внутри сегмента, называются **открытыми рамками считывания (ORF)**. ORF в пределах одного генома могут перекрываться, потому что существует шесть возможных рамок считывания.

Поскольку существует три стоп-кодона, каждый триплет нуклеотидов в случайно сгенерированной цепи ДНК имеет вероятность 3/64 оказаться стоп-кодоном. Таким образом, ожидаемое количество кодонов между двумя последовательными стоп-кодонами в случайно сгенерированной нуклеотидной цепи (в заданной рамке считывания) равно 64/3. Это означает, что мы ожидаем найти стоп-кодон примерно через каждые 64 нуклеотида (в заданной рамке считывания) в случайно сгенерированной нуклеотидной последовательности.

Однако типичная длина бактериального гена составляет порядка 1000 нуклеотидов. Таким образом, алгоритм предсказания генов может выбирать в качестве генов-кандидатов рамку ORF, длина которой превышает некоторую пороговую длину. К сожалению, такой алгоритм не смог бы обнаружить короткие гены.

Многие алгоритмы предсказания генов также полагаются на тонкие статистические различия между кодирующими и некодирующими областями, такие как систематическая ошибка в **использовании кодонов** или частота каждого кодона. Например, существует шесть кодонов, кодирующих лейцин, но, в то время как CUG кодирует 47 % всех вхождений лейцина в *E. coli*, CUA кодирует только 4 %. Следовательно, ORF с гораздо большим числом вхождений CUG, чем CUA, является геном-кандидатом.

Предсказание бактериального гена также использует несколько консервативных мотивов, часто обнаруживаемых в геномных областях вблизи начала транскрипции РНК. Например, **Прибнов-бокс** (Также известен как Прибнов-Шаллер-бокс. – *Прим. ред.*) представляет собой последовательность из шести нуклеотидов с консенсусом TATAAT, который является важным компонентом для инициации транскрипции у бактерий.

Поиск всех путей в графе

В XIX веке Шарль Пьер Тремо разработал алгоритм навигации по лабиринтам, описанный ниже. Проходя через лабиринт, водите мелом по земле позади себя. Когда вы достигаете перекрестка, неотмеченные пути соответствуют еще неизведанным путям. Так что идите по неизведанному пути так долго, как сможете, пока не встретите тупик или перекресток, на котором отмечены все исходящие пути. В этом случае возвращайтесь назад, пока не встретите выход или перекресток с немаркированным путем или пока не доберетесь до исходной точки; в этом случае лабиринт не имеет выхода.

Алгоритм лабиринта Тремо является примером **поиска в глубину (DFS)**, метода обхода узлов графа. DFS начинается с заданного узла и исследует граф, насколько это возможно, пока не достигнет узла, который не имеет исходящих ребер или из которого мы уже исследовали все ребра. Затем мы возвращаемся назад, пока не достигнем узла с неисследованными ребрами. Обходы предва-

рительного порядка, с которым мы столкнулись в разделе **СОПУТСТВУЮЩИЕ МАТЕРИАЛЫ: От суффиксных деревьев до суффиксных массивов**, предлагает один пример применения DFS к корневым деревьям.

Следующий рекурсивный алгоритм предлагает основанный на DFS метод поиска всех путей между узлом v и стоком узла в графе DAG.

```
AllPaths(Graph, v, sink)
    if v = sink
        Paths ← набор путей, состоящий из одноузлового пути v
    else
        Paths ← пустой набор путей
    for всех исходящих из v ребер (v, ω)
        PathsFromDescendant ← AllPaths(Graph, v, sink)
        добавить (v, ω) в качестве первого ребра к каждому пути
в PathsFromDescendant
        добавить PathsFromDescendant к Paths
    return Paths
```

Задача антисимметричного пути

В основном тексте мы видели, что не каждый путь от *source* до *sink* в *Graph*(*Spectrum*) представляет собой решение задачи декодирования идеального спектра. Эта задача вызвана тем фактом, что каждую массу в спектре можно интерпретировать либо как массу префикса, либо как массу суффикса. Следовательно, у каждого узла, соответствующего массе s, есть узел-близнец (соответствующий *Mass*(*Peptide*) – s). При заданном произвольном узле и его близнеце в *Graph*(*Spectrum*), для того чтобы получить решение, правильный путь от источника к стоку должен проходить ровно через один из этих узлов.

Задача декодирования идеального спектра – это частный случай следующей более общей задачи. Для заданного набора **запрещенных пар** узлов в графе (при восстановлении пептидов запрещенные пары соответствуют близнецам) путь в графе называется **антисимметричным**, если он содержит ровно один узел из каждой запрещенной пары.

Задача антисимметричного пути: *найти антисимметричный путь в DAG.*

Input: DAG с узлами *source* и *sink*, а также набор запрещенных пар узлов в этом DAG.

Output: антисимметричный путь в этом DAG от *source* до *sink*.

Задача антисимметричного пути является *NP*-сложной, но мы не должны терять надежду найти эффективный алгоритм для задачи декодирования идеального спектра, поскольку последняя является частным случаем первой. В частности, запрещенные пары при секвенировании пептидов обладают тем дополнительным свойством, что сумма масс каждой запрещенной пары равна массе всего пептида. На самом деле существует полиномиальный алгоритм решения задачи антисимметричного пути для DAG, удовлетворяющий этому дополнительному свойству, но этот алгоритм выходит за рамки нашей книги.

Преобразование спектров в спектральные векторы

Наша цель – разработать вероятностную модель, описывающую, как пептидный вектор генерирует *целочисленный* спектр, и использовать эту модель для преобразования спектра в спектральный вектор. Чтобы решить эту задачу, мы сначала введем абстрактную модель, которая, по-видимому, не имеет ничего общего с секвенированием пептидов, а скорее описывает вероятностный процесс, преобразующий пептидный вектор $P = (p_1, \ldots, p_m)$ в *бинарный* вектор $X = (x_1, \ldots, x_m)$ той же самой длины. Позже мы увидим, как идеи, развитые для этой модели, помогут нам анализировать реальные спектры.

Определим вероятность того, что P порождает X как $Pr(X|P) = \prod_{i=1}^{m} Pr(x_i|p_i)$, – вероятность того, что p_i в P порождает x_i в X (рис. 11.26). Например, вероятность того, что единица в P порождает единицу в X, записывается как $Pr(1|1)$ и равна некоторому параметру ρ. Вероятность того, что 0 в P порождает 1 в X, записывается как $Pr(1|0)$ и равна некоторому параметру θ. Вероятность того, что 1 в P порождает 0 в X, равна $Pr(0|1) = 1 - \rho$, а вероятность того, что 0 в P порождает 0 в X, равна $Pr(0|0) = 1 - \theta$.

		Символ в *P*	
		0	1
Символ в *X*	0	$1 - \vartheta$	$1 - \rho$
	1	ϑ	ρ

Рис. 11.26 Матрица, описывающая вероятностный процесс преобразования пептидного вектора P в бинарный вектор X

Для алфавита мини-аминокислот, содержащего только две аминокислоты с массами 2 и 3, на рисунке ниже показан пептидный вектор P = (0, 1, 0, 1, 0, 0, 1), генерирующий бинарный вектор X = (0, 0 , 0, 1, 1, 0, 1) с вероятностью

$$Pr(X|P) = (1 - \theta) \cdot (1 - \rho) \cdot (1 - \theta) \cdot \rho \cdot \theta \cdot (1 - \theta) \cdot \rho.$$

Упражнение. Какой бинарный вектор с большей вероятностью будет генерирован пептидным вектором P на рисунке ниже: (0,0,0,1,1,0,1) или (1,0,0,0,1,0,1)?

Пептидный вектор *P*	0	1	0	1	0	0	1
Бинарный вектор *X*	0	0	0	1	1	0	1
$Pr(X\|P)$	$Pr(0\|0)\cdot$	$Pr(0\|1)\cdot$	$Pr(0\|0)\cdot$	$Pr(1\|1)\cdot$	$Pr(1\|0)\cdot$	$Pr(0\|0)\cdot$	$Pr(1\|1)$
	$(1-\vartheta)\cdot$	$(1-\rho)\cdot$	$(1-\vartheta)\cdot$	$\rho\cdot$	$\vartheta\cdot$	$(1-\vartheta)\cdot$	ρ
$Pr(X\|\vec{0})$	$Pr(0\|0)\cdot$	$Pr(0\|0)\cdot$	$Pr(0\|0)\cdot$	$Pr(1\|0)\cdot$	$Pr(1\|0)\cdot$	$Pr(0\|0)\cdot$	$Pr(1\|0)$
	$(1-\vartheta)\cdot$	$(1-\vartheta)\cdot$	$(1-\vartheta)\cdot$	$\vartheta\cdot$	$\vartheta\cdot$	$(1-\vartheta)\cdot$	ϑ
LIKELIHOOD($X\|P$)	$1\cdot$	$\frac{Pr(0\|1)}{Pr(0\|0)}\cdot$	$1\cdot$	$\frac{Pr(1\|1)}{Pr(1\|0)}\cdot$	$1\cdot$	$1\cdot$	$\frac{Pr(1\|1)}{Pr(1\|0)}$
	$1\cdot$	$\frac{1-\rho}{1-\vartheta}\cdot$	$1\cdot$	$\frac{\rho}{\vartheta}\cdot$	$1\cdot$	$1\cdot$	$\frac{\rho}{\vartheta}$
$\log_2$(LIKELIHOOD($X\|P$))	$0+$	$\log_2\frac{1-\rho}{1-\vartheta}+$	$0+$	$\log_2\frac{\rho}{\vartheta}+$	$0+$	$0+$	$\log_2\frac{\rho}{\vartheta}$

Рис. 11.27 Пептидный вектор $P = (0, 1, 0, 1, 0, 0, 1)$ порождает бинарный вектор $X = (0, 0, 0, 1, 1, 0, 1)$ с вероятностью $Pr(X|P)$

Нас интересует следующая задача.

Задача поиска наиболее вероятного пептидного вектора: *найдите наиболее вероятный пептидный вектор для данного бинарного вектора.*

Input: бинарный вектор X и параметры ρ и θ такие, что $0 \le \rho, \theta \le 1$.

Output: пептидный вектор P, который максимизирует $Pr(X|P)$, как определено вероятностями ρ и θ среди всех возможных пептидных векторов.

Определим $Likelihood(X|P)$ как $Pr(X|P)/Pr(X|\vec{0})$, где $\vec{0}$ представляет собой **нулевой вектор**, состоящий из одних нулей. Пептидный вектор $P = (0, 1, 0, 1, 0, 0, 1)$ генерирует бинарный вектор $X = (0, 0, 0, 1, 1, 0, 1)$ с

$$Likelihood(X|P) = \frac{Pr(0|1)}{Pr(0|0)} \cdot \frac{Pr(0|1)}{Pr(1|0)} \cdot \frac{Pr(1|1)}{Pr(1|0)} = \frac{1-\rho}{1-\theta} \cdot \frac{\rho}{\theta} \cdot \frac{\rho}{\theta}.$$

Чтобы не иметь дело с чрезвычайно малыми значениями, полученными в результате многократных умножений в $Likelihood(X|P)$, вместо этого мы будем использовать **логарифмическую вероятность** $\log_2(Likelihood(X|P))$. Поиск пептидного вектора, максимизирующего логарифмическую вероятность, эквивалентен поиску наиболее вероятного пептидного вектора.

На рисунке ниже показано, что пептидный вектор $(0, 1, 0, 1, 0, 0, 1)$ генерирует бинарный вектор $(0, 0, 0, 1, 1, 0, 1)$ с логарифмической вероятностью, равной

$$\log_2\frac{1-\rho}{1-\theta} + \log_2\frac{\rho}{\theta} + \log_2\frac{\rho}{\theta}.$$

Пептидный вектор *P*	0	1	0	1	0	0	1
Бинарный вектор *X*	0	0	0	1	1	0	1
Спектральный вектор *S*	$\log_2\frac{1-\rho}{1-\vartheta}$	$\log_2\frac{1-\rho}{1-\vartheta}$	$\log_2\frac{1-\rho}{1-\vartheta}$	$\log_2\frac{\rho}{\vartheta}$	$\log_2\frac{\rho}{\vartheta}$	$\log_2\frac{1-\rho}{1-\vartheta}$	$\log_2\frac{\rho}{\vartheta}$
Score(*P*, *S*)		$\log_2\frac{1-\rho}{1-\vartheta}$	+	$\log_2\frac{\rho}{\vartheta}$	+		$\log_2\frac{\rho}{\vartheta}$

Рис. 11.28 Сравнение пептидного вектора P со спектральным вектором S в виде скалярного произведения *Score*(*P, S*)

Теперь мы преобразуем бинарный вектор $X = (x_1 \ldots x_m)$ в спектральный вектор $S = (s_1, \ldots, s_m)$, заменяя каждое вхождение 0 на амплитуду $\log_2[(1 - \rho)/(1 - \theta)]$ и каждое вхождение 1 в амплитуду $\log_2[\rho/\theta]$. Например, двоичный вектор (0, 0, 0, 1, 1, 0, 1) будет преобразован в спектральный вектор

$$\left(\log_2\frac{1-\rho}{1-\theta}, \log_2\frac{1-\rho}{1-\theta}, \log_2\frac{1-\rho}{1-\theta}, \log_2\frac{\rho}{\theta}, \log_2\frac{\rho}{\theta}, \log_2\frac{1-\rho}{1-\theta}, \log_2\frac{\rho}{\theta}\right).$$

Обратите внимание, что $\log_2(Likelihood(X|P)$ – это просто скалярное произведение пептидного вектора $P = (p_1, \ldots, p_m)$ и спектрального вектора $S = (s_1, \ldots, s_m)$,

$$P \cdot S = p_1 \cdot s_1 + \ldots + p_m \cdot s_m.$$

Обозначаим $P \cdot S$ как *Score*(*P, S*) (рис. 11.29) и определим счет между пептидным вектором и спектральным вектором разной длины как $-\infty$. Таким образом, мы преобразовали задачу наиболее вероятного пептидного вектора в задачу секвенирования пептидов (рис. 11.28).

Наше преобразование бинарного вектора *X* в спектральный вектор *S* было основано на простой вероятностной модели (описывающей, как пептидный вектор генерирует бинарный вектор) всего с двумя параметрами, ρ и θ. Оставшаяся часть этого раздела посвящена тому, как на практике спектры преобразуются в спектральные векторы.

Если бы масс-спектрометры генерировали бинарные спектры, то мы могли бы начать с формирования большой **тестовой выборки** аннотированных спектров, для которых известны пептиды, генерирующие эти спектры. Затем мы могли бы оценить ρ (как частоту единиц в бинарных спектрах, генерируемых единицами в пептидных векторах) и θ (как частоту единиц в бинарных спектрах, генерируемых нулями в пептидных векторах) по всем аннотированным спектрам в тестовой выборке. Но, поскольку реальные масс-спектрометры генерируют *целочисленные*, а не *бинарные* спектры, получение результатов становится более сложным.

ОСТАНОВИТЕСЬ и задумайтесь. Можете ли вы разработать вероятностную модель, которая преобразует пептидный вектор (0, 1, 0, 1, 0, 0, 1) в целочисленный вектор (3, 4, 2, 6, 9, 4, 7)?

Однако похожая вероятностная модель будет работать, если мы определим вероятность преобразования нулей и единиц в пептидном векторе в различные интенсивности в реальных спектрах (а не в нули и единицы, как раньше). Фактически преобразование реальных спектров в спектральные векторы основано на аналогичной модели логарифмической вероятности, использующей десятки вероятностных параметров. Алгоритмы преобразования реальных спектров в спектральные векторы будут пытаться оптимизировать эти параметры так, чтобы амплитуды в префиксных координатах были максимальными, а амплитуды в шумовых координатах минимизированы.

Чтобы получить эти параметры, нам снова нужно построить большую обучающую выборку аннотированных спектров. Мы можем рассмотреть все пики с определенным уровнем интенсивности во всех спектрах и вычислить, какая часть из них аннотирована префиксными или суффиксными пептидами. Например, только 19 и 45 % из десяти пиков максимальной интенсивности в спектрах диссоциации, вызванной столкновениями (которые аналогичны тем, которые были получены в лаборатории Асара), объясняются префиксными и суффиксными пептидами соответственно. Остальные высокоинтенсивные пики рассматриваются как шум. Имея спектр, созданный неизвестным пептидным вектором $P = (p_1, \ldots, p_m)$, его спектральный вектор $S = (s_1, \ldots, s_m)$ выводится с использованием этих частот, так что s_i представляет собой соотношение логарифмической вероятности $\log_2(Pr_1/Pr_0)$, где Pr_1 является оценкой вероятности того, что $p_i = 1$ и Pr_0 является оценкой вероятности того, что $p_i = 0$. Полное обсуждение деталей алгоритма генерации спектральных векторов выходит за рамки этого материала.

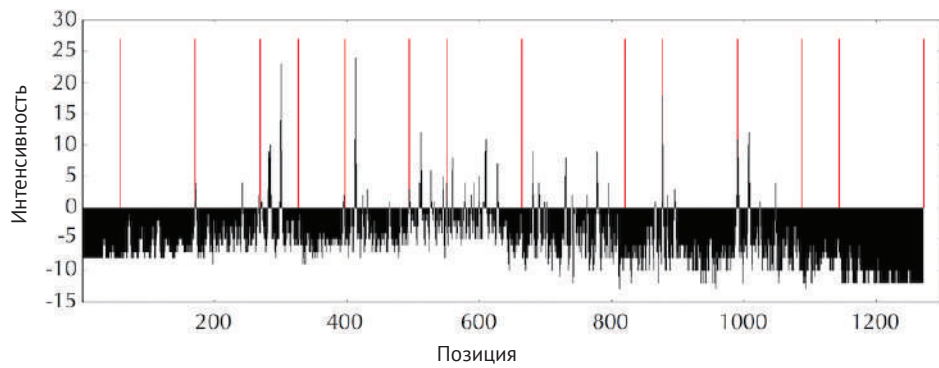


Рис. 11.29 Амплитуды спектрального вектора

На рисунке ниже показан набор префиксных масс для *DinosaurPeptide* вместе с амплитудами спектрального вектора, соответствующими этим массам. Обратите внимание на рис. 11.29, где большинство амплитуд спектрального вектора отрицательны. Синие элементы на рисунке ниже соответствуют позициям, которые значительно превышают среднее значение амплитуды.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
аминокислота	G	L	V	G	A	P	G	L	R	G	L	P	G	K
масса	57	113	99	57	71	97	57	113	156	57	113	97	57	128
масса префикса	57	170	269	326	397	494	551	664	820	877	990	1087	1144	1272
амплитуда	-8	+1	-4	-6	-6	+3	+1	-4	-8	+18	+11	-10	-7	0

Рис. 11.30 Массы аминокислот в GLVGAPGLRGLPGK (вторая строка), массы префикса для этого пептида (третья строка) и соответствующие элементы спектрального вектора для *DinosaurSpectrum* воспроизведены ниже

Синие элементы соответствуют амплитудам, которые значительно выше среднего, что является отрицательным. Обратите внимание, что второй и третий по высоте пики в спектре (обозначенные как b_{10} и b_{11}) соответствуют максимальным амплитудам +18 и +11 в спектральном векторе *DinosaurSpectrum*. Также обратите внимание, что, поскольку на рис. 11.31 (посередине) нет пика b_{12} (или y_2), $s_{1087} = 10$ очень мало.

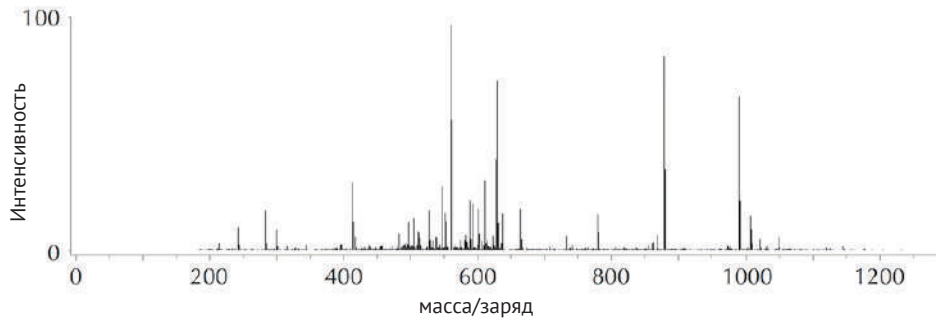


Рис. 11.31 Амплитуды спектрального вектора *DinosaurSpectrum*

Теорема о бесконечных обезьянах

В «Путешествиях Гулливера» Джонатана Свифта профессор Большой академии в Лагадо просит своих студентов генерировать случайные цепи букв, вращая рукоятки машины. По словам профессора, со временем Академия будет выпускать блестящие работы по всем предметам.

Вдохновленная сатирической насмешкой Свифта над определенными академиками, **теорема о бесконечных обезьянах** утверждает, что бессмертная обезьяна, печатающая бесконечную последовательность символов на пишущей машинке, однажды воспроизведет «Гамлета». В более технических терминах эта теорема утверждает, что бесконечная случайная строка **почти наверняка** содержит произвольный заданный текст в качестве подстроки или с вероятностью, равной 1.

ОСТАНОВИТЕСЬ и задумайтесь. В 2003 году исследователи поместили пишущую машинку в вольер для обезьян и обнаружили, что обезьяны печатали букву «S» снова и снова. Возможно, что бесконечная случайная строка, сгенерированная обезьяной, будет содержать только букву «S». Как же тогда эта строка представит нам текст «Гамлета» почти наверняка?

Вероятностное пространство пептидов в словаре

В основном тексте мы определили вероятность *Peptide* как $1/20^{|Peptide|}$, а вероятность коллекции пептидов *Dictionary* мы определили как

$$Pr(Dictionary) = \sum_{\text{each peptide } Peptide \text{ in } Dictionary} \frac{1}{20^{|Peptide|}}.$$

Но почему мы использовали вероятностную запись? В конце концов, рассмотрим следующее упражнение, которое показывает, что *Pr*(*Dictionary*) может быть больше 1.

Упражнение. Если *Dictionary* – это набор всех пептидов длины не более 10, что такое *Pr*(*Dictionary*)?

Однако вспомним, что пептид может соответствовать спектру тогда и только тогда, когда его масса равна массе спектра. Таким образом, ни один пептид в *спектральном* словаре не может содержать другой пептид в словаре в качестве подстроки, т. е. спектральный словарь образует **множество, свободное от подстрок**.

Упражнение. Докажите, что если *Dictionary* – это множество, не содержащее подстрок, то $Pr(Dictionary) \leq 1$.

Однако вам может быть интересно, какое событие соответствует *Pr*(*Dictionary*). Точнее, что такое «вероятностное пространство» основных исходов, из которых формируется *Dictionary*? Предлагаемое нами вероятностное пространство содержит все протеомы-приманки длины n, где n – длина самого длинного пептида в спектральном словаре *Dictionary*. В этом пространстве мы будем предполагать, что каждый протеом-приманка имеет одинаковую вероятность. Таким образом, наше вероятностное пространство состоит из 20^n элементов, каждый с вероятностью $1/20^n$. Обратите внимание на изменение: вместо рассмотрения вероятностного пространства всех пептидов в *Dictionary* мы переключились на рассмотрение всех протеомов-приманок.

Каждая строка *Peptide* в *Dictionary* встречается ровно в $20^{n-|Peptide|}$ протеомах-приманках в качестве их первого пептида. Суммарные вероятности всех этих протеомов-приманок составляют

$$20^{n-|Peptide|} \cdot \frac{1}{20^n} = \frac{1}{20^{|Peptide|}},$$

что и представляет собой $Pr(Peptide)$. Поскольку спектральные словари не содержат подстрок, каждый протеом-приманка имеет не более одного пептида из спектрального словаря в его первой позиции. Таким образом, $Pr(Dictionary)$ – это просто объединенная вероятность всех протеомов-приманок, которые начинаются с одного из пептидов в *Dictionary*.

Действительно ли динозавры являются предками птиц?

Помимо загадочного присутствия пептидов гемоглобина в спектрах *T. rex* у Асары, ученые недавно подвергли сомнению гипотезу о том, что птицы произошли от наземных динозавров, таких как тираннозавр, и что полет был осуществлен с помощью биофизически невероятной **наземной модели**. Эта гипотеза предполагает, что для того, чтобы превратиться в птиц, динозавры сначала должны были уменьшить свой размер, одновременно развив перья (возможно, самое сложное эволюционное изобретение для полета).

Большинство ранних исследований динозавров отдавали предпочтение доказательствам существования небольшого древесного животного как более логичной гипотезы о предке птиц. Эта гипотеза предполагает, что до того, как развилась система устойчивого полета, ранние птицы использовали аэродинамику с помощью гравитации – парашютирование и планирование (последнее используется современными белками-летягами). Так, крошечный *Scansoriopteryx*, окаменелости которого содержат отпечатки перьев и чьи приспособления к ногам указывают на древесный образ жизни, соперничает с *T. rex* за честь быть предком птиц.

Рис. 11.32 Художественное воссоздание *Scansoriopteryx*

Если вам интересно узнать больше про эволюционные споры о происхождении птиц, мы предлагаем две статьи, по одной с каждой стороны дискуссии:

- «Юрский архозавр – птица, не относящаяся к динозаврам» Стивена Черкаса и Алана Федуччиа («Jurassic archosaur is a non–dinosaurian bird», https://link.springer.com/article/10.1007/s10336-014-1098-9).

- «Три генома крокодилов раскрывают наследственные закономерности эволюции среди архозавров» Ричарда Грина и др. («Three crocodilian genomes reveal ancestral patterns of evolution among archosaurs», https://www.science.org/doi/full/10.1126/science.1254449).

Библиографические примечания

Пептиды *T. rex* были описаны Asara et al., 2007[1], и подверглись критике в Pevzner, Kim, and Ng, 2008[2], и Buckley et al., 2008[3]. Статья «ДНК динозавра» Woodward, Weyand, and Bunnell, 1994[4], была опровергнута Hedges and Schweitzer, 1995[5]. Czerkas and Feduccia, 2014[6], недавно утверждали, что птицы не произошли от динозавров, в то время как Green et al., 2014[7], недавно утверждали обратное.

Chen et al., 2001[8], решили задачу антисимметричного пути в случае графов, происходящих из масс-спектров. Поиск в базе данных белков с целью идентификации пептидов в масс-спектрометрии был впервые проведен Eng, McCormack, and Yates, 1994[9]. Алгоритм спектрального выравнивания был представлен Pevzner, Dancík, and Tang, 2000[10]. Концепция спектрального словаря и алгоритм для оценки статистической значимости PSM были представлены Kim, Gupta, and Pevzner, 2008[11], и Kim et al., 2009[12].

1 https://www.ncbi.nlm.nih.gov/pubmed/17431180.
2 https://science.sciencemag.org/content/321/5892/1040.2.
3 https://www.ncbi.nlm.nih.gov/pubmed/18174420.
4 https://science.sciencemag.org/content/266/5188/1229.
5 https://www.ncbi.nlm.nih.gov/pubmed/7761839.
6 https://link.springer.com/article/10.1007/s10336-014-1098-9.
7 https://www.ncbi.nlm.nih.gov/pubmed/25504731.
8 https://www.ncbi.nlm.nih.gov/pubmed/11535179.
9 https://www.ncbi.nlm.nih.gov/pubmed/24226387.
10 https://stepik.org/lesson/240423/step/Mutation-Tolerant%20Protein%20Identification%20by%20Mass%20Spectrometry.
11 https://www.ncbi.nlm.nih.gov/pubmed/18597511.
12 https://www.ncbi.nlm.nih.gov/pubmed/18703573.

Предметный указатель

И

К

Л

М

Н

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Филлип Компо и Павел Певзнер

Алгоритмы биоинформатики

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Люско И. Л.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 55,41. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**