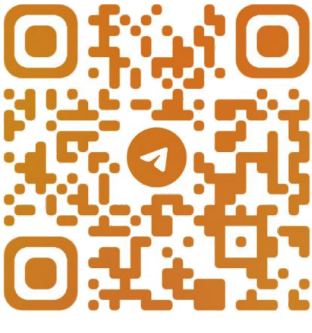
Классические задачи Computer Science на языке Java

Дэвид Копец

MANNING

Classic Computer Science Problems in Java

DAVID KOPEC



MANNING
SHELTER ISLAND

Дэвид Копец

Классические задачи Computer Science на языке Java

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.1
УДК 004.43
К65

Копец Дэвид

К65 Классические задачи Computer Science на языке Java. — СПб.: Питер, 2022. — 288 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-3911-8

Столкнулись с «неразрешимой» проблемой при разработке программного обеспечения? Скорее всего, кто-то уже справился с этой задачей, и вы можете не ломать голову. Дэвид Копец собрал наиболее полезные готовые решения, принципы и алгоритмы. «Классические задачи Computer Science на языке Java» — это мастер-класс по программированию, содержащий 55 практических примеров, затрагивающих самые актуальные темы: базовые алгоритмы, ограничения, искусственный интеллект и многое другое.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1617297601 англ.
ISBN 978-5-4461-3911-8

Краткое содержание

Оглавление

Хочу выразить огромную благодарность Синбэнь Чэню из SUNY Suffolk и Вэй Киань Чэню из Champlain College, которые были для меня наставниками и благодаря которым я стал преподавать.

Благодарности

Хочу сказать спасибо издательству Manning и всем, кто помогал в создании этой книги. Особенно хочу поблагодарить редактора Дженни Стаут (Jenny Stout) за ее доброту и поддержку в самые трудные для меня времена; редактора отдела технического развития Фрэнсис Буонтемпо (Frances Buontempo) за внимание к деталям; рецензента Брайана Сойера (Brian Sawyer) за то, что он всегда верил в развитие компьютерных наук и всегда был нашим голосом разума; редактора Энди Кэрролла (Andy Carroll) за обнаружение моих недочетов; Радмилу Эрчеговац (Radmila Ercegovac) за помощь в продвижении книг по всему миру; технического редактора Жан-Франсуа Морена (Jean-François Morin) за поиск способов сделать код более чистым и современным. Выражаю особую благодарность редактору моего проекта Дейрдре Хиам (Deirdre Hiam), корректору Кэти Теннант (Katie Tennant) и научному редактору Алексу Драгосавлевичу (Aleks Dragosavljević). Я очень благодарен всем специалистам издательства Manning за содействие в подготовке этой книги.

Спасибо Брайану Гетцу (Brian Goetz) за очень занимательное и увлекательное, а главное, полезное для читателей интервью. Для меня было большой честью побеседовать с таким человеком.

Спасибо моей жене Ребекке и маме за неизменную поддержку в этот сложный год.

Благодарю всех рецензентов: Андреса Сакко (Andres Sacco), Эзре Симелоффа (Ezra Simeloff), Яна ван Нимвегена (Jan van Nimwegen), Келума Прабата Сенанаяке (Kelum Prabath Senanayake), Кимберли Уинстон-Джексон (Kimberly

Winston-Jackson), Рафаэлле Вентаглио (Raffaella Ventaglio), Раушана Джа (Raushan Jha), Саманту Берк (Samantha Berk), Саймона Чоке (Simon Tschöke), Виктора Дурана (Víctor Durán) и Уильяма Уиллера (William Wheeler). Ваши идеи сделали эту книгу лучше. Я ценю внимание каждого и время, затраченное на рецензирование этой книги.

А самое главное, я благодарен читателям, поддержавшим серию книг «Классические задачи Computer Science». Если вам понравилась эта книга, пожалуйста, оставьте свои отзывы. Мы будем рады получить их!

Об авторе

Дэвид Копец — старший преподаватель на кафедре компьютерных наук и инноваций в колледже Шамплейн в Берлингтоне, штат Вермонт. Он опытный разработчик программного обеспечения и автор книг *Classic Computer Science Problems in Python*[1] (Manning, 2019), *Classic Computer Science Problems in Swift* (Manning, 2018) и *Dart for Absolute Beginners* (Apress, 2014). А еще Дэвид ведет подкаст.

[1] *Копец Д.* Классические задачи Computer Science на языке Python. — СПб.: Питер, 2020.

Об иллюстрации на обложке

Иллюстрация на обложке называется «Леди с Варварийского берега во всей своей красе»[1] (второе название: «Парадное одеяние знатной дамы из Берберии, 1700 год»[2]) и позаимствована из книги «Коллекция платьев разных народов, старинных и современных»[3], изданной в Лондоне в 1757–1772 годах. Как указано на титульном листе, это гравюры, выполненные на медных пластинах и раскрашенные гуммиарабиком.

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

Разнообразие иллюстраций в книгах Джефферис — яркое свидетельство уникальности и оригинальности народов мира в то время. С тех пор тенденции

---

[1] Dame de la Côte de Barbarie dans tout sa parure.
[2] Full dress of a lady of quality in Barbary, in 1700.
[3] A Collection of the Dresses of Different Nations, Ancient and Modern.

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

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

От издательства

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

Мы будем рады узнать ваше мнение!

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

Введение

Благодарю вас за покупку книги «Классические задачи Computer Science на языке Java». На протяжении двух десятилетий Java был одним из самых популярных языков программирования. Сегодня он доминирует в разработке корпоративных приложений, приложений в сфере высшего образования и мобильных приложений для Android. В этой книге я постараюсь научить вас чему-то большему, чем просто язык программирования. Я надеюсь, что вы начнете думать о Java как об инструменте для решения вычислительных задач. Задачи, описанные в этой книге, среднего уровня. С их помощью опытные программисты смогут освежить свои знания в области информатики, изучив некоторые расширенные возможности языка. Студенты и программисты-самоучки смогут быстрее обучиться программированию, рассматривая общеприменимые методы решения проблем. Эта книга охватывает такое разнообразие методов решения задач, что каждый найдет в ней что-то для себя.

Это издание не является введением в Java. На эту тему есть множество других превосходных книг, выпущенных как в Manning, так и в других издательствах. В книге предполагается, что вы уже программист среднего или высокого уровня на Java. Для примеров использовалась относительно свежая версия Java 11, но вам не нужно знать все тонкости последней версии языка. Фактически примеры были подобраны так, чтобы они послужили учебным материалом, помогающим читателям достичь высокого уровня мастерства. Однако эта книга не подходит для читателей, совершенно не знакомых с Java.

Есть мнение, что компьютеры в информатике — то же самое, что телескопы в астрономии. Если это так, то, возможно, язык программирования подобен объективу

телескопа. В любом случае термин «классические задачи программирования» здесь означает «задачи программирования, которые обычно преподают в курсе информатики для студентов».

Существуют задачи программирования, которые предлагают решить начинающим программистам и которые уже стали достаточно распространенными, чтобы считаться классическими, как в аудитории во время экзамена на степень бакалавра (в области информатики, разработки программного обеспечения и т. п.), так и в рамках учебника среднего уровня по программированию (например, в книге начального уровня по искусственному интеллекту или алгоритмам). В данном издании вы найдете набор именно таких задач.

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

ДЛЯ КОГО ЭТА КНИГА

Java используется в самых разных сферах, таких как разработка мобильных приложений, создание корпоративного программного обеспечения, e-commerce и многое другое. Java иногда критикуют за многословность и отсутствие некоторых современных функций. Однако сегодня Java — один из самых популярных языков программирования. Первоначально его создатель, Джеймс Гослинг, задумывал Java как более совершенную версию языка C++. Он стремился создать более мощный объектно-ориентированный язык программирования с функциями безопасности, оптимизируя некоторые из наиболее слабых сторон C++. В этом отношении, на мой взгляд, язык Java добился больших успехов.

Java — мощный объектно-ориентированный язык программирования. Однако многие люди, будь то разработчики Android или создатели корпоративных приложений, в основном пользуются «гибридными API». Вместо того чтобы работать над решением интересных вопросов, они тратят время, изучая каждый нюанс SDK или библиотеки. Цель этой книги — упорядочить ваши знания. А еще существуют разработчики, не имеющие специального образования в области программирования. Если вы один из таких программистов с опытом разработки на языке Java, но не знакомы с CS, эта книга будет очень полезна для вас.

Обычно Java изучают как второй, третий, четвертый или пятый язык, много лет проработав в сфере разработки программного обеспечения. Таким специалистам видение старых проблем, с которыми они сталкивались, работая с другими

языками, поможет ускорить изучение Java. Для них эта книга может быть хорошим справочником, который стоит пролистать перед собеседованием при приеме на работу. Также в ней можно найти некоторые способы решения проблем, которые разработчики ранее и не думали использовать в своей работе. В первую очередь я бы посоветовал изучить оглавление, чтобы увидеть, есть ли в этой книге темы, которые вас волнуют.

Эта книга предназначена как для начинающих, так и для опытных программистов. Последние, желая усовершенствовать свои знания в Java, найдут здесь темы для углубленного изучения. Программисты среднего уровня смогут изучить темы на свой выбор. Также эта книга будет ценным справочным материалом для подготовки к собеседованию.

Издание полезно не только для профессиональных программистов, но и для студентов, обучающихся по программам бакалавриата и интересующихся Java. Эту книгу нельзя считать строгим введением в структуры данных и алгоритмы. Это не учебник по структурам данных и алгоритмам. Здесь вы не найдете доказательств или широкого использования нотации «О большого». Книга представляет собой доступное практическое руководство по методам решения проблемных вопросов, которые должны стать конечным продуктом изучения структуры данных, алгоритмов и классов искусственного интеллекта.

В книге описаны синтаксис и семантика Java. Читатель без опыта программирования мало что извлечет из нее, а программист без опыта программирования на Java наверняка столкнется с трудностями. Иначе говоря, это издание, предназначенное для разработчиков с опытом программирования на языке Java и студентов, изучающих компьютерные технологии.

КАКИЕ ЗАДАЧИ ПРЕДСТАВЛЕНЫ В ИЗДАНИИ

Глава 1 познакомит вас с методами решения задач, которые, вероятно, известны большинству читателей. Такие действия, как рекурсия, запоминание и манипулирование битами, — важные строительные блоки других методов, которые будут рассмотрены в последующих главах.

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

В главе 3 мы построим структуру для решения широкого круга задач, которые могут быть абстрактно определены переменными ограниченных областей

изменения (limited domains). Это такая классика, как задача восьми ферзей, задача о раскрашивании карты Австралии и криптоарифметика SEND + MORE = MONEY.

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

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

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

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

Глава 8 посвящена состязательному поиску в идеальных информационных играх для двух игроков. Вы изучите алгоритм поиска, известный как минимакс, который можно применять для разработки искусственного противника, способного хорошо играть в такие игры, как шахматы, шашки и Connect Four.

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

Глава 10 представляет собой интервью с Брайаном Гетцем, архитектором языка Java в Oracle. В ней я приведу несколько мудрых советов Брайана.

ОБ ИСХОДНОМ КОДЕ

Исходный код в этой книге написан в соответствии с версией 11 языка Java. В нем используются функции, которые стали доступными только в Java 11, поэтому часть кода не будет работать на более ранних версиях Java. Вместо того чтобы пытаться заставить примеры работать в более ранней версии, просто загрузите последнюю версию Java, прежде чем начинать работу с книгой. Я выбрал версию 11, так как это самая последняя LTS (с долгосрочной поддержкой) версия

Java, выпущенная на момент написания книги. Код должен работать с более поздними (и будущими) версиями Java. Фактически значительная часть кода будет работать с версиями Java начиная с Java 8. Я знаком с программистами, которые по разным причинам все еще применяют Java 8 (cough Android), но чтобы подчеркнуть ценность изучения новых функций языка, я стану использовать более новую его версию.

В этой книге применяется только стандартная библиотека Java, поэтому весь приведенный в ней код должен работать на любой платформе, где поддерживается Java (macOS, Windows, GNU/Linux и т. п.). Код был протестирован только на OpenJDK (основная реализация Java, доступная на http://openjdk.java.net), хотя, скорее всего, бóльшая его часть будет работать в любой реализации Java.

В книге не объясняется, как использовать инструменты Java, такие как редакторы, IDE и отладчики. Исходный код книги доступен в интернете в репозитории GitHub по адресу https://github.com/davecom/ClassicComputerScienceProblemsInJava. Исходный код разделен на папки по главам. Имена исходных файлов вы увидите в тексте глав в заголовках листингов кода. Эти исходные файлы находятся в соответствующих папках хранилища.

Обратите внимание на то, что репозиторий организован как рабочее пространство Eclipse. Eclipse — это популярная бесплатная программная платформа с открытым исходным кодом Java IDE, существующая для всех основных операционных систем и доступная на eclipse.org. Самый простой способ использовать репозиторий исходного кода — открыть его как рабочее пространство Eclipse после его загрузки. Затем для решения данного вопроса вы можете развернуть каталог `src`, развернуть архив, представляющий главу, щелкнуть правой кнопкой мыши (или щелкнуть, удерживая клавишу Control на Mac) на файле, содержащем метод `main()`, и выбрать в раскрывающемся меню пункт Run As ▸ Java Application (Запуск от имени ▸ Приложение Java). Я не буду предоставлять учебное пособие по Eclipse, так как, думаю, для большинства программистов среднего уровня это нецелесообразно. Также я предполагаю, что многие программисты предпочтут использовать эту книгу с альтернативными средами Java.

Поскольку мы работаем с классическим языком Java, вы можете запустить любой исходный код из этой книги в выбранной вами среде IDE, будь то NetBeans, IntelliJ или другая удобная для вас. Если вы решите это сделать, обратите внимание на то, что я не смогу предложить поддержку импорта проектов в выбранную вами среду. Большинство IDE можно импортировать из Eclipse.

Если вы только начинаете изучать Java, то самый простой способ настроить свой компьютер с использованием исходного кода из этой книги — сделать следующее.

1. Загрузить и установить релиз Java 11 или более новый с сайта openjdk.java.net.
2. Загрузить и установить Eclipse с сайта eclipse.org.
3. Загрузить исходный код книги из репозитория https://github.com/davecom/ClassicComputerScienceProblemsInJava.
4. Открыть репозиторий целиком в виде рабочей области в Eclipse.
5. Щелкнуть правой кнопкой мыши на файле исходного кода, который нужно запустить, и выбрать команду меню Run As ▸ Java Application (Запуск от имени ▸ Приложение Java).

В этой книге нет примеров, которые формировали бы графический вывод или применяли графический пользовательский интерфейс (GUI). Почему? Цель ее состоит в том, чтобы решить поставленные задачи с помощью максимально кратких и удобочитаемых методов. Зачастую вывод графики мешает или делает решения значительно более сложными, чем нужно, чтобы проиллюстрировать рассматриваемые методику или алгоритм.

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

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

ДОПОЛНИТЕЛЬНЫЕ ОНЛАЙН-РЕСУРСЫ

Это третья книга из серии «Классические задачи Computer Science», опубликованная издательством Manning. Первой была *Classic Computer Science Problems in Swift*, вышедшая в 2018 году, а затем *Classic Computer Science Problems in Python*,

увидевшая свет в 2019-м. В каждой книге этой серии мы стремимся представить видение конкретного языка в свете решения одних и тех же (в основном) задач программирования.

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

1 Простые задачи

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

1.1. РЯД ФИБОНАЧЧИ

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

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

Первое число в последовательности Фибоначчи — 0, четвертое — 2. Отсюда следует, что для получения значения любого числа *n* в последовательности Фибоначчи можно использовать формулу

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

1.1.1. Первый вариант рекурсии

Приведенная формула для определения числа в последовательности Фибоначчи (рис. 1.1) представляет собой псевдокод, который можно легко перевести в *рекурсивный* метод Java. (Рекурсивным называется метод, который вызывает сам себя.) Такой механический перевод станет нашей первой попыткой написать

метод, возвращающий заданное значение последовательности Фибоначчи (листинг 1.1).

Листинг 1.1. Fib1.java

```
package chapter1;

public class Fib1 {

    // Этот метод вызывает ошибку java.lang.StackOverflowError
    private static int fib1(int n) {
        return fib1(n − 1) + fib1(n − 2);
    }
```

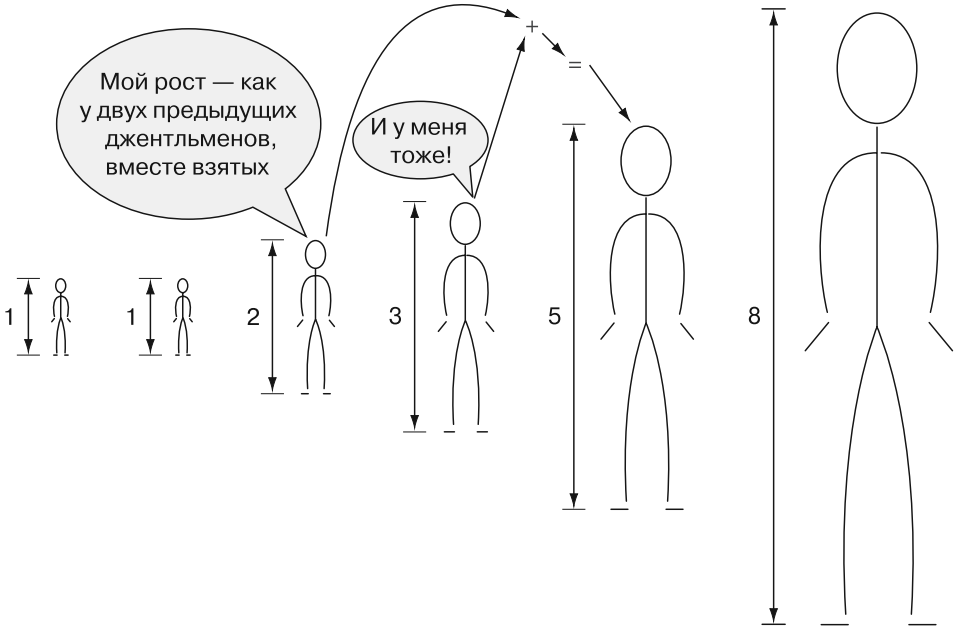


Рис. 1.1. Рост каждого человечка равен сумме роста двух предыдущих

Попробуем запустить этот метод, вызвав его и передав ему значение (листинг 1.2).

Листинг 1.2. Fib1.java (продолжение)

```
    public static void main(String[] args) {
        // Не запускайте это!
        System.out.println(fib1(5));
    }
}
```

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

```
Exception in thread "main" java.lang.StackOverflowError
```

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

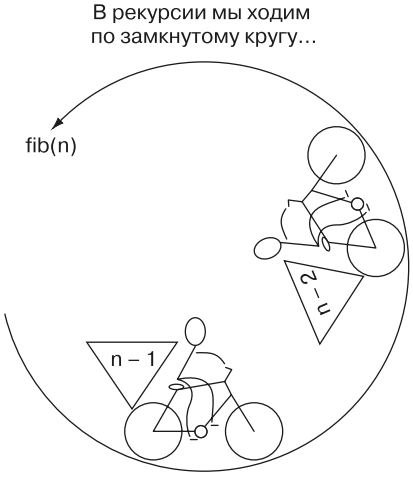


Рис. 1.2. Рекурсивная функция fib(n) вызывает сама себя с аргументами *n* – 1 и *n* – 2

1.1.2. Использование базовых случаев

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

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

ПРИМЕЧАНИЕ

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

Листинг 1.3. Fib2.java

```
package chapter1;

public class Fib2 {
    private static int fib2(int n) {
        if (n < 2) { return n; }
        return fib2(n - 1) + fib2(n - 2);
    }
```

Метод `fib2()` может быть успешно вызван и возвращает правильные результаты. Попробуйте вызвать его для нескольких небольших значений (листинг 1.4).

Листинг 1.4. Fib2.java (продолжение)

```
    public static void main(String[] args) {
        System.out.println(fib2(5));
        System.out.println(fib2(10));
    }
}
```

Не пытайтесь вызывать метод `fib2(40)`. Он будет выполняться очень долго! Почему? Потому что каждый вызов `fib2()` приводит к двум новым вызовам `fib2()` — рекурсивным вызовам `fib2(n-1)` и `fib2(n-2)` (рис. 1.3). Другими словами, дерево вызовов растет в геометрической прогрессии. Например, вызов `fib2(4)` приводит к такой последовательности вызовов:

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

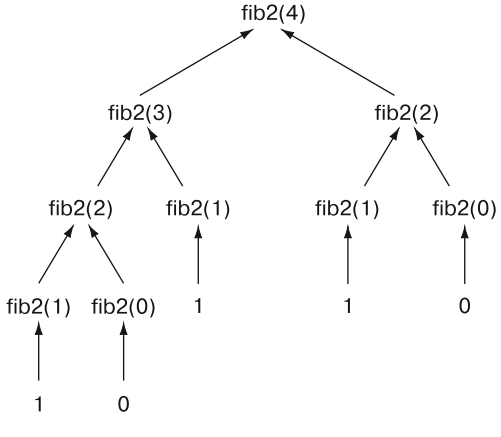


Рис. 1.3. Каждый вызов fib2(), не являющийся базовым случаем, приводит к еще двум вызовам fib2()

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

1.1.3. Спасение — в мемоизации

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

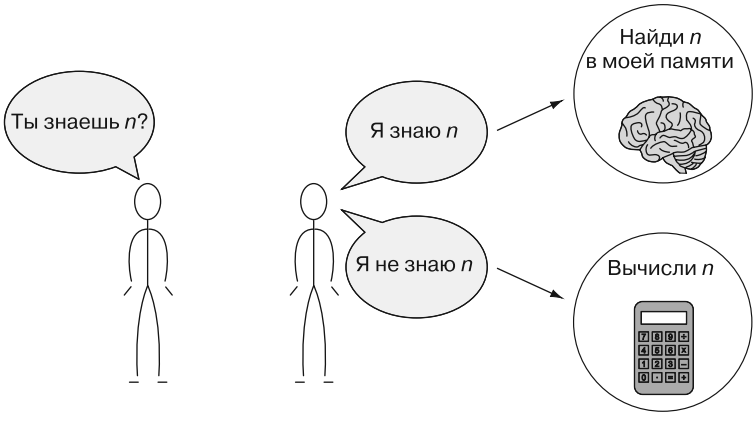


Рис. 1.4. Механизм мемоизации для людей

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

Листинг 1.5. Fib3.java

```
package chapter1;

import java.util.HashMap;
import java.util.Map;

public class Fib3 {

    // Map.of() введена в версии Java 9 и возвращает неизменную карту
```

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

```
    // Создается карта с 0->0 и 1->1,
    // представляющая наши базовые варианты
    static Map<Integer, Integer> memo = new HashMap<>(Map.of(0, 0, 1, 1));

    private static int fib3(int n) {
        if (!memo.containsKey(n)) {
            // шаг мемоизации
            memo.put(n, fib3(n - 1) + fib3(n - 2));
        }
        return memo.get(n);
    }
```

Теперь можно смело вызывать `fib3(40)` (листинг 1.6).

Листинг 1.6. Fib3.java (продолжение)

```
    public static void main(String[] args) {
        System.out.println(fib3(5));
        System.out.println(fib3(40));
    }
}
```

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

1.1.4. Будьте проще, Фибоначчи!

Существует еще более быстрый вариант. Мы можем решить задачу Фибоначчи старым добрым итеративным методом (листинг 1.7).

Листинг 1.7. Fib4.java

```
package chapter1;

public class Fib4 {

    private static int fib4(int n) {
        int last = 0, next = 1; // fib(0), fib(1)
        for (int i = 0; i < n; i++) {
            int oldLast = last;
            last = next;
            next = oldLast + next;
        }
        return last;
    }

    public static void main(String[] args) {
        System.out.println(fib4(20));
        System.out.println(fib4(40));
    }
}
```

Суть в том, что переменной `last` присваивается предыдущее значение `next`, а `next` — предыдущее значение `last` плюс предыдущее значение `next`. Временная переменная `oldLast` облегчает выполнение этих действий.

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

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

1.1.5. Генерация чисел Фибоначчи с помощью потока

До сих пор мы писали методы, которые выводят одно значение из последовательности Фибоначчи. А что, если вместо этого мы хотим вывести всю последовательность вплоть до некоторого значения? Легко преобразовать `fib4()` в поток Java, задействуя шаблон генератора кода. В процессе итерирования генератора на каждой итерации будет выводиться значение последовательности Фибоначчи с использованием лямбда-функции, возвращающей следующее число (листинг 1.8).

Листинг 1.8. Fib5.java

```
package chapter1;

import java.util.stream.IntStream;

public class Fib5 {
    private int last = 0, next = 1; // fib(0), fib(1)

    public IntStream stream() {
        return IntStream.generate(() -> {
            int oldLast = last;
            last = next;
            next = oldLast + next;
            return oldLast;
        });
    }

    public static void main(String[] args) {
        Fib5 fib5 = new Fib5();
        fib5.stream().limit(41).forEachOrdered(System.out::println);
    }
}
```

Если запустить `Fib5.java`, то будут напечатаны первые 41 число из последовательности Фибоначчи. Для каждого числа в последовательности функция `Fib5` запускает один раз лямбда-функцию `generate()`, которая управляет переменными последнего и следующего экземпляров. Вызов функции `limit()` гарантирует, что потенциально бесконечный поток после достижения 41-го элемента перестанет извлекать числа.

1.2. ПРОСТЕЙШЕЕ СЖАТИЕ

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

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

Сжать данные самыми простыми способами можно тогда, когда вы понимаете, что типы хранилищ данных используют больше битов, чем необходимо для их содержимого. Например, на низком уровне, если целые числа со знаком, значения которых никогда не превысят 32 767, сохраняются в памяти как 64-разрядные целые числа — `long`, то они сохраняются неэффективно. Вместо этого их можно хранить как 16-разрядные целые числа — `short`. Это уменьшит потребление пространства для фактического хранения чисел на 75 % (16 бит вместо 64). Таким неэффективным способом хранятся миллионы чисел, — это может означать до 1 Мбайт впустую потраченного пространства.

В Java из соображений простоты (что, конечно же, законная цель) разработчик иногда избавлен от размышлений. В основном в коде для хранения целых чисел применяется 32-битный тип `int`. Это нормально. Но если хранить миллионы

целых чисел или необходимы целые числа определенной точности, то стоит подумать, какой тип использовать.

ПРИМЕЧАНИЕ

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

Если число различных значений, которые способен представлять тип, меньше, чем число значений, которые могут представлять биты, используемые для его хранения, то такой тип может быть сохранен более эффективно. Рассмотрим, к примеру, нуклеотиды, которые образуют ген в ДНК. Каждый нуклеотид может принимать только одно из четырех значений: A, C, G или T. Однако если ген хранится как тип `String`, что можно рассматривать как коллекцию символов Unicode, каждый нуклеотид будет представлен символом, для хранения которого обычно требуется 16 битов (по умолчанию Java использует кодировку UTF-16). В двоичном коде для хранения типа с четырьмя возможными значениями требуется всего 2 бита. Значения 00, 01, 10 и 11 — это четыре различных значения, которые могут быть представлены 2 битами. Если A соответствует 00, C — 01, G — 10, а T — 11, то объем хранилища, необходимого для строки нуклеотидов, может быть уменьшен на 87,5 % (с 16 до 2 битов на каждый нуклеотид).

Вместо того чтобы хранить нуклеотиды как тип `String`, их можно сохранить как строку битов (рис. 1.5). *Строка битов* — это именно то, чем она кажется: последовательность нулей и единиц произвольной длины. К счастью, стандартная библиотека Java содержит готовую конструкцию для работы с битовыми строками произвольной длины, которая называется `BitSet`.

Следующий код (листинг 1.9) преобразует строку, состоящую из букв A, C, G и T, в строку битов и обратно. Строка битов хранится в `BitSet` с помощью метода `compress()`. Для обратного преобразования в строку используется метод `decopress()`.

Листинг 1.9. CompressedGene.java

```
package chapter1;

import java.util.BitSet;

public class CompressedGene {
```

```
    private BitSet bitSet;
    private int length;

    public CompressedGene(String gene) {
        compress(gene);
    }
```

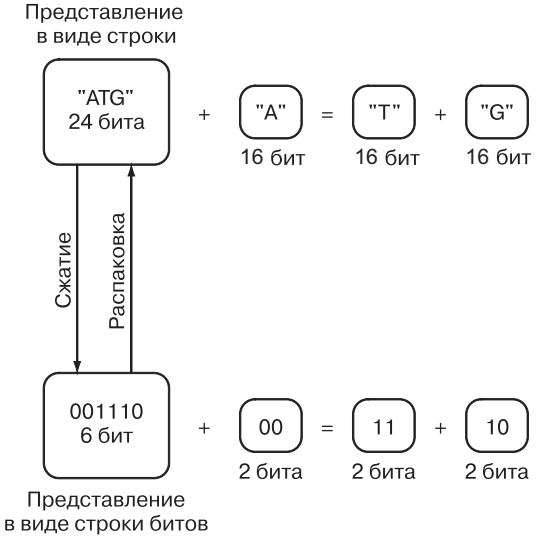


Рис. 1.5. Сжатие типа String, представляющего ген, в строку битов, где выделяется по 2 бита на нуклеотид

В классе `CompressedGene` предоставляется строка символов типа `String`, описывающих нуклеотиды в гене. Внутри класса хранится последовательность нуклеотидов в виде `BitSet`. Основное назначение конструктора состоит в инициализации конструкции `BitSet` соответствующими данными. Конструктор вызывает `compress()`, который выполняет всю грязную работу по фактическому преобразованию предоставленной строки нуклеотидов `String` в `BitSet`.

Теперь посмотрим, как на самом деле можно выполнить сжатие (листинг 1.10).

Листинг 1.10. CompressedGene.java (продолжение)

```
private void compress(String gene) {
    length = gene.length();
    // резервирование пространства для всех битов
    bitSet = new BitSet(length * 2);
    // преобразование в верхний регистр для единообразия
    final String upperGene = gene.toUpperCase();
    // преобразование строки нуклеотидов в биты
```

```
    for (int i = 0; i < length; i++) {
        final int firstLocation = 2 * i;
        final int secondLocation = 2 * i + 1;
        switch (upperGene.charAt(i)) {
        case 'A': // 00 — следующие два бита
            bitSet.set(firstLocation, false);
            bitSet.set(secondLocation, false);
            break;
        case 'C': // 01 — следующие два бита
            bitSet.set(firstLocation, false);
            bitSet.set(secondLocation, true);
            break;
        case 'G': // 10 — следующие два бита
            bitSet.set(firstLocation, true);
            bitSet.set(secondLocation, false);
            break;
        case 'T': // 11 — следующие два бита
            bitSet.set(firstLocation, true);
            bitSet.set(secondLocation, true);
            break;
        default:
            throw new IllegalArgumentException("The provided gene String contains
                                                characters other than ACGT");
        }
    }
}
```

Метод `compress()` последовательно просматривает каждый символ в строке нуклеотидов типа `String`. Встретив символ A, он добавляет в строку битов 00, встретив C — 01, и т. д. Для класса `BitSet` логические значения `true` и `false` соответствуют значениям `1` и `0` соответственно.

Каждый нуклеотид добавляется с помощью двух вызовов метода `set()`. Другими словами, мы постоянно добавляем 2 новых бита в конец строки битов. Два добавляемых бита определяются типом нуклеотида.

Наконец, мы реализуем декомпрессию (листинг 1.11).

Листинг 1.11. CompressedGene.java (продолжение)

```
public String decompress() {
    if (bitSet == null) {
        return "";
    }
    // создание изменяемой позиции для символов в правильном порядке
    StringBuilder builder = new StringBuilder(length);
    for (int i = 0; i < (length * 2); i += 2) {
        final int firstBit = (bitSet.get(i) ? 1 : 0);
        final int secondBit = (bitSet.get(i + 1) ? 1 : 0);
        final int lastBits = firstBit << 1 | secondBit;
        switch (lastBits) {
        case 0b00: // 00 = 'A'
```

```
            builder.append('A');
            break;
        case 0b01: // 01 = 'C'
            builder.append('C');
            break;
        case 0b10: // 10 = 'G'
            builder.append('G');
            break;
        case 0b11: // 11 = 'T'
            builder.append('T');
            break;
        }
    }
    return builder.toString();
}
```

Метод `decompress()` считывает из строки битов по 2 бита и использует их, чтобы определить, какой символ добавить в конец представления гена типа `String`, построенный с помощью `StringBuilder`. Два бита складываются вместе в переменной `lastBits`. Переменная `lastBits` получается в результате сдвига первого бита на одну позицию назад, а затем применения оператора `OR` (оператор `|`) со вторым битом. При использовании оператора `<<` на освободившееся место всегда ставятся нули. Оператор `OR` означает следующее: если любой из этих битов равен 1, поставьте `1`. Таким образом, в результате применения оператора `OR` с `secondBit` и `0` всегда будет получаться значение `secondBit`. Проверим, как это работает (листинг 1.12).

Листинг 1.12. CompressedGene.java (продолжение)

```
    public static void main(String[] args) {
        final String original =
     "TAGGGATTAACCGTTATATATATATAGCCATGGATCGATTATATAGGGATTAACCGTTATATATATATAGCC
     ATGGATCGATTATA";
        CompressedGene compressed = new CompressedGene(original);
        final String decompressed = compressed.decompress();
        System.out.println(decompressed);
        System.out.println("original is the same as decompressed: " +
            original.equalsIgnoreCase(decompressed));
    }

}
```

Метод `main()` выполняет сжатие и распаковку. Он проверяет, совпадает ли окончательный результат с исходной строкой, используя метод `equalsIgnoreCase()` (листинг 1.13).

Листинг 1.13. CompressedGene.java (вывод)

```
TAGGGATTAACCGTTATATATATATAGCCATGGATCGATTATATAGGGATTAACCGTTATATATATATAGCCA
TGGATCGATTATA
original is the same as decompressed: true
```

1.3. НЕВСКРЫВАЕМОЕ ШИФРОВАНИЕ

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

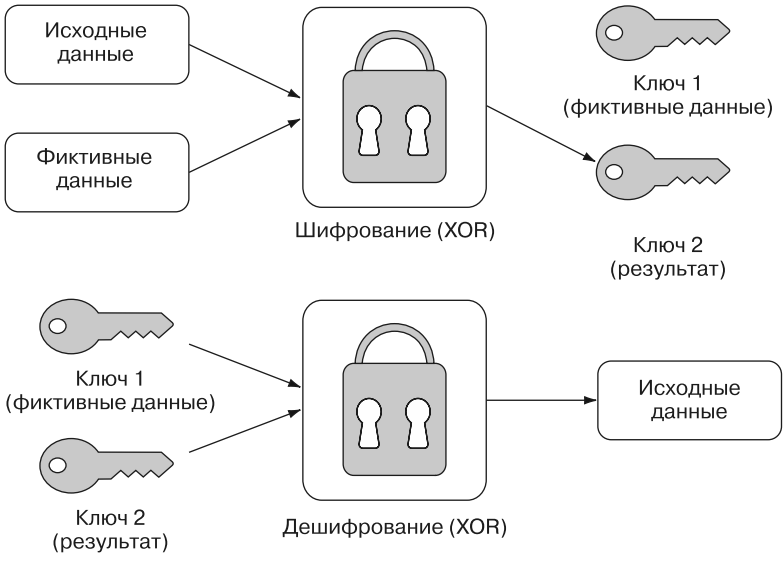


Рис. 1.6. Одноразовый шифр приводит к созданию двух ключей, которые можно разделить, а затем объединить для восстановления исходных данных

1.3.1. Получение данных в заданной последовательности

В этом примере мы зашифруем строку `String`, используя одноразовый шифр. Один из способов представления типа `String` в Java — это последовательность байтов UTF-16 (UTF-16 — это кодировка символов Unicode). Каждый символ UTF-16 состоит из 16 бит (отсюда и 16) и может быть дополнительно разбит на 2 байта (по 8 бит каждый). Строку можно преобразовать в массив байтов, представленный как массив `byte`, с помощью метода `getBytes()`. Точно так же массив байтов можно преобразовать обратно в строку, используя один из встроенных

конструкторов типа `String`. Нам понадобится промежуточная форма для хранения пары ключей, которая будет состоять из двух массивов байтов. Это цель работы класса `KeyPair` (листинг 1.14).

Листинг 1.14. KeyPair.java

```
package chapter1;

public final class KeyPair {
    public final byte[] key1;
    public final byte[] key2;
    KeyPair(byte[] key1, byte[] key2) {
        this.key1 = key1;
        this.key2 = key2;
    }
}
```

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

В этом примере задействуем метод генерации псевдослучайных данных `nextBytes()` (включен в стандартную библиотеку, в класс `Random`). Наши данные не будут действительно случайными в том смысле, что внутри класса `Random` все еще применяется генератор псевдослучайных чисел, но для наших целей он будет достаточно близок к случайному. Сгенерируем случайный ключ для использования в качестве фиктивных данных (листинг 1.15).

Листинг 1.15. UnbreakableEncryption.java

```
package chapter1;

import java.util.Random;

public class UnbreakableEncryption {
    // генерировать length случайных байтов
    private static byte[] randomKey(int length) {
        byte[] dummy = new byte[length];
        Random random = new Random();
        random.nextBytes(dummy);
        return dummy;
    }
```

Этот метод создает массив байтов, содержащий байты случайной длины. В конечном итоге байты будут служить фиктивным ключом в нашей ключевой паре.

1.3.2. Шифрование и дешифрование

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

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

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

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

Листинг 1.16. UnbreakableEncryption.java (продолжение)

```
public static KeyPair encrypt(String original) {
    byte[] originalBytes = original.getBytes();
    byte[] dummyKey = randomKey(originalBytes.length);
    byte[] encryptedKey = new byte[originalBytes.length];
    for (int i = 0; i < originalBytes.length; i++) {
        // Побитовая операция XOR
        encryptedKey[i] = (byte) (originalBytes[i] ^ dummyKey[i]);
    }
    return new KeyPair(dummyKey, encryptedKey);
}
```

Расшифровка — это просто рекомбинация пары ключей, сгенерированной с помощью `encrypt()`. Она достигается посредством повторного выполнения операции XOR побитово для двух ключей. Окончательный результат должен быть преобразован обратно в тип `String`. Это достигается с помощью конструктора

из класса `String`, который принимает массив байтов в качестве единственного аргумента (листинг 1.17).

Листинг 1.17. UnbreakableEncryption.java (продолжение)

```
public static String decrypt(KeyPair kp) {
    byte[] decrypted = new byte[kp.key1.length];
    for (int i = 0; i < kp.key1.length; i++) {
        // Побитовая операция XOR
        decrypted[i] = (byte) (kp.key1[i] ^ kp.key2[i]);
    }
    return new String(decrypted);
}
```

Если наше шифрование с одноразовыми ключами работает, то мы должны шифровать и дешифровать одну и ту же строку Unicode без проблем (листинг 1.18).

Листинг 1.18. UnbreakableEncryption.java (продолжение)

```
    public static void main(String[] args) {
        KeyPair kp = encrypt("One Time Pad!");
        String result = decrypt(kp);
        System.out.println(result);
    }
}
```

Если в консоль выводится фраза One Time Pad!, значит, все заработало. Протестируйте собственные значения.

1.4. ВЫЧИСЛЕНИЕ ЧИСЛА π

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

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

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

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

Листинг 1.19. PiCalculator.java

```
package chapter1;

public class PiCalculator {

    public static double calculatePi(int nTerms) {
        final double numerator = 4.0;
        double denominator = 1.0;
        double operation = 1.0;
        double pi = 0.0;
        for (int i = 0; i < nTerms; i++) {
            pi += operation * (numerator / denominator);
            denominator += 2.0;
            operation *= -1.0;
        }
        return pi;
    }

    public static void main(String[] args) {
        System.out.println(calculatePi(1000000));
    }
}
```

СОВЕТ

На большинстве платформ значения float в Java являются 64-битными числами с плавающей точкой и предполагают бóльшую точность, чем 32-битные числа с плавающей точкой.

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

ПРИМЕЧАНИЕ

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

1.5. ХАНОЙСКИЕ БАШНИ

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

под номером 2, располагался бы над диском 1. Наконец, самый узкий диск, под номером 3, лежал бы на диске 2.

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

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

Схематически задача показана на рис. 1.7.

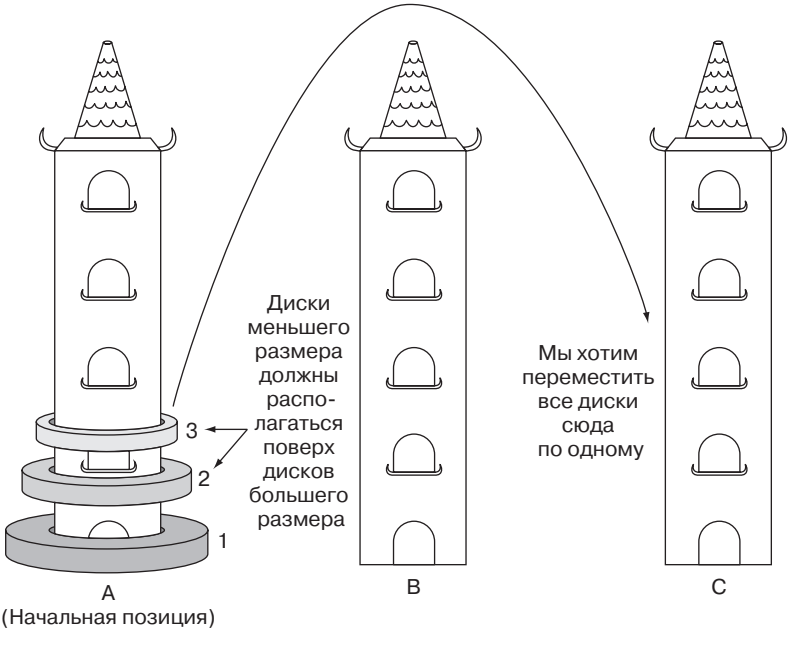


Рис. 1.7. Задача состоит в том, чтобы переместить три диска по одному с башни A на башню C. Диск большего размера никогда не может располагаться поверх диска меньшего размера

1.5.1. Моделирование башен

Стек — это структура данных, смоделированная по принципу «последним пришел — первым вышел» (last-in-first-out, LIFO). То, что попадает в стек последним, становится первым, что оттуда извлекается. Представьте себя на месте учителя,

оценивающего стопку работ. Последняя работа, находящаяся на самом верху стопки, — это первый лист, который учитель берет для проверки. Две основные операции стека — это `push` (поместить) и `pop` (извлечь). Операция `push` помещает новый элемент в стек, а `pop` удаляет из стека и возвращает последний вставленный элемент. Стандартная библиотека Java содержит встроенный класс `Stack`, который включает методы для `push()` и `pop()`.

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

Определим башни как объекты `Stack` и заполним первую из них дисками (листинг 1.20).

Листинг 1.20. Hanoi.java

```
package chapter1;

import java.util.Stack;

public class Hanoi {
    private final int numDiscs;
    public final Stack<Integer> towerA = new Stack<>();
    public final Stack<Integer> towerB = new Stack<>();
    public final Stack<Integer> towerC = new Stack<>();

    public Hanoi(int discs) {
        numDiscs = discs;
        for (int i = 1; i <= discs; i++) {
            towerA.push(i);
        }
    }
```

1.5.2. Решение задачи о ханойских башнях

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

Чтобы понять рекурсивный случай, рассмотрим конкретный пример. Предположим, у нас есть три диска — верхний, средний и нижний, расположенные на башне A, и мы хотим переместить их на башню C. (Впоследствии это поможет

схематически описать задачу.) Сначала мы могли бы переместить верхний диск на башню C. Затем — переместить средний диск на башню B, а после этого — верхний диск с башни C на башню B. Теперь у нас есть нижний диск, все еще расположенный на башне A, и два верхних диска на башне B. По существу, мы уже успешно переместили два диска с одной башни (A) на другую (B). Перемещение нижнего диска с A на C — это базовый случай (перемещение одного диска). Теперь можем переместить два верхних диска с B на C посредством той же процедуры, что и с A на B. Перемещаем верхний диск на A, средний — на C и, наконец, верхний — с A на C.

СОВЕТ

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

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

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

Удивительно, но этот рекурсивный алгоритм работает не только для трех, но и для любого количества дисков. Закодируем его как функцию `move()`, которая отвечает за перемещение дисков с одной башни на другую, задействуя третью временную башню (листинг 1.21).

Листинг 1.21. Hanoi.java (продолжение)

```
private void move(Stack<Integer> begin, Stack<Integer> end, Stack<Integer> temp,
int n) {
    if (n == 1) {
        end.push(begin.pop());
    } else {
        move(begin, temp, end, n – 1);
        move(begin, end, temp, 1);
        move(temp, end, begin, n – 1);
    }
}
```

Наконец, для перемещения всех дисков с башни A на башню C вспомогательная функция `resolve()` вызовет функцию `move()`. После вызова `resolve()` необходимо

проверить башни A, B и C, чтобы убедиться, что диски были успешно перемещены (листинг 1.22).

Листинг 1.22. Hanoi.java (продолжение)

```
    public void solve() {
        move(towerA, towerC, towerB, numDiscs);
    }

    public static void main(String[] args) {
        Hanoi hanoi = new Hanoi(3);
        hanoi.solve();
        System.out.println(hanoi.towerA);
        System.out.println(hanoi.towerB);
        System.out.println(hanoi.towerC);
    }
}
```

Вы обнаружите, что диски действительно были перемещены. При кодировании решения задачи о ханойских башнях не обязательно понимать каждый шаг, необходимый для перемещения нескольких дисков с башни A на башню C. Мы пришли к пониманию общего рекурсивного алгоритма перемещения любого количества дисков и систематизировали его, позволяя компьютеру сделать все остальное. В этом и заключается сила формулирования рекурсивных решений задач: мы зачастую можем представлять себе решения абстрактно, не тратя силы на мысленное представление каждого отдельного действия.

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

1.6. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

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

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

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

Сжатие сделало мир, подключенный к интернету с его ограниченной пропускной способностью, более терпимым. Метод битовых строк, рассмотренный в разделе 1.2, применим для простых типов данных в реальном мире, имеющих ограниченное число возможных значений, для которых даже 1 байт оказывается избыточным. Однако большинство алгоритмов сжатия работают путем поиска шаблонов или структур в наборе данных, которые позволяют устранить повторяющуюся информацию. Они значительно сложнее, чем описано в разделе 1.2.

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

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

1.7. УПРАЖНЕНИЯ

1. Напишите еще одну функцию, которая вычисляет *n*-й элемент последовательности Фибоначчи, используя метод вашей собственной разработки. Напишите модульные тесты, которые оценивали бы правильность этой функции и ее

производительность по сравнению с другими версиями, представленными в этой главе.

2. Класс `BitSet` в стандартной библиотеке Java имеет недостаток: хотя он отслеживает количество битов, установленное в значение `true`, но не отслеживает общее количество битов, включая биты, установленные в значение `false` (поэтому нам понадобилась переменная экземпляра длины). Напишите эргономичный подкласс `BitSet`, который будет точно отслеживать количество битов, установленное в значение `true` или `false`. Переопределите `CompressedGene` с помощью этого подкласса.
3. Напишите программу решения задачи о ханойских башнях, которая работала бы для любого количества башен.
4. Воспользуйтесь одноразовым шифром для шифрования и дешифровки изображений.

2 Задачи поиска

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

2.1. ПОИСК ДНК

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

2.1.1. Хранение ДНК

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

Листинг 2.1. Gene.java

```
package chapter2;

import java.util.ArrayList;
import java.util.Collections;
```

```
import java.util.Comparator;

public class Gene {

    public enum Nucleotide {
        A, C, G, T
    }
```

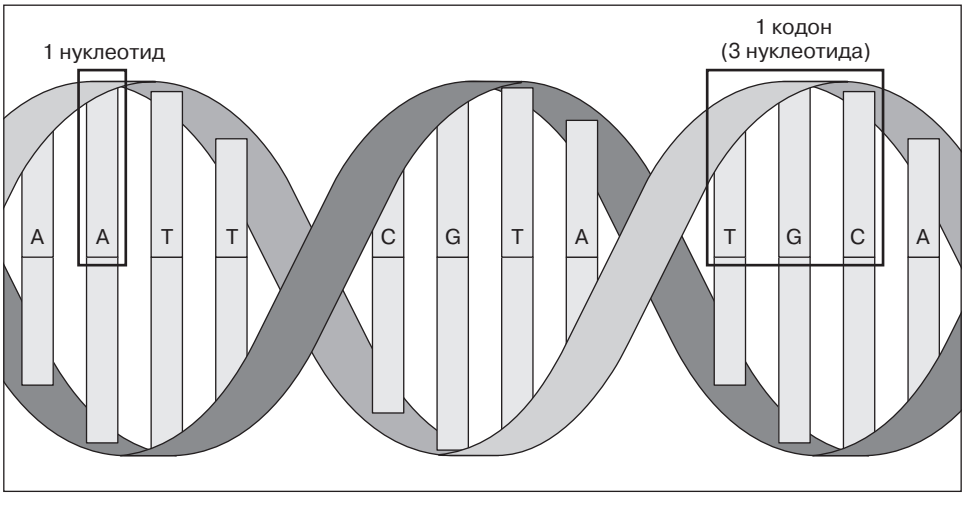


Рис. 2.1. Нуклеотид представлен буквой A, C, G или T. Кодон состоит из трех нуклеотидов, а ген — из нескольких кодонов

Кодоны состоят из трех нуклеотидов. Конструктор класса `Codon` преобразует строку из трех букв в `Codon`. Чтобы реализовать методы поиска, необходимо иметь возможность сравнивать один кодон с другим. Интерфейс `Comparable` содержит метод, который сравнивает один объект с другим.

Для реализации интерфейса `Comparable` необходимо определить один метод `compareTo()`.

Метод `compareTo()` возвращает:

- отрицательное число, если первый объект меньше второго (сравниваемого объекта);
- `0`, если два объекта равны;
- положительное число, если первый объект больше второго (сравниваемого объекта).

На практике обычно можно избежать реализации этого вручную и взамен использовать встроенный интерфейс стандартной библиотеки Java `Comparator`, как показано в листинге 2.2. В данном примере объект `Codon` сначала будет сравниваться с другим объектом `Codon` по его первому нуклеотиду, затем по второму, если первые эквивалентны, и, наконец, по третьему, если вторые эквивалентны. Объекты связаны с помощью метода `thenComparing()`.

Листинг 2.2. Gene.java (продолжение)

```
    public static class Codon implements Comparable<Codon> {
        public final Nucleotide first, second, third;
        private final Comparator<Codon> comparator =
            Comparator.comparing((Codon c) -> c.first)
                .thenComparing((Codon c) -> c.second)
                .thenComparing((Codon c) -> c.third);

        public Codon(String codonStr) {
            first = Nucleotide.valueOf(codonStr.substring(0, 1));
            second = Nucleotide.valueOf(codonStr.substring(1, 2));
            third = Nucleotide.valueOf(codonStr.substring(2, 3));
    }

    @Override
    public int compareTo(Codon other) {
        // сначала сравнивается первый объект, затем второй и т. д.
        // другими словами, первый объект имеет приоритет перед
        // вторым, а второй — перед третьим
        return comparator.compare(this, other);
    }
}
```

ПРИМЕЧАНИЕ

Codon — это статический класс. Вложенные классы, определенные как статические, могут быть созданы безотносительно к их включающему классу (вам не нужен экземпляр включающего класса для создания экземпляра статического вложенного класса), но они не могут ссылаться ни на одну из переменных экземпляра их включающего класса. Это имеет смысл для классов, которые определены как вложенные, и в первую очередь для организационных, а не логистических целей.

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

Листинг 2.3. Пример geneStr (продолжение)

```
String geneStr = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT";
```

Единственное состояние `Gene` — это `ArrayList` кодонов `Codon`. Также понадобится конструктор для преобразования `geneStr` в `Gene` (преобразование строки в `ArrayList` кодонов) (листинг 2.4).

Листинг 2.4. Gene.java (продолжение)

```
private ArrayList<Codon> codons = new ArrayList<>();

public Gene(String geneStr) {
    for (int i = 0; i < geneStr.length() - 3; i += 3) {
        // Из каждых трех символов в строке сформируйте кодон
        codons.add(new Codon(geneStr.substring(i, i + 3)));
    }
}
```

Данный конструктор постоянно проходит строку и преобразует каждые три символа в объекты `Codon`, которые добавляет в конец нового объекта `Gene`. В данном случае для преобразования трехсимвольной строки `String` в `Codon` используется конструктор `Codon`.

2.1.2. Линейный поиск

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

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

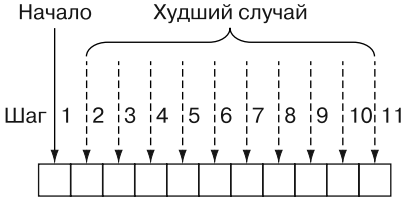


Рис. 2.2. В худшем случае при линейном поиске нужно последовательно просмотреть каждый элемент массива

Определить функцию, которая выполняет линейный поиск, очень легко. Она просто должна перебрать все элементы структуры данных и проверить каждый

из них на эквивалентность искомому элементу. Для этого в `main()` используется следующий код (листинг 2.5).

Листинг 2.5. Gene.java (продолжение)

```
    public boolean linearContains(Codon key) {
        for (Codon codon : codons) {
            if (codon.compareTo(key) == 0) {
                return true; // поиск совпадений
            }
        }
        return false;
    }

    public static void main(String[] args) {
        String geneStr = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGA
                         CTCCCTTT";
        Gene myGene = new Gene(geneStr);
        Codon acg = new Codon("ACG");
        Codon gat = new Codon("GAT");
        System.out.println(myGene.linearContains(acg)); // true
        System.out.println(myGene.linearContains(gat)); // false
    }
}
```

ПРИМЕЧАНИЕ

Эта функция имеет демонстрационное назначение. Все классы стандартной библиотеки Java, реализующие интерфейс Collection (например, ArrayList и LinkedList), содержат метод contains(), который, вероятно, будет реализован лучше, чем все, что мы пишем.

2.1.3. Бинарный поиск

Существует более быстрый способ поиска, чем просмотр всех элементов. Но при этом требуется заранее знать кое-что о последовательности структуры данных. Если известно, что структура отсортирована и мы можем мгновенно получить доступ к любому ее элементу по его индексу, то можно выполнить бинарный поиск.

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

Предположим, что у нас есть список отсортированных по алфавиту слов (`cat` (кошка), `dog` (собака), `kangaroo` (кенгуру), `llama` (лама), `rabbit` (кролик), `rat` (крыса), `zebra` (зебра)) и мы ищем слово `rat` (крыса).

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

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

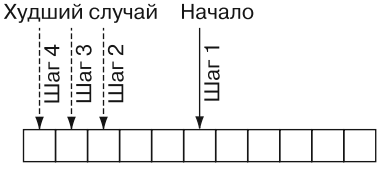


Рис. 2.3. В худшем случае при бинарном поиске придется просматривать только lg(n) элементов списка

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

Написание функции бинарного поиска для гена и кодона мало отличается от написания функции для любого другого типа данных, поскольку объекты типа `Codon` можно сравнивать между собой, а тип `Gene` — это просто список `ArrayList` (листинг 2.6). Обратите внимание на то, что в следующем примере мы в первую очередь выполняем сортировку кодонов. Это устраняет все преимущества выполнения бинарного поиска, так как сортировка займет больше времени,

чем поиск. Однако в демонстрационных целях она необходима, поскольку при запуске данного примера мы не можем знать, что кодоны в списке `ArrayList` отсортированы.

Листинг 2.6. Gene.java (продолжение)

```
public boolean binaryContains(Codon key) {
    // бинарный поиск работает только с отсортированными коллекциями
    ArrayList<Codon> sortedCodons = new ArrayList<>(codons);
    Collections.sort(sortedCodons);
    int low = 0;
    int high = sortedCodons.size() - 1;
    while (low <= high) { // while there is still a search space
        int middle = (low + high) / 2;
        int comparison = codons.get(middle).compareTo(key);
        if (comparison < 0) { // средний кодон меньше искомого
            low = middle + 1;
        } else if (comparison > 0) { // средний кодон больше искомого
            high = middle - 1;
        } else { // средний кодон равен искомому
            return true;
        }
    }
    return false;
}
```

Рассмотрим выполнение этой функции построчно:

```
int low = 0;
int high = sortedCodons.size() - 1;
```

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

```
while (low <= high) {
```

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

```
int middle = (low + high) / 2;
```

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

```
int comparison = codons.get(middle).compareTo(key);
if (comparison < 0) { // средний кодон меньше искомого
    low = middle + 1;
```

Если искомый элемент находится после среднего элемента текущего диапазона, то изменяем диапазон, который будем рассматривать на следующей итерации

цикла, перемещая `low` на следующий после текущего среднего элемента. Именно на этом шаге мы вдвое сокращаем диапазон поиска для следующей итерации:

```
} else if (comparison > 0) { // средний кодон больше искомого
    high = middle – 1;
```

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

```
} else { // средний кодон меньше искомого
    return true;
}
```

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

Мы можем попробовать запустить функцию с теми же геном и кодоном (листинг 2.7). Для этого можем изменить `main()`.

Листинг 2.7. Gene.java (продолжение)

```
public static void main(String[] args) {
    String geneStr = "ACGTGGCTCTCTAACGTACGTACGTACGGGGTTTATATATACCCTAGGACTCCCTTT";
    Gene myGene = new Gene(geneStr);
    Codon acg = new Codon("ACG");
    Codon gat = new Codon("GAT");
    System.out.println(myGene.linearContains(acg)); // true
    System.out.println(myGene.linearContains(gat)); // false
    System.out.println(myGene.binaryContains(acg)); // true
    System.out.println(myGene.binaryContains(gat)); // false
}
```

СОВЕТ

Как и в случае с линейным поиском, вам никогда не придется самостоятельно реализовывать бинарный поиск, потому что такая реализация содержится в стандартной библиотеке Java. Для бинарного поиска можно воспользоваться функцией Collections.binarySearch(), способной выполнять поиск в любом отсортированном списке (например, ArrayList).

2.1.4. Параметризованный пример

Методы `linearContains()` и `binaryContains()` можно обобщить для работы практически с любой последовательностью Java. Следующие обобщенные версии почти идентичны версиям, которые вы видели ранее, изменились лишь некоторые имена и аннотации типов.

ПРИМЕЧАНИЕ

В листинге 2.8 задействовано много импортированных типов. В этой главе мы еще раз воспользуемся файлом GenericSearch.java для многих дополнительных универсальных алгоритмов поиска, что избавит нас от необходимости импорта.

Листинг 2.8. GenericSearch.java

```
package chapter2;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.Stack;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.ToDoubleFunction;

public class GenericSearch {

    public static <T extends Comparable<T>> boolean linearContains(List<T>
      list, T key) {
        for (T item : list) {
            if (item.compareTo(key) == 0) {
                return true; // поиск совпадений
            }
        }
        return false;
    }

    // предполагается, что *список* уже отсортирован
    public static <T extends Comparable<T>> boolean binaryContains(List<T>
     list, T key) {
        int low = 0;
        int high = list.size() - 1;
        while (low <= high) { // пока еще есть место для поиска
            int middle = (low + high) / 2;
            int comparison = list.get(middle).compareTo(key);
            if (comparison < 0) { // средний кодон меньше искомого
                low = middle + 1;
            } else if (comparison > 0) { // средний кодон больше искомого
                high = middle - 1;
            } else { // средний кодон равен искомому
                return true;
            }
        }
        return false;
```

```
    }

    public static void main(String[] args) {
        System.out.println(linearContains(List.of(1, 5, 15, 15, 15, 15, 20),
            5)); // true
        System.out.println(binaryContains(List.of("a", "d", "e", "f", "z"),
            "f")); // true
        System.out.println(binaryContains(List.of("john", "mark", "ronald",
            "sarah"), "sheila")); // false
    }
}
```

ПРИМЕЧАНИЕ

Ключевое слово extends в коде T extends Comparable<T> означает, что T должен быть типом, реализующим интерфейс Comparable.

Теперь вы можете попробовать выполнить поиск для других типов данных. Эти методы можно использовать практически для любой коллекции Java. В этом преимущество общего написания кода.

2.2. ПРОХОЖДЕНИЕ ЛАБИРИНТА

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

Наш лабиринт представляет собой двумерную сеть ячеек — объектов `Cell` (листинг 2.9). `Cell` — это перечисление значений `String`, где `" "` означает пустое пространство, а `"X"` — занятое. В иллюстративных целях при выводе лабиринта на печать существуют и другие варианты заполнения ячеек.

Листинг 2.9. Maze.java

```
package chapter2;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import chapter2.GenericSearch.Node;

public class Maze {

    public enum Cell {
        EMPTY(" "),
        BLOCKED("X"),
        START("S"),
```

```
        GOAL("G"),
        PATH("*");

        private final String code;

        private Cell(String c) {
            code = c;
        }

        @Override
        public String toString() {
            return code;
        }
    }
```

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

Нам понадобится способ сослаться на конкретную точку в лабиринте. Это будет просто `MazeLocation` со свойствами, представляющими строку и столбец, к которым относится данная ячейка в сетке (листинг 2.10). Однако также необходимо внутри класса сравнить его экземпляры с другими экземплярами того же типа. Это нужно для правильного использования нескольких классов в структуре коллекций, таких как `HashSet` и `HashMap`. Чтобы избежать дубликатов, поскольку классы допускают только уникальные экземпляры, применяются методы `equals()` и `hashCode()`.

К счастью, IDE могут выполнить за нас сложную работу. Два метода, следующие за конструктором в листинге 2.10, были автоматически созданы с помощью `Eclipse`. Они гарантируют, что два экземпляра `MazeLocation` с одними и теми же строкой и столбцом будут рассматриваться как эквивалентные. Для создания этих методов вы можете в `Eclipse` щелкнуть правой кнопкой мыши и выбрать пункт меню Source ▸ Generate hashCode() and equals() (Источник ▸ Создать hashCode() и equals()). В диалоговом окне необходимо указать, какие переменные экземпляра используются для выполнения равенства.

Листинг 2.10. Maze.java (продолжение)

```
public static class MazeLocation {
    public final int row;
    public final int column;

    public MazeLocation(int row, int column) {
        this.row = row;
        this.column = column;
    }
```

```
    // автоматически создается с помощью Eclipse
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + column;
        result = prime * result + row;
        return result;
    }

    // автоматически создается с помощью Eclipse
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        MazeLocation other = (MazeLocation) obj;
        if (column != other.column) {
            return false;
        }
        if (row != other.row) {
            return false;
        }
        return true;
    }
}
```

2.2.1. Создание случайного лабиринта

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

Сгенерированный лабиринт должен быть достаточно разреженным, чтобы почти всегда существовал путь от заданного начального до конечного местоположения. (В конце концов, это нужно для тестирования наших алгоритмов.) Мы позволим функции, вызывающей новый лабиринт, точно задавать степень разреженности, но определим значение по умолчанию, составляющее 20 % заблокированных ячеек. Если случайное число превысит порог, заданный параметром `sparseness`, просто заменим пустое пространство стеной. Если так поступать для каждой возможной точки лабиринта, то статистически разреженность лабиринта в целом будет приближаться к заданному параметру `sparseness` (листинг 2.11).

Листинг 2.11. Maze.java (продолжение)

```
private final int rows, columns;
private final MazeLocation start, goal;
private Cell[][] grid;

public Maze(int rows, int columns, MazeLocation start, MazeLocation goal,
     double sparseness) {
    // инициализация базовых переменных экземпляра
    this.rows = rows;
    this.columns = columns;
    this.start = start;
    this.goal = goal;
    // заполнение сетки пустыми ячейками
    grid = new Cell[rows][columns];
    for (Cell[] row : grid) {
        Arrays.fill(row, Cell.EMPTY);
    }
    // заполнение сетки заблокированными ячейками
    randomlyFill(sparseness);
    // заполнение начальной и конечной позиций в лабиринте
    grid[start.row][start.column] = Cell.START;
    grid[goal.row][goal.column] = Cell.GOAL;
}

public Maze() {
    this(10, 10, new MazeLocation(0, 0), new MazeLocation(9, 9), 0.2);
}

private void randomlyFill(double sparseness) {
    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < columns; column++) {
            if (Math.random() < sparseness) {
                grid[row][column] = Cell.BLOCKED;
            }
        }
    }
}
```

Теперь, когда у нас есть лабиринт, нужно, чтобы он кратко выводился в консоль. Мы хотим, чтобы элементы лабиринта располагались близко друг к другу и все было похоже на настоящий лабиринт (листинг 2.12).

Листинг 2.12. Maze.java (продолжение)

```
// вывести красиво отформатированную версию лабиринта для печати
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (Cell[] row : grid) {
        for (Cell cell : row) {
            sb.append(cell.toString());
        }
        sb.append(System.lineSeparator());
    }
    return sb.toString();
}
```

Теперь протестируем эти функции лабиринта в `main()` (листинг 2.13).

Листинг 2.13. Maze.java (продолжение)

```
    public static void main(String[] args) {
        Maze m = new Maze();
        System.out.println(m);
    }
}
```

2.2.2. Мелкие детали лабиринта

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

Листинг 2.14. Maze.java (продолжение)

```
public boolean goalTest(MazeLocation ml) {
    return goal.equals(ml);
}
```

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

Листинг 2.15. Maze.java (продолжение)

```
public List<MazeLocation> successors(MazeLocation ml) {
    List<MazeLocation> locations = new ArrayList<>();
    if (ml.row + 1 < rows && grid[ml.row + 1][ml.column] != Cell.BLOCKED) {
        locations.add(new MazeLocation(ml.row + 1, ml.column));
    }
    if (ml.row - 1 >= 0 && grid[ml.row - 1][ml.column] != Cell.BLOCKED) {
        locations.add(new MazeLocation(ml.row - 1, ml.column));
    }
    if (ml.column + 1 < columns && grid[ml.row][ml.column + 1] != Cell.BLOCKED) {
        locations.add(new MazeLocation(ml.row, ml.column + 1));
    }
    if (ml.column - 1 >= 0 && grid[ml.row][ml.column - 1] != Cell.BLOCKED) {
        locations.add(new MazeLocation(ml.row, ml.column - 1));
    }
    return locations;
}
```

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

2.2.3. Поиск в глубину

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

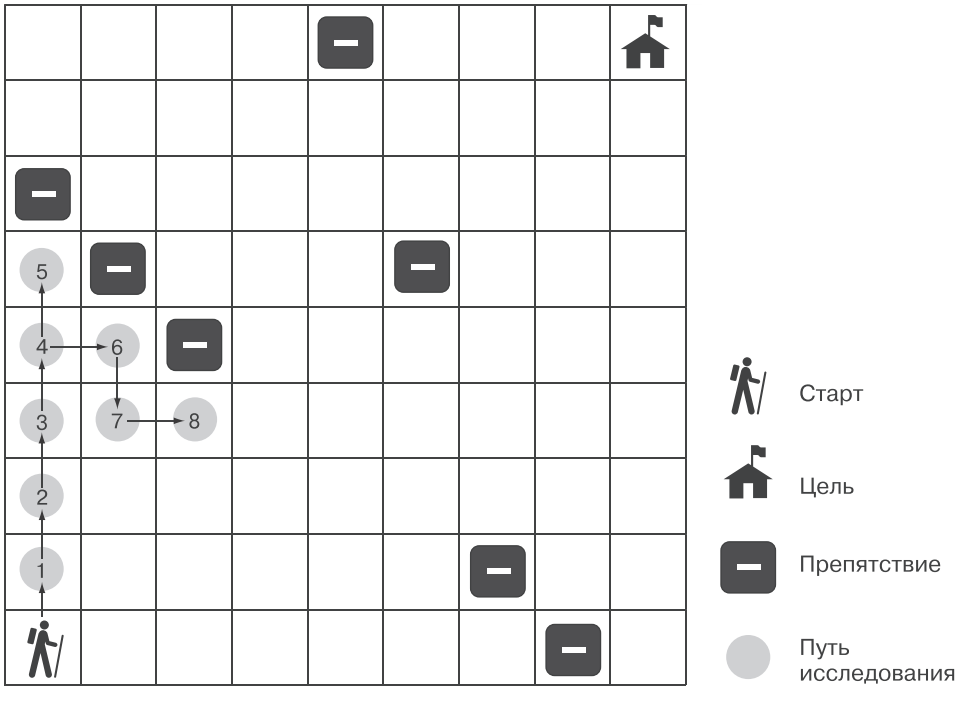


Рис. 2.4. Поиск в глубину проходит по непрерывному пути вглубь, пока не дойдет до препятствия и не будет вынужден вернуться к последней точке принятия решения

Стеки

Алгоритм поиска в глубину опирается на структуру данных, известную как *стек*. (Если вы читали о стеках в главе 1, можете спокойно пропустить этот раздел.) Стек — это структура данных, которая работает по принципу «последним пришел — первым вышел» (last-in-first-out, LIFO). Стек можно представить как стопку документов. Последний документ, помещенный сверху стопки, будет первым, который оттуда извлекут. Обычно стек реализуется на основе более примитивной структуры данных, такой как список (добавление элементов в начало списка и удаление элементов в его конце). Мы могли бы легко реализовать стек самостоятельно, но стандартная библиотека Java содержит удобный класс `Stack`.

Обычно стек имеет минимум две операции:

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

Другими словами, стек — это метаструктура, обеспечивающая порядок удаления в списке. Последний элемент, помещенный в стопку, должен быть следующим элементом, удаленным из стопки.

Алгоритм DFS

Понадобится учесть еще один момент, прежде чем можно будет приступить к реализации DFS. Требуется класс `Node`, с помощью которого мы станем отслеживать переход из одного состояния в другое (или из одного места лабиринта в другое) во время поиска. `Node` можно представить как обертку вокруг состояния. В случае прохождения лабиринта эти состояния имеют тип `MazeLocation`. Будем считать `Node` состоянием, унаследованным от `parent`. Кроме того, для класса `Node` мы определим свойства `cost` и `heuristic`, чтобы использовать его позже в алгоритме A* (листинг 2.16). Пока не беспокойтесь. `Класс Node` становится `Comparable` путем сравнения комбинации `его` свойств `cost` и `heuristic`.

Листинг 2.16. GenericSearch.java (продолжение)

```
public static class Node<T> implements Comparable<Node<T>> {
    final T state;
    Node<T> parent;
    double cost;
    double heuristic;

    // для dfs и bfs мы не будем использовать свойства cost и heuristic
    Node(T state, Node<T> parent) {
        this.state = state;
        this.parent = parent;
    }
```

```
    // для astar мы будем использовать свойства cost и heuristic
    Node(T state, Node<T> parent, double cost, double heuristic) {
        this.state = state;
        this.parent = parent;
        this.cost = cost;
        this.heuristic = heuristic;
    }

    @Override
    public int compareTo(Node<T> other) {
        Double mine = cost + heuristic;
        Double theirs = other.cost + other.heuristic;
        return mine.compareTo(theirs);
    }
}
```

СОВЕТ

В данном случае compareTo() вызывает compareTo() для другого типа. Это обычная закономерность.

ПРИМЕЧАНИЕ

Если объект Node не имеет свойства parent, в качестве указателя будем использовать null.

В процессе поиска в глубину нужно отслеживать две структуры данных: стек рассматриваемых состояний (мест), который мы назовем `frontier`, и набор уже просмотренных состояний — `explored`. До тех пор, пока в `frontier` остаются состояния, DFS будет продолжать проверять, являются ли они целью поиска (найдя искомое состояние, DFS остановит поиск и возвратит его), и добавлять их наследников в `frontier`. Также алгоритм будет помечать все просмотренные состояния как `explored`, чтобы поиск не ходил по кругу и уже изученные состояния не становились наследниками. Если `frontier` окажется пустым, это будет означать, что объектов для поиска не осталось (листинг 2.17).

Листинг 2.17. GenericSearch.java (продолжение)

```
public static <T> Node<T> dfs(T initial, Predicate<T> goalTest,
        Function<T, List<T>> successors) {
    // frontier — то, что нам нужно проверить
    Stack<Node<T>> frontier = new Stack<>();
    frontier.push(new Node<>(initial, null));
    // explored — то, где мы уже
    Set<T> explored = new HashSet<>();
    explored.add(initial);

    // продолжаем, пока есть что просматривать
    while (!frontier.isEmpty()) {
        Node<T> currentNode = frontier.pop();
```

```
        T currentState = currentNode.state;
        // если мы нашли искомое, заканчиваем
        if (goalTest.test(currentState)) {
            return currentNode;
        }
        // проверяем, куда можно двинуться дальше и что мы еще не исследовали
        for (T child : successors.apply(currentState)) {
            if (explored.contains(child)) {
                continue; // пропустить состояния, которые уже исследовали
            }
            explored.add(child);
            frontier.push(new Node<>(child, currentNode));
        }
    }
    return null; // все проверили, пути к целевой точке не нашли
}
```

Обратите внимание на ссылки на функции `goalTest` и `successors`. Это позволяет подключать различные функции к `dfs()` для разных приложений, что делает `dfs()` пригодным для большего количества сценариев, чем просто лабиринты. Это еще один пример общего решения проблемы. `GoalTest`, будучи `Predicate<T>`, — это любая функция, которая принимает параметр `T` (в нашем случае `MazeLocation`) и возвращает логическое значение. `Successors` — это любая функция, которая принимает параметр `T` и возвращает список `T`.

Если `dfs()` завершается успешно, то возвращается `Node`, в котором инкапсулировано искомое состояние. Для того чтобы восстановить путь от начала до целевой ячейки, нужно двигаться в обратном направлении от этого `Node` к его предкам, используя свойство `parent` (листинг 2.18).

Листинг 2.18. GenericSearch.java (продолжение)

```
public static <T> List<T> nodeToPath(Node<T> node) {
    List<T> path = new ArrayList<>();
    path.add(node.state);
    // двигаемся назад, от конца к началу
    while (node.parent != null) {
        node = node.parent;
        path.add(0, node.state); // добавить в начало
    }
    return path;
}
```

При отображении полезно будет разметить лабиринт, указав успешный путь, а также начальное и конечное положения. Хорошо иметь и возможность удалить путь, чтобы можно было применить разные алгоритмы поиска для одного и того же лабиринта. Для этого в файле `Maze.java` нужно добавить в класс `Maze` следующие два метода (листинг 2.19).

Листинг 2.19. Maze.java (продолжение)

```
public void mark(List<MazeLocation> path) {
    for (MazeLocation ml : path) {
        grid[ml.row][ml.column] = Cell.PATH;
    }
    grid[start.row][start.column] = Cell.START;
    grid[goal.row][goal.column] = Cell.GOAL;
}

public void clear(List<MazeLocation> path) {
    for (MazeLocation ml : path) {
        grid[ml.row][ml.column] = Cell.EMPTY;
    }
    grid[start.row][start.column] = Cell.START;
    grid[goal.row][goal.column] = Cell.GOAL;
}
```

Это было долгое путешествие, но оно подходит к концу — мы готовы пройти по лабиринту (листинг 2.20).

Листинг 2.20. Maze.java (продолжение)

```
public static void main(String[] args) {
    Maze m = new Maze();
    System.out.println(m);

    Node<MazeLocation> solution1 = GenericSearch.dfs(m.start, m::goalTest,
     m::successors);
    if (solution1 == null) {
        System.out.println("No solution found using depth-first search!");
    } else {
        List<MazeLocation> path1 = GenericSearch.nodeToPath(solution1);
        m.mark(path1);
        System.out.println(m);
        m.clear(path1);
    }
}
}
```

Успешное решение будет выглядеть примерно так:

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

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

2.2.4. Поиск в ширину

Вы могли заметить, что пути прохода по лабиринту, найденные с помощью поиска в глубину, кажутся неестественными. Обычно это не самые короткие пути. Поиск в ширину (breadth-first search, BFS) всегда находит кратчайший путь, просматривая на каждой итерации поиска ближайший по отношению к исходному состоянию слой узлов. Для одних задач поиск в глубину позволяет найти решение быстрее, чем поиск в ширину, а для других — наоборот. Поэтому выбор алгоритма поиска иногда оказывается компромиссом между возможностью быстрого решения и гарантированной возможностью найти кратчайший путь к цели, если таковой существует. На рис. 2.5 проиллюстрирован поиск пути по лабиринту в ширину.

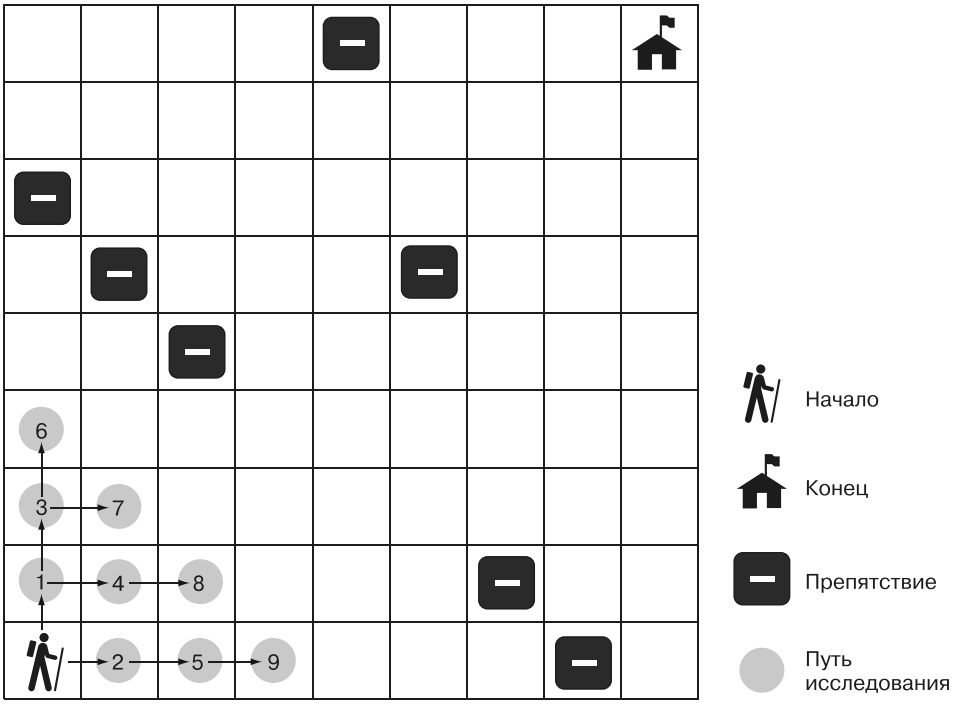


Рис. 2.5. При поиске в ширину сначала просматриваются ближайшие к начальной позиции элементы

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

Чтобы получить более полное представление о том, почему при поиске в ширину всегда определяется кратчайший путь, если только он существует, попробуйте найти железнодорожный маршрут между Бостоном и Нью-Йорком с наименьшим количеством остановок. Если вы будете продолжать двигаться в выбранном направлении и возвращаться назад, когда попали в тупик (как при поиске в глубину), то можете сначала найти маршрут до Сиэтла, прежде чем попасть обратно в Нью-Йорк. Однако при поиске в ширину вы сначала проверите все станции, находящиеся в одной остановке от Бостона. Затем — все станции, находящиеся в двух остановках от Бостона. Затем — в трех остановках от Бостона. Так будет продолжаться до тех пор, пока вы не доберетесь до Нью-Йорка. Поэтому, найдя Нью-Йорк, поймете, что определили маршрут с наименьшим количеством станций, поскольку уже проверили все станции, находящиеся на меньшем расстоянии от Бостона, и ни одна из них не была Нью-Йорком.

Очереди

Для реализации алгоритма BFS нужна структура данных, известная как *очередь*. Если стек — это структура LIFO, то очередь — структура FIFO. Очередь встречается нам и в обычной жизни — например, цепочка людей, ожидающих у входа в туалет. Кто раньше занял очередь, тот первым туда идет. Очередь как минимум имеет те же методы `push()` и `pop()`, что и стек. В сущности, различие между стеком и очередью заключается в том, что очередь удаляет элементы из списка на конце, противоположном тому, куда она их вставляет. Это гарантирует, что самые старые элементы (элементы, «ожидающие» дольше всех) всегда удаляются первыми.

ПРИМЕЧАНИЕ

Как ни странно, но в стандартной библиотеке Java нет класса Queue, однако есть класс Stack. Вместо этого в Java есть интерфейс Queue, реализующий несколько классов стандартной библиотеки Java, включая LinkedList. Еще больше сбивает с толку то, что в интерфейсе Queue стандартной библиотеки Java метод push() называется offer(), а метод pop() — poll().

Алгоритм BFS

Удивительно, но алгоритм поиска в ширину идентичен алгоритму поиска в глубину, только область поиска не стек, а очередь. Такое изменение области поиска

меняет последовательность просмотра состояний и гарантирует, что первыми будут просмотрены состояния, ближайшие к начальному (листинг 2.21).

Листинг 2.21. GenericSearch.java (продолжение)

```
public static <T> Node<T> bfs(T initial, Predicate<T> goalTest,
        Function<T, List<T>> successors) {
    // frontier — это то, что мы только собираемся проверить
    Queue<Node<T>> frontier = new LinkedList<>();
    frontier.offer(new Node<>(initial, null));
    // explored — это то, что уже проверено
    Set<T> explored = new HashSet<>();
    explored.add(initial);

    // продолжаем, пока есть что проверять
    while (!frontier.isEmpty()) {
        Node<T> currentNode = frontier.poll();
        T currentState = currentNode.state;
        // если мы нашли искомое, то процесс закончен
        if (goalTest.test(currentState)) {
            return currentNode;
        }
        // ищем неисследованные ячейки, в которые можно перейти
        for (T child : successors.apply(currentState)) {
            if (explored.contains(child)) {
                continue; // пропустить состояния, которые уже исследовали
            }
            explored.add(child);
            frontier.offer(new Node<>(child, currentNode));
        }
    }
    return null; // все проверили, пути к целевой точке не нашли
}
```

Если вы запустите `bfs()`, то увидите, что эта функция всегда находит кратчайшее решение для рассматриваемого лабиринта. Теперь в `Maze.java` можно изменить метод `main()`, чтобы протестировать два разных способа решения одного и того же лабиринта (листинг 2.22).

Листинг 2.22. Maze.java (продолжение)

```
public static void main(String[] args) {
    Maze m = new Maze();
    System.out.println(m);

    Node<MazeLocation> solution1 = GenericSearch.dfs(m.start, m::goalTest,
     m::successors);
    if (solution1 == null) {
        System.out.println("No solution found using depth-first search!");
    } else {
        List<MazeLocation> path1 = GenericSearch.nodeToPath(solution1);
        m.mark(path1);
```

```
        System.out.println(m);
        m.clear(path1);
    }

    Node<MazeLocation> solution2 =
        GenericSearch.bfs(m.start, m::goalTest, m::successors);
    if (solution2 == null) {
        System.out.println("No solution found using breadth-first search!");
    } else {
        List<MazeLocation> path2 = GenericSearch.nodeToPath(solution2);
        m.mark(path2);
        System.out.println(m);
        m.clear(path2);
    }
}
```

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

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

2.2.5. Поиск по алгоритму A*

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

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

Эвристическая функция $h(n)$ позволяет оценить затраты, необходимые для того, чтобы из заданного состояния достичь целевого состояния. Можно доказать, что

если $h(n)$ — *допустимая эвристическая функция*, то найденный конечный путь будет оптимальным. Допустимая эвристическая функция — это функция, которая никогда не переоценивает затраты на достижение цели. На двумерной плоскости пример такой эвристической функции — расстояние по прямой линии, поскольку прямая линия — всегда кратчайший путь[1].

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

Очереди с приоритетом

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

Стандартная библиотека Java содержит класс `PriorityQueue`, который, в свою очередь, содержит те же методы `offer()` и `poll()`, что и интерфейс `Queue`. Все, что помещено в `PriorityQueue`, должно быть `Comparable`. Чтобы определить приоритет одного элемента по сравнению с другим элементом такого же типа, `PriorityQueue` сравнивает их с помощью метода `compareTo()`. Вот почему ранее нам нужно было реализовать этот метод. Объекты `Node` сравниваются между собой по их значениям $f(n)$, которые являются простой суммой свойств `cost` и `heuristic`.

Эвристика

Эвристика — это интуитивное представление о том, как решить задачу[1]. В случае прохода по лабиринту эвристика стремится выбрать в нем лучшую точку для поиска следующей точки в желании добраться до цели. Другими словами, это обоснованное предположение о том, какие узлы из области поиска находятся ближе всего к цели. Как уже упоминалось, если эвристика, используемая при поиске по алгоритму A*, дает точный относительный результат и допустима (никогда не переоценивает расстояние), то A* составит кратчайший путь. Эвристика,

---

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

которая вычисляет меньшие значения, в итоге дает поиск по большему количеству состояний, тогда как эвристика, значение которой ближе к точному реальному расстоянию (но не больше него, иначе эвристика будет недопустимой), выполняет поиск по меньшему количеству состояний. Следовательно, идеальная эвристика максимально приближается к реальному расстоянию, но не превышает его.

Евклидово расстояние

Как известно из геометрии, кратчайший путь между двумя точками — это прямая. Следовательно, вполне оправданно, что для поиска пути по лабиринту прямолинейная эвристика всегда будет допустимой. Согласно теореме Пифагора евклидово расстояние составляет `distance = √((difference in x)² + (difference in y)²)`. Для наших лабиринтов разность по оси X эквивалентна разности в столбцах между двумя точками лабиринта, а разность по оси Y — разности в строках. Обратите внимание: мы снова реализуем этот код в `Maze.java` (листинг 2.23).

Листинг 2.23. Maze.java (продолжение)

```
public double euclideanDistance(MazeLocation ml) {
    int xdist = ml.column – goal.column;
    int ydist = ml.row – goal.row;
    return Math.sqrt((xdist * xdist) + (ydist * ydist));
}
```

`euclideanDistance()` — это функция, которая берет местоположение лабиринта и возвращает значение расстояния по прямой линии до цели. Функция, возвращаемая `distance()`, принимает в качестве аргумента только начальную точку прохода по лабиринту и всегда знает конечную точку.

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

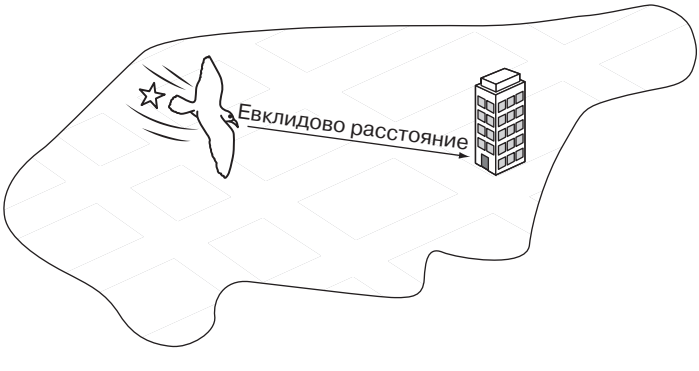


Рис. 2.6. Евклидово расстояние — это длина прямой линии, соединяющей начальную и конечную точки

Манхэттенское расстояние

Евклидово расстояние — это прекрасно, но для нашей конкретной задачи (лабиринт, в котором можно передвигаться только в одном из четырех направлений) можно добиться еще большего. Манхэттенское расстояние определяется навигацией по улицам Манхэттена — самого известного района Нью-Йорка, улицы которого образуют регулярную сетку. Чтобы добраться из любой его точки в любую другую точку, нужно пройти определенное количество кварталов по горизонтали и вертикали. (На Манхэттене почти нет диагональных улиц.) Расстояние в Манхэттене определяется путем простого вычисления разности в строках между двумя точками лабиринта и суммирования ее с разностью в столбцах (листинг 2.24). Вычисление манхэттенского расстояния показано на рис. 2.7.

Листинг 2.24. Maze.java (продолжение)

```
public double manhattanDistance(MazeLocation ml) {
    int xdist = Math.abs(ml.column – goal.column);
    int ydist = Math.abs(ml.row – goal.row);
    return (xdist + ydist);
}
```

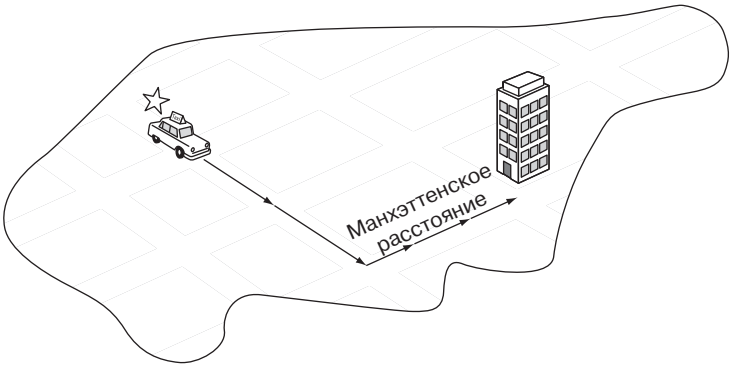


Рис. 2.7. У манхэттенского расстояния нет диагоналей. Путь должен проходить по параллельным или перпендикулярным линиям

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

будут оптимальными, поскольку манхэттенское расстояние допустимо (никогда не превышает реальное расстояние) для лабиринтов, в которых разрешены только четыре направления движения.

Алгоритм A*

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

Для простоты функция `astar()` не принимает в качестве параметра функцию вычисления затрат. Вместо этого мы считаем, что затраты для каждого шага в лабиринте равны 1. Каждому новому узлу назначаются затраты, основанные на этой простой формуле, а также эвристический показатель, вычисленный с использованием новой функции `heuristic()`, передаваемой функции поиска в качестве параметра. За исключением этих изменений, функция `astar()` очень похожа на `bfs()` (листинг 2.25). Откройте их в соседних окнах для сравнения.

Листинг 2.25. GenericSearch.java

```
public static <T> Node<T> astar(T initial, Predicate<T> goalTest,
        Function<T, List<T>> successors, ToDoubleFunction<T> heuristic) {
    // frontier — то, куда мы хотим двигаться
    PriorityQueue<Node<T>> frontier = new PriorityQueue<>();
    frontier.offer(new Node<>(initial, null, 0.0,
        heuristic.applyAsDouble(initial)));
    // explored — то, что мы уже просмотрели
    Map<T, Double> explored = new HashMap<>();
    explored.put(initial, 0.0);
    // продолжаем, пока есть что просматривать
    while (!frontier.isEmpty()) {
        Node<T> currentNode = frontier.poll();
        T currentState = currentNode.state;
        // если цель найдена, мы закончили
        if (goalTest.test(currentState)) {
            return currentNode;
        }
        // проверяем, в какую из неисследованных ячеек направиться
        for (T child : successors.apply(currentState)) {
```

```
            // 1 – для сетки, для более сложных приложений
            // здесь должна быть функция затрат
            double newCost = currentNode.cost + 1;
            if (!explored.containsKey(child) || explored.get(child) > newCost) {
                explored.put(child, newCost);
                frontier.offer(new Node<>(child, currentNode, newCost,
                  heuristic.applyAsDouble(child)));
            }
        }
    }

    return null; // все проверили, пути к целевой точке не нашли
}
```

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

Вперед — попробуйте применить `astar()` к тому же лабиринту, что был задействован в разделе тестирования в файле `Maze.java` (листинг 2.26).

Листинг 2.26. Maze.java (продолжение)

```
public static void main(String[] args) {
    Maze m = new Maze();
    System.out.println(m);

    Node<MazeLocation> solution1 = GenericSearch.dfs(m.start,
      m::goalTest, m::successors);
    if (solution1 == null) {
        System.out.println("No solution found using depth-first search!");
    } else {
        List<MazeLocation> path1 = GenericSearch.nodeToPath(solution1);
        m.mark(path1);
        System.out.println(m);
        m.clear(path1);
    }
    Node<MazeLocation> solution2 = GenericSearch.bfs(m.start,
      m::goalTest, m::successors);
```

```
    if (solution2 == null) {
        System.out.println("No solution found using breadth-first search!");
    } else {
        List<MazeLocation> path2 = GenericSearch.nodeToPath(solution2);
        m.mark(path2);
        System.out.println(m);
        m.clear(path2);
    }

    Node<MazeLocation> solution3 = GenericSearch.astar(m.start,
      m::goalTest, m::successors, m::manhattanDistance);
    if (solution3 == null) {
        System.out.println("No solution found using A*!");
    } else {
        List<MazeLocation> path3 = GenericSearch.nodeToPath(solution3);
        m.mark(path3);
        System.out.println(m);
        m.clear(path3);
    }
}
```

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

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

2.3. МИССИОНЕРЫ И ЛЮДОЕДЫ

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

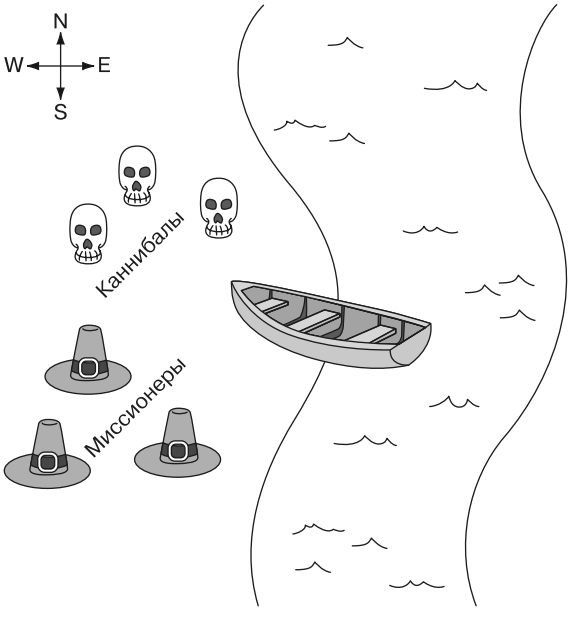


Рис. 2.8. Миссионеры и каннибалы должны использовать свою единственную лодку, чтобы переправить всех через реку с западного берега на восточный. Если людоедов в какой-то момент окажется больше, чем миссионеров, они их съедят

2.3.1. Представление задачи

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

Прежде всего создадим небольшую вспомогательную переменную для отслеживания максимального количества миссионеров или людоедов. Затем определим основной класс (листинг 2.27).

Листинг 2.27. Missionaries.java

```
package chapter2;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

import chapter2.GenericSearch.Node;
```

```
public class MCState {
    private static final int MAX_NUM = 3;
    private final int wm; // миссионеры с западного берега
    private final int wc; // людоеды с западного берега
    private final int em; // миссионеры с восточного берега
    private final int ec; // людоеды с восточного берега
    private final boolean boat; // лодка на западном берегу?

public MCState(int missionaries, int cannibals, boolean boat) {
    wm = missionaries;
    wc = cannibals;
    em = MAX_NUM – wm;
    ec = MAX_NUM – wc;
    this.boat = boat;
}

@Override
public String toString() {
    return String.format(
    "On the west bank there are %d missionaries and %d cannibals.%n"
    + "On the east bank there are %d missionaries and %d cannibals.%n"
    + "The boat is on the %s bank.","
    wm, wc, em, ec,
    boat ? "west" : "east");
}
```

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

Действуя в рамках существующих функций поиска, мы должны определить функцию проверки того, является ли состояние целевым, и функцию для поиска преемников для любого состояния. Функция проверки целевого состояния, как и в задаче прохода по лабиринту, довольно проста. Наша цель состоит в том, чтобы просто достичь разрешенного состояния, при котором все миссионеры и людоеды оказываются на восточном берегу. Мы добавим это в `MCState` в качестве метода (листинг 2.28).

Листинг 2.28. Missionaries.java (продолжение)

```
public boolean goalTest() {
    return isLegal() && em == MAX_NUM && ec == MAX_NUM;
}
```

Чтобы создать функцию преемников, необходимо перебрать все возможные шаги, которые могут быть сделаны для перевозки с одного берега на другой, и затем проверить, приводит ли каждый из них к разрешенному состоянию. Напомню, что разрешенное состояние — такое, при котором на каждом берегу количество людоедов не превышает количество миссионеров. Чтобы определить это, мы

можем создать вспомогательное свойство (как метод в `MCState`), проверяющее, допустимо ли данное состояние (листинг 2.29).

Листинг 2.29. Missionaries.java (продолжение)

```
public boolean isLegal() {
    if (wm < wc && wm > 0) {
        return false;
    }
    if (em < ec && em > 0) {
        return false;
    }
    return true;
}
```

Реальная функция `successors` немного многословна. Это сделано для того, чтобы она была понятной. Функция пытается добавить каждую возможную комбинацию из одного или двух человек, пересекающих реку с того берега, на котором в настоящий момент находится лодка. Когда все возможные ходы будут добавлены, функция отфильтровывает те, что действительно допустимы, используя списковое включение (`removeIf()`). Функция `Predicate.not()`добавлена в Java 11. Эта функция также является методом `MCState` (листинг 2.30).

Листинг 2.30. Missionaries.java (продолжение)

```
public static List<MCState> successors(MCState mcs) {
    List<MCState> sucs = new ArrayList<>();
    if (mcs.boat) { // лодка на западном берегу
        if (mcs.wm > 1) {
            sucs.add(new MCState(mcs.wm – 2, mcs.wc, !mcs.boat));
        }
        if (mcs.wm > 0) {
            sucs.add(new MCState(mcs.wm – 1, mcs.wc, !mcs.boat));
        }
        if (mcs.wc > 1) {
            sucs.add(new MCState(mcs.wm, mcs.wc – 2, !mcs.boat));
        }
        if (mcs.wc > 0) {
            sucs.add(new MCState(mcs.wm, mcs.wc – 1, !mcs.boat));
        }
        if (mcs.wc > 0 && mcs.wm > 0) {
            sucs.add(new MCState(mcs.wm – 1, mcs.wc – 1, !mcs.boat));
        }
    } else { // лодка на восточном берегу
        if (mcs.em > 1) {
            sucs.add(new MCState(mcs.wm + 2, mcs.wc, !mcs.boat));
        }
        if (mcs.em > 0) {
            sucs.add(new MCState(mcs.wm + 1, mcs.wc, !mcs.boat));
        }
        if (mcs.ec > 1) {
```

```
            sucs.add(new MCState(mcs.wm, mcs.wc + 2, !mcs.boat));
        }
        if (mcs.ec > 0) {
            sucs.add(new MCState(mcs.wm, mcs.wc + 1, !mcs.boat));
        }
        if (mcs.ec > 0 && mcs.em > 0) {
            sucs.add(new MCState(mcs.wm + 1, mcs.wc + 1, !mcs.boat));
        }
    }
    sucs.removeIf(Predicate.not(MCState::isLegal));
    return sucs;
}
```

2.3.2. Решение

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

Функция `displaySolution()` преобразует путь решения в наглядный вывод — удобочитаемое решение задачи. Эта функция перебирает все состояния, вошедшие в готовый путь, и отслеживает последнее состояние. Она анализирует разницу между последним состоянием и состоянием, которое отображается в настоящий момент, чтобы выяснить, сколько миссионеров и людоедов в каком направлении пересекло реку (листинг 2.31).

Листинг 2.31. Missionaries.java (продолжение)

```
public static void displaySolution(List<MCState> path) {
    if (path.size() == 0) { // проверка правильности
        return;
    }
    MCState oldState = path.get(0);
    System.out.println(oldState);
    for (MCState currentState : path.subList(1, path.size())) {
        if (currentState.boat) {
            System.out.printf("%d missionaries and %d cannibals moved from
     the east bank to the west bank.%n",
                    oldState.em – currentState.em,
                    oldState.ec – currentState.ec);
        } else {
            System.out.printf("%d missionaries and %d cannibals moved from
     the west bank to the east bank.%n",
                    oldState.wm – currentState.wm,
                    oldState.wc – currentState.wc);
        }
```

```
        System.out.println(currentState);
        oldState = currentState;
    }
}
```

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

Последнее, что нам остается сделать, — на самом деле решить задачу о миссионерах и каннибалах. Для этого удобно будет повторно задействовать функцию поиска, которую мы уже написали, потому что она параметризована. Здесь применяется `bfs()`. Для корректной работы с функциями поиска помните, что исследуемая структура данных требует, чтобы выполнялось равенство. Итак, здесь мы снова позволяем Eclipse автоматически генерировать `hashCode()` и `equals()` перед решением проблемы в `main()` (листинг 2.32).

Листинг 2.32. Missionaries.java (продолжение)

```
    // автоматическое создание Eclipse
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + (boat ? 1231 : 1237);
        result = prime * result + ec;
        result = prime * result + em;
        result = prime * result + wc;
        result = prime * result + wm;
        return result;
    }

    // автоматическое создание Eclipse
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        MCState other = (MCState) obj;
        if (boat != other.boat) {
            return false;
        }
        if (ec != other.ec) {
            return false;
        }
        if (em != other.em) {
            return false;
```

```
        }
        if (wc != other.wc) {
            return false;
        }
        if (wm != other.wm) {
            return false;
        }
        return true;
    }

    public static void main(String[] args) {
        MCState start = new MCState(MAX_NUM, MAX_NUM, true);
        Node<MCState> solution =
          GenericSearch.bfs(start, MCState::goalTest, MCState::successors);
        if (solution == null) {
            System.out.println("No solution found!");
        } else {
            List<MCState> path = GenericSearch.nodeToPath(solution);
            displaySolution(path);
        }
    }
}
```

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

```
On the west bank there are 3 missionaries and 3 cannibals.
On the east bank there are 0 missionaries and 0 cannibals.
The boast is on the west bank.
0 missionaries and 2 cannibals moved from the west bank to the east bank.
On the west bank there are 3 missionaries and 1 cannibals.
On the east bank there are 0 missionaries and 2 cannibals.
The boast is on the east bank.
0 missionaries and 1 cannibals moved from the east bank to the west bank.
…
On the west bank there are 0 missionaries and 0 cannibals.
On the east bank there are 3 missionaries and 3 cannibals.
The boast is on the east bank.
```

2.4. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

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

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

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

2.5. УПРАЖНЕНИЯ

1. Продемонстрируйте преимущество в производительности бинарного поиска по сравнению с линейным поиском, создав список из миллиона чисел и определив, сколько времени потребуется созданным в этой главе функциям `linearContains()` и `binaryContains()` для поиска в нем различных чисел.
2. Добавьте в `dfs()`, `bfs()` и `astar()` счетчик, который позволит увидеть, сколько состояний просматривает каждая из этих функций в одном и том же лабиринте. Найдите значения счетчика для 100 различных лабиринтов, чтобы получить статистически значимые результаты.
3. Найдите решение задачи о миссионерах и каннибалах для разного числа миссионеров и каннибалов.

3 Задачи с ограничениями

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

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

В некоторых языках программирования, таких как Prolog и Picat, есть встроенные средства для решения задач с ограничениями. В других языках обычным подходом является создание структуры, которая включает в себя поиск с возвратами и несколько эвристик для повышения производительности поиска. В этой главе мы сначала создадим структуру для CSP-задач, которая будет решать их простым

рекурсивным поиском с возвратами. Затем воспользуемся этой структурой для решения нескольких примеров таких задач.

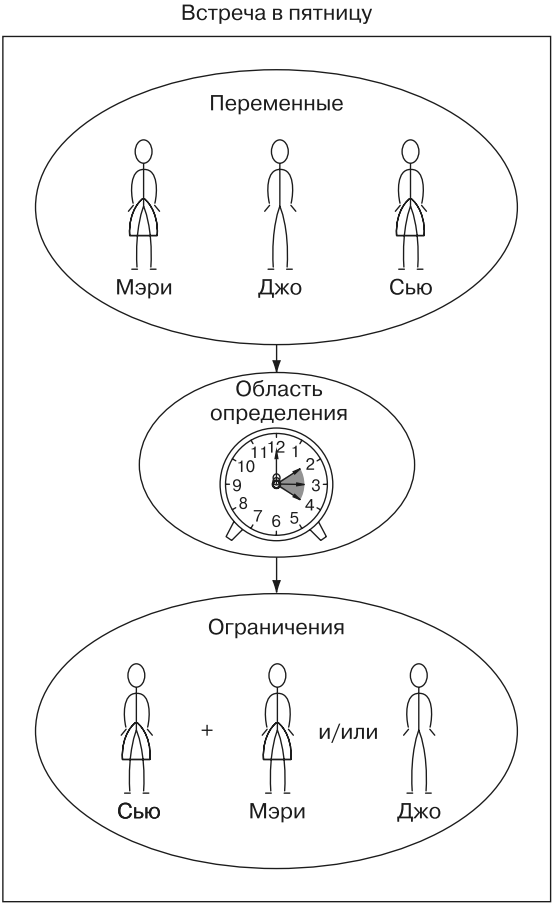


Рис. 3.1. Задачи планирования — это классическое применение структур для удовлетворения ограничений

3.1. ПОСТРОЕНИЕ СТРУКТУРЫ ДЛЯ ЗАДАЧИ С ОГРАНИЧЕНИЯМИ

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

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

Листинг 3.1. Constraint.java

```
package chapter3;

import java.util.List;
import java.util.Map;

// V – тип переменной, D – тип домена.
public abstract class Constraint<V, D> {

    // Переменные, для которых существует ограничение
    protected List<V> variables;

    public Constraint(List<V> variables) {
        this.variables = variables;
    }

    // Необходимо переопределить в подклассах
    public abstract boolean satisfied(Map<V, D> assignment);
}
```

СОВЕТ

В Java бывает сложно выбрать между абстрактным классом и интерфейсом. Абстрактные классы могут содержать переменные экземпляра. Поскольку у нас есть переменные экземпляра переменных, мы используем абстрактный класс.

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

Конструктор создает карту `constraints` `Map`. Метод `addConstraint()` просматривает все переменные, к которым относится данное ограничение, и добавляет себя в соответствие `constraints` для каждой такой переменной. Оба метода имеют простейшую

проверку ошибок и вызывают исключение, если `variable` отсутствует в области определения или существует `constraint` для несуществующей переменной.

Листинг 3.2. CSP.java (продолжение)

```
package chapter3;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class CSP<V, D> {
    private List<V> variables;
    private Map<V, List<D>> domains;
    private Map<V, List<Constraint<V, D>>> constraints = new HashMap<>();

    public CSP(List<V> variables, Map<V, List<D>> domains) {
        this.variables = variables;
        this.domains = domains;
        for (V variable : variables) {
            constraints.put(variable, new ArrayList<>());
            if (!domains.containsKey(variable)) {
                throw new IllegalArgumentException("Every variable should
                      have a domain assigned to it.");
            }
        }
    }

public void addConstraint(Constraint<V, D> constraint) {
    for (V variable : constraint.variables) {
        if (!variables.contains(variable)) {
            throw new IllegalArgumentException("Variable in constraint
                  not in CSP");
        }
        constraints.get(variable).add(constraint);
    }
}
```

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

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

ограничение, наложенное на переменную, не выполняется, возвращается значение `False`.

Листинг 3.3. CSP.java (продолжение)

```
// Проверяем, соответствует ли присваивание значения,
// проверяя все ограничения для данной переменной
public boolean consistent(V variable, Map<V, D> assignment) {
    for (Constraint<V, D> constraint : constraints.get(variable)) {
        if (!constraint.satisfied(assignment)) {
            return false;
        }
    }
    return true;
}
```

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

Листинг 3.4. CSP.java (продолжение)

```
public Map<V, D> backtrackingSearch(Map<V, D> assignment) {
    // присваивание завершено, если существует присваивание
    // для каждой переменной (базовый случай)
    if (assignment.size() == variables.size()) {
        return assignment;
    }
    // получить все переменные из CSP, но не из присваивания
    V unassigned = variables.stream().filter(v ->
      !assignment.containsKey(v)).findFirst().get();
    // получить все возможные значения области определения
    // для первой переменной без присваивания
    for (D value : domains.get(unassigned)) {
        // мелкая копия присваивания, которую мы можем изменить
        Map<V, D> localAssignment = new HashMap<>(assignment);
        localAssignment.put(unassigned, value);
        // если нет противоречий, продолжаем рекурсию
        if (consistent(unassigned, localAssignment)) {
            Map<V, D> result = backtrackingSearch(localAssignment);
            // если результат не найден, заканчиваем возвраты
            if (result != null) {
                return result;
            }
        }
```

```
        }
        return null;
    }

    // вспомогательный класс функции backtrackingSearch
    public Map<V, D> backtrackingSearch() {
        return backtrackingSearch(new HashMap<>());
    }
}
```

Исследуем `backtrackingSearch()` построчно.

```
if (assignment.size() == variables.size()) {
    return assignment;
}
```

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

```
V unassigned = variables.stream().filter(v ->
    !assignment.containsKey(v)).findFirst().get();
```

Чтобы выбрать новую переменную, область определения которой будем исследовать, мы просто просматриваем все переменные и находим первую, которая не имеет присваивания. Для этого создаем поток данных, отфильтрованных по `assignment`, и извлекаем первую, которая не назначена, с помощью `findFirst()`. `filter()`, принимая `Predicate`. `Predicate` — это функциональный интерфейс, описывающий функцию, который принимает один аргумент и возвращает логическое значение. Предикат — это лямбда-выражение (`v -> !Assignment.containsKey(v)`), которое возвращает значение `true`, если `assignment` не содержит аргумент, который в данном случае будет переменной для нашего `CSP`.

```
for (D value : domains.get(unassigned)) {
    Map<V, D> localAssignment = new HashMap<>(assignment);
    localAssignment.put(unassigned, value);
```

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

```
if (consistent(unassigned, localAssignment)) {
    Map<V, D> result = backtrackingSearch(localAssignment);
    if (result != null) {
        return result;
    }
}
```

Для данной переменной мы пытаемся определить все возможные значения домена. Каждое новое значение будет храниться на локальной карте с именем `localAssignment`.

```
return null;
```

Наконец, если мы рассмотрели все возможные значения области определения для конкретной переменной и не обнаружили решения, в котором использовался бы существующий набор назначений, то возвращаем `null`, что указывает на отсутствие решения. В результате по цепочке рекурсии будет выполнен возврат к точке, в которой могло быть принято другое предварительное присваивание.

3.2. ЗАДАЧА РАСКРАШИВАНИЯ КАРТЫ АВСТРАЛИИ

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

Ответ: да. Попробуйте сами (самый простой способ — напечатать карту Австралии с белым фоном). Мы, люди, можем быстро найти решение путем изучения карты и небольшого количества проб и ошибок. На самом деле это тривиальная задача, которая отлично подойдет в качестве первой для нашей программы решения задач с ограничениями методом поиска с возвратом (рис. 3.2).

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

Чтобы реализовать такие двоичные ограничения в коде, нужно создать подкласс класса `Constraint`. Конструктор подкласса `MapColoringConstraint` будет принимать две переменные — две области, имеющие общую границу. Его

переопределенный метод `satisfied()` сначала проверит, присвоены ли этим двум областям значения (цвета) из области определения. Если нет, то ограничение считается тривиально выполненным до тех пор, пока цвета не будут присвоены. (Пока у одного из регионов нет цвета, конфликт невозможен.) Затем метод проверит, присвоен ли двум областям один и тот же цвет. Очевидно, что если цвета одинаковы, то существует конфликт, означающий: ограничение не выполняется.

Рис. 3.2. При раскраске карты Австралии никакие два смежных региона не могут быть окрашены в один и тот же цвет

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

Листинг 3.5. MapColoringConstraint.java

```
package chapter3;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public final class MapColoringConstraint extends Constraint<String, String> {
    private String place1, place2;

    public MapColoringConstraint(String place1, String place2) {
        super(List.of(place1, place2));
        this.place1 = place1;
        this.place2 = place2;
    }

    @Override
    public boolean satisfied(Map<String, String> assignment) {
        // если какой-либо регион place отсутствует в присваивании,
        // то его цвета не могут привести к конфликту
        if (!assignment.containsKey(place1) ||
            !assignment.containsKey(place2)) {
            return true;
        }
        // проверяем, не совпадает ли цвет, присвоенный
        // place1, с цветом, присвоенным place2
        return !assignment.get(place1).equals(assignment.get(place2));
    }
```

Теперь, когда есть способ реализации ограничений между регионами, можно легко уточнить задачу о раскрашивании карты Австралии с помощью программы решения методом CSP — нужно лишь заполнить области определения и переменные, а затем добавить ограничения (листинг 3.6).

Листинг 3.6. MapColoringConstraint.java (продолжение)

```
    public static void main(String[] args) {
        List<String> variables = List.of("Western Australia", "Northern
            Territory", "South Australia", "Queensland", "New South Wales",
            "Victoria", "Tasmania");
        Map<String, List<String>> domains = new HashMap<>();
        for (String variable : variables) {
            domains.put(variable, List.of("red", "green", "blue"));
        }
        CSP<String, String> csp = new CSP<>(variables, domains);
        csp.addConstraint(new MapColoringConstraint("Western Australia",
            "Northern Territory"));
        csp.addConstraint(new MapColoringConstraint("Western Australia",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("South Australia",
            "Northern Territory"));
        csp.addConstraint(new MapColoringConstraint("Queensland",
            "Northern Territory"));
```

```
        csp.addConstraint(new MapColoringConstraint("Queensland",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Queensland",
            "New South Wales"));
        csp.addConstraint(new MapColoringConstraint("New South Wales",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Victoria",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Victoria",
            "New South Wales"));
        csp.addConstraint(new MapColoringConstraint("Victoria", "Tasmania"));
```

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

Листинг 3.7. MapColoringConstraint.java (продолжение)

```
        Map<String, String> solution = csp.backtrackingSearch();
        if (solution == null) {
            System.out.println("No solution found!");
        } else {
            System.out.println(solution);
        }
    }

}
```

Правильное решение будет представлять собой список цветов, присвоенных регионам:

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

3.3. ЗАДАЧА ВОСЬМИ ФЕРЗЕЙ

Шахматная доска — это сетка размером 8 × 8 клеток. Ферзь — шахматная фигура, которая может перемещаться по шахматной доске на любое количество клеток по любой горизонтали, вертикали или диагонали. Если за один ход ферзь может переместиться на клетку, на которой стоит другая фигура, не перепрыгивая ни через какую другую фигуру, то ферзь атакует эту фигуру. (Другими словами, если фигура находится в зоне прямой видимости ферзя, то она подвергается атаке.) Задача восьми ферзей состоит в том, как разместить восемь ферзей на шахматной доске таким образом, чтобы ни один из них не атаковал другого (рис. 3.3).

Чтобы представить клетки на шахматной доске, присвоим каждой из них два целых числа, обозначающих горизонталь и вертикаль. Мы можем гарантировать, что никакая пара из восьми ферзей не находится на одной вертикали, просто присвоив им последовательно номера вертикалей с первого по восьмой.

Переменными в задаче с ограничениями могут быть просто вертикали рассматриваемых ферзей. Области определения могут быть допустимыми горизонталями, тоже с номерами с первого по восьмой. В листинге 3.8 показан код, помещенный в конец нашего файла, где присвоены значения этим переменным и областям определения.

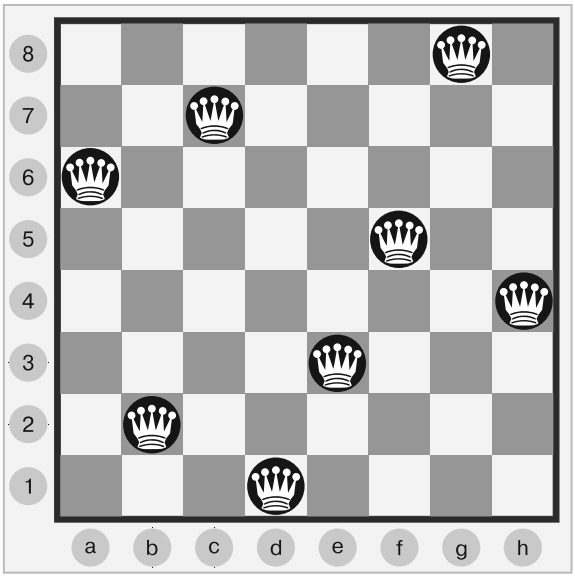


Рис. 3.3. В решении задачи восьми ферзей (существует много решений) никакие два ферзя не могут угрожать друг другу

Листинг 3.8. QueensConstraint.java

```
public static void main(String[] args) {
    List<Integer> columns = List.of(1, 2, 3, 4, 5, 6, 7, 8);
    Map<Integer, List<Integer>> rows = new HashMap<>();
    for (int column : columns) {
        rows.put(column, List.of(1, 2, 3, 4, 5, 6, 7, 8));
    }
    CSP<Integer, Integer> csp = new CSP<>(columns, rows);
```

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

Обратите внимание на то, что следующий код (листинг 3.9) расположен вверху исходного файла.

Листинг 3.9. QueensConstraint.java (продолжение)

```
package chapter3;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

public class QueensConstraint extends Constraint<Integer, Integer> {
    private List<Integer> columns;

    public QueensConstraint(List<Integer> columns) {
        super(columns);
        this.columns = columns;
    }

    @Override
    public boolean satisfied(Map<Integer, Integer> assignment) {
        for (Entry<Integer, Integer> item : assignment.entrySet()) {
            // q1c = ферзь на 1-й вертикали, q1r = ферзь на 1-й горизонтали
            int q1c = item.getKey();
            int q1r = item.getValue();
            // q2c = ферзь на 2-й вертикали
            for (int q2c = q1c + 1; q2c <= columns.size(); q2c++) {
                if (assignment.containsKey(q2c)) {
                    // q2r = ферзь на 2-й горизонтали
                    int q2r = assignment.get(q2c);
                    // тот же ряд?
                    if (q1r == q2r) {
                        return false;
                    }
                    // одна диагональ?
                    if (Math.abs(q1r – q2r) == Math.abs(q1c – q2c)) {
                        return false;
                    }
                }
            }
        }
        return true; // нет конфликтов
    }
```

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

Листинг 3.10. QueensConstraint.java (продолжение)

```
        csp.addConstraint(new QueensConstraint(columns));
        Map<Integer, Integer> solution = csp.backtrackingSearch();
        if (solution == null) {
```

```
            System.out.println("No solution found!");
        } else {
            System.out.println(solution);
        }
    }
}
```

Обратите внимание на то, что мы довольно легко приспособили схему решения задач с ограничениями, созданную для раскрашивания карты, к совершенно другому типу задач. В этом и заключается эффективность написания обобщенного кода! Алгоритмы должны быть реализованы настолько широко, насколько это возможно, если только не потребуется специализация для оптимизации производительности конкретного приложения.

Правильное решение присваивает каждому ферзю вертикаль и горизонталь:

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

3.4. ПОИСК СЛОВА

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

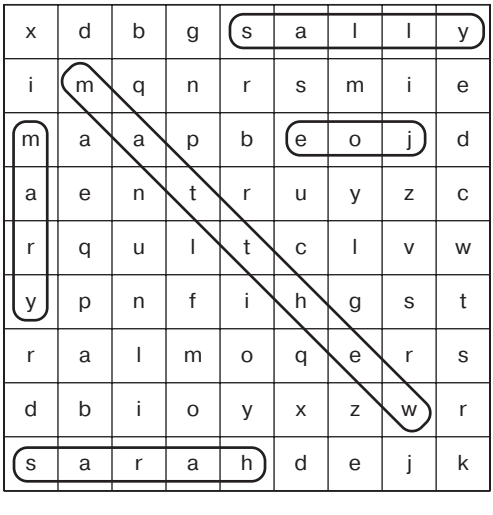


Рис. 3.4. Классическая головоломка для поиска слов, которую можно найти в детской книге головоломок

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

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

Листинг 3.11. WordGrid.java

```
package chapter3;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class WordGrid {

    public static class GridLocation {
        public final int row, column;

        public GridLocation(int row, int column) {
            this.row = row;
            this.column = column;
        }

        // автоматическое создание Eclipse
        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + column;
            result = prime * result + row;
            return result;
        }
        // автоматическое создание Eclipse
        @Override
        public boolean equals(Object obj) {
            if (this == obj) {
                return true;
            }
            if (obj == null) {
                return false;
            }
            if (getClass() != obj.getClass()) {
                return false;
            }
            GridLocation other = (GridLocation) obj;
            if (column != other.column) {
                return false;
            }
            if (row != other.row) {
```

```
                return false;
            }
            return true;
        }
    }
```

Сначала заполним сетку буквами английского алфавита (`A-Z`), генерируя случайные символьные коды (целые числа), эквивалентные буквам кода ASCII. Нам также понадобится метод для отметки слова в сетке с учетом списка местоположений и метод отображения сетки (листинг 3.12).

Листинг 3.12. WordGrid.java (продолжение)

```
    private final char ALPHABET_LENGTH = 26;
    private final char FIRST_LETTER = 'A';
    private final int rows, columns;
    private char[][] grid;

    public WordGrid(int rows, int columns) {
        this.rows = rows;
        this.columns = columns;
        grid = new char[rows][columns];

        // инициализируем сетку случайными буквами
        Random random = new Random();
        for (int row = 0; row < rows; row++) {
            for (int column = 0; column < columns; column++) {
                char randomLetter = (char) (random.nextInt(ALPHABET_LENGTH)
                  + FIRST_LETTER);
                grid[row][column] = randomLetter;
            }
        }
    }

    public void mark(String word, List<GridLocation> locations) {
        for (int i = 0; i < word.length(); i++) {
            GridLocation location = locations.get(i);
            grid[location.row][location.column] = word.charAt(i);
        }
    }

    // получаем красивую печатную версию сетки
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (char[] rowArray : grid) {
            sb.append(rowArray);
            sb.append(System.lineSeparator());
        }
        return sb.toString();
    }
```

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

Листинг 3.13. WordGrid.java (продолжение)

```
public List<List<GridLocation>> generateDomain(String word) {
    List<List<GridLocation>> domain = new ArrayList<>();
    int length = word.length();

    for (int row = 0; row < rows; row++) {
        for (int column = 0; column < columns; column++) {
            if (column + length <= columns) {
                // слева направо
                fillRight(domain, row, column, length);
                // по диагонали снизу направо
                if (row + length <= rows) {
                    fillDiagonalRight(domain, row, column, length);
                }
            }
            if (row + length <= rows) {
                // сверху вниз
                fillDown(domain, row, column, length);
                // по диагонали снизу налево
                if (column – length >= 0) {
                    fillDiagonalLeft(domain, row, column, length);
                }
            }
        }
    }
    return domain;
}

private void fillRight(List<List<GridLocation>> domain,
  int row, int column, int length) {
    List<GridLocation> locations = new ArrayList<>();
    for (int c = column; c < (column + length); c++) {
        locations.add(new GridLocation(row, c));
    }
    domain.add(locations);
}

private void fillDiagonalRight(List<List<GridLocation>> domain,
  int row, int column, int length) {
    List<GridLocation> locations = new ArrayList<>();
    int r = row;
    for (int c = column; c < (column + length); c++) {
        locations.add(new GridLocation(r, c));
```

```
            r++;
        }
        domain.add(locations);
    }

    private void fillDown(List<List<GridLocation>> domain, int row,
      int column, int length) {
        List<GridLocation> locations = new ArrayList<>();
        for (int r = row; r < (row + length); r++) {
            locations.add(new GridLocation(r, column));
        }
        domain.add(locations);
    }

    private void fillDiagonalLeft(List<List<GridLocation>> domain,
      int row, int column, int length) {
        List<GridLocation> locations = new ArrayList<>();
        int c = column;
        for (int r = row; r < (row + length); r++) {
            locations.add(new GridLocation(r, c));
            c--;
        }
        domain.add(locations);
    }
}
```

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

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

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

Листинг 3.14. WordSearchConstraint.java (продолжение)

```
package chapter3;

import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Random;
import java.util.Set;
import java.util.stream.Collectors;

import chapter3.WordGrid.GridLocation;

public class WordSearchConstraint extends Constraint<String,
    List<GridLocation>> {

    public WordSearchConstraint(List<String> words) {
        super(words);
    }

    @Override
    public boolean satisfied(Map<String, List<GridLocation>> assignment) {
        // объединение всех GridLocations в один огромный список
        List<GridLocation> allLocations = assignment.values().stream()
          .flatMap(Collection::stream).collect(Collectors.toList());
        // наличие дубликатов положений сетки означает наличие совпадения
        Set<GridLocation> allLocationsSet = new HashSet<>(allLocations);
        // если какие-либо повторяющиеся местоположения сетки найдены,
        // значит, есть перекрытие
        return allLocations.size() == allLocationsSet.size();
    }
```

Листинг 3.15. WordSearchConstraint.java (продолжение)

```
    public static void main(String[] args) {
        WordGrid grid = new WordGrid(9, 9);
        List<String> words = List.of("MATTHEW", "JOE", "MARY", "SARAH", "SALLY");
        // генерация доменов для всех слов
        Map<String, List<List<GridLocation>>> domains = new HashMap<>();
        for (String word : words) {
            domains.put(word, grid.generateDomain(word));
        }
        CSP<String, List<GridLocation>> csp = new CSP<>(words, domains);
        csp.addConstraint(new WordSearchConstraint(words));
        Map<String, List<GridLocation>> solution = csp.backtrackingSearch();
        if (solution == null) {
            System.out.println("No solution found!");
        } else {
            Random random = new Random();
            for (Entry<String, List<GridLocation>> item : solution.entrySet()) {
```

```
                String word = item.getKey();
                List<GridLocation> locations = item.getValue();
                // в половине случаев случайным выбором — задом наперед
                if (random.nextBoolean()) {
                    Collections.reverse(locations);
                }
                grid.mark(word, locations);
            }
            System.out.println(grid);
        }
    }
}
```

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

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

3.5. SEND + MORE = MONEY

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

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

```
SEND
+MORE
=MONEY
```

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

Листинг 3.16. SendMoreMoneyConstraint.java

```
package chapter3;

import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;

public class SendMoreMoneyConstraint extends Constraint<Character, Integer> {
    private List<Character> letters;

    public SendMoreMoneyConstraint(List<Character> letters) {
        super(letters);
        this.letters = letters;
    }

    @Override
    public boolean satisfied(Map<Character, Integer> assignment) {
        // если есть повторяющиеся значения, то решение не подходит
        if ((new HashSet<>(assignment.values())).size() < assignment.size())
    {
            return false;
        }

        // если все переменные назначены, проверяем, правильна ли сумма
        if (assignment.size() == letters.size()) {
            int s = assignment.get('S');
            int e = assignment.get('E');
            int n = assignment.get('N');
            int d = assignment.get('D');
            int m = assignment.get('M');
            int o = assignment.get('O');
            int r = assignment.get('R');
            int y = assignment.get('Y');
            int send = s * 1000 + e * 100 + n * 10 + d;
            int more = m * 1000 + o * 100 + r * 10 + e;
            int money = m * 10000 + o * 1000 + n * 100 + e * 10 + y;
            return send + more == money;
        }
        return true; // нет противоречий
    }
```

Метод `SendMoreMoneyConstraint satisfied()` выполняет несколько действий. Во-первых, он проверяет, соответствуют ли разные буквы одинаковым цифрам. Если это так, то решение неверно и метод возвращает `False`. Затем метод проверяет, всем ли буквам назначены цифры. Если это так, то он проверяет, корректна ли формула SEND + MORE = MONEY при данном присваивании. Если да, то решение найдено и метод возвращает `True`. В противном случае возвращается `False`. Наконец, если еще не всем буквам присвоены цифры, то возвращается `True`. Это сделано для того, чтобы после получения частичного решения работа продолжалась.

Попробуем это запустить (листинг 3.17).

Листинг 3.17. SendMoreMoneyConstraint.java (продолжение)

```
    public static void main(String[] args) {
        List<Character> letters = List.of('S', 'E', 'N', 'D', 'M', 'O', 'R',
     'Y');
        Map<Character, List<Integer>> possibleDigits = new HashMap<>();
        for (Character letter : letters) {
            possibleDigits.put(letter, List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9));
        }
        // мы не принимаем ответы, начинающиеся с 0
        possibleDigits.replace('M', List.of(1));
        CSP<Character, Integer> csp = new CSP<>(letters, possibleDigits);
        csp.addConstraint(new SendMoreMoneyConstraint(letters));
        Map<Character, Integer> solution = csp.backtrackingSearch();
        if (solution == null) {
            System.out.println("No solution found!");
        } else {
            System.out.println(solution);
        }
    }
}
```

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

Решение должно выглядеть примерно так:

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

3.6. РАЗМЕЩЕНИЕ ЭЛЕМЕНТОВ НА ПЕЧАТНОЙ ПЛАТЕ

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

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

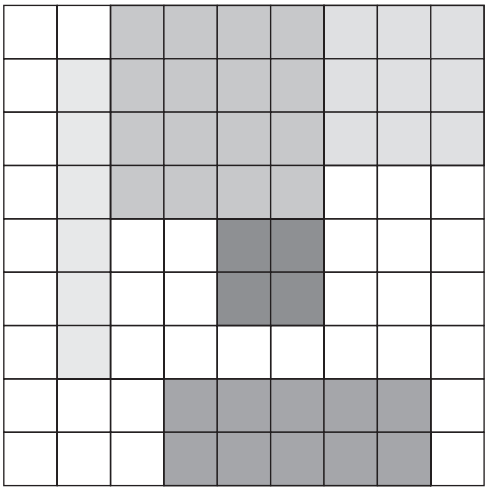


Рис. 3.5. Задача размещения элементов на печатной плате очень похожа на задачу поиска слов, но прямоугольники имеют разную ширину

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

3.7. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

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

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

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

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

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

3.8. УПРАЖНЕНИЯ

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

4 Графовые задачи

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

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

4.1. КАРТА КАК ГРАФ

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

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

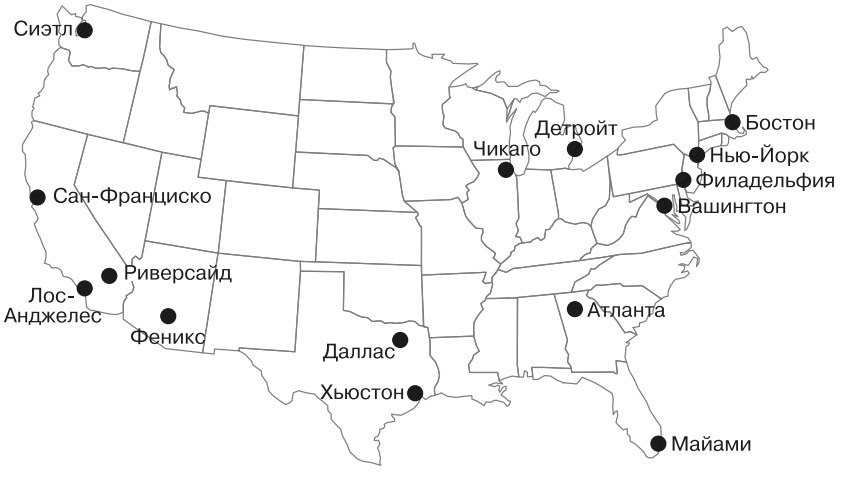


Рис. 4.1. Карта 15 крупнейших муниципальных статистических районов США

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

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

На рис. 4.2 показан граф с вершинами, соответствующими 15 крупнейшим MSA Соединенных Штатов, и ребрами, обозначающими потенциальные маршруты

---

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

[2] *Musk E.* Hyperloop Alpha, https://www.tesla.com/sites/default/files/blog_images/hyperloop-alpha.pdf.

Hyperloop между городами. Маршруты были выбраны в иллюстративных целях. Конечно, в состав сети Hyperloop могут входить и другие потенциальные маршруты.

Такое абстрактное представление реальной задачи подчеркивает эффективность графов. Благодаря этой абстракции можно игнорировать географию Соединенных Штатов и сосредоточиться на изучении потенциальной сети Hyperloop только в контексте соединения городов. В сущности, мы можем представить задачу в виде любого графа, если только его ребра остаются неизменными. Например, на рис. 4.3

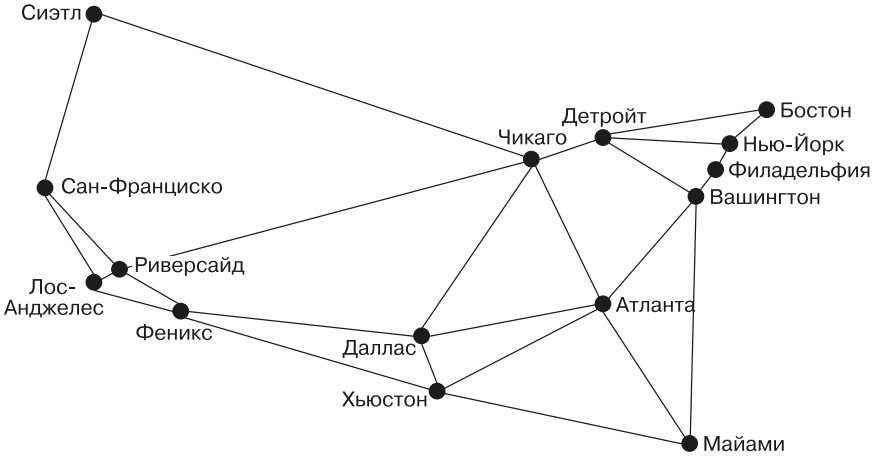


Рис. 4.2. Граф, в котором вершины представляют 15 крупнейших муниципальных статистических районов США, а ребра — потенциальные маршруты Hyperloop между ними

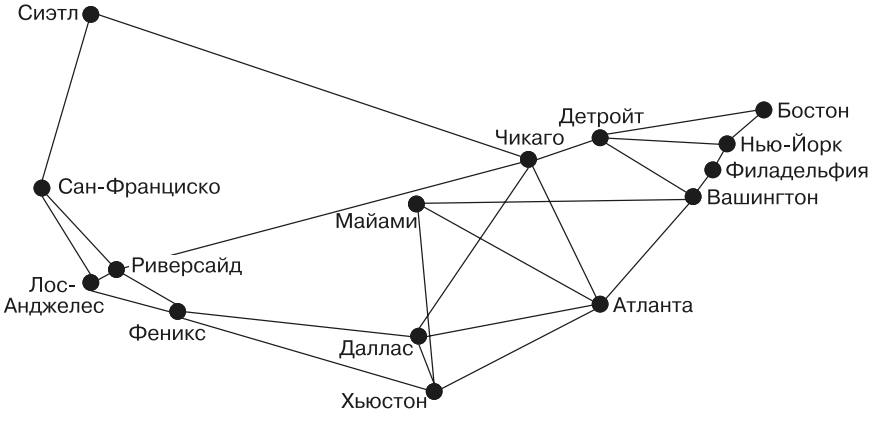


Рис. 4.3. Граф, эквивалентный показанному на рис. 4.2, с измененным положением Майами

изменено положение Майами. Граф, изображенный здесь, будучи абстрактным представлением, позволяет решать те же фундаментальные вычислительные задачи, что и граф на рис. 4.2, несмотря на то что Майами на нем находится не там, где мы ожидаем. Но из соображений здравого смысла будем придерживаться представления, приведенного на рис. 4.2.

4.2. ПОСТРОЕНИЕ ГРАФОВОЙ СТРУКТУРЫ

В данном разделе мы построим два типа графов: *невзвешенный* и *взвешенный*. Во взвешенных графах, которые будут описаны позже, с каждым ребром ассоциируется вес — определенное число, такое как длина в нашем примере.

По своей сути Java — это объектно-ориентированный язык. Мы используем модель наследования, которая является фундаментальной для объектно-ориентированных иерархий классов Java, поэтому не станем дублировать усилия. Классы как для невзвешенных, так и для взвешенных графов будут производными от абстрактного базового класса `Graph`. Это позволит таким классам наследовать большую часть функциональности с небольшими изменениями, отличающими взвешенный граф от невзвешенного.

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

Начнем построение структуры с того, что определим класс `Edge` — простейший механизм в графовой структуре (листинг 4.1).

Листинг 4.1. Edge.java

```
package chapter4;

public class Edge {
    public final int u; // вершина "откуда"
    public final int v; // вершина "куда"

    public Edge(int u, int v) {
        this.u = u;
        this.v = v;
    }

    public Edge reversed() {
        return new Edge(v, u);
    }
```

```
    @Override
    public String toString() {
        return u + " -> " + v;
    }
}
```

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

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

Листинг 4.2. Graph.java

```
package chapter4;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

// V — тип вершин графа
// E — тип ребер
public abstract class Graph<V, E extends Edge> {

    private ArrayList<V> vertices = new ArrayList<>();
    protected ArrayList<ArrayList<E>> edges = new ArrayList<>();

    public Graph() {
    }

    public Graph(List<V> vertices) {
        this.vertices.addAll(vertices);
        for (V vertex : vertices) {
            edges.add(new ArrayList<>());
        }
    }
```

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

Существует множество способов реализации структуры данных графа, но самые распространенные — использование *матрицы вершин* или *списков смежности*. В матрице вершин каждая ячейка представляет собой пересечение двух вершин графа, и значение этой ячейки указывает на связь (или ее отсутствие) между этими вершинами. В нашей структуре данных графа задействуются списки смежности. В таком представлении у каждой вершины есть список вершин, с которыми она связана. В нашем конкретном представлении применяется список списков ребер, поэтому для каждой вершины существует список ребер, которыми она связана с другими вершинами. Этот список списков называется `edges`.

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

Листинг 4.3. Graph.java (продолжение)

```
// Количество вершин
public int getVertexCount() {
    return vertices.size();
}

// Количество ребер
public int getEdgeCount() {
    return edges.stream().mapToInt(ArrayList::size).sum();
}

// Добавляем вершину в граф и возвращаем ее индекс
public int addVertex(V vertex) {
    vertices.add(vertex);
    edges.add(new ArrayList<>());
    return getVertexCount() – 1;
}

// Поиск вершины по индексу
public V vertexAt(int index) {
```

```
        return vertices.get(index);
    }

    // Поиск индекса вершины в графе
    public int indexOf(V vertex) {
        return vertices.indexOf(vertex);
    }

    // Поиск вершин, с которыми связана вершина с заданным индексом
    public List<V> neighborsOf(int index) {
        return edges.get(index).stream()
                .map(edge -> vertexAt(edge.v))
                .collect(Collectors.toList());
    }

    // Поиск индекса вершины; возвращает ее соседей (удобный метод)
    public List<V> neighborsOf(V vertex) {
        return neighborsOf(indexOf(vertex));
    }

    // Возвращает все ребра, связанные с вершиной, имеющей заданный индекс
    public List<E> edgesOf(int index) {
        return edges.get(index);
    }

    // Поиск индекса вершины; возвращает ее ребра (удобный метод)
    public List<E> edgesOf(V vertex) {
        return edgesOf(indexOf(vertex));
    }

    // Упрощенный красивый вывод графа
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < getVertexCount(); i++) {
            sb.append(vertexAt(i));
            sb.append(" -> ");
            sb.append(Arrays.toString(neighborsOf(i).toArray()));
            sb.append(System.lineSeparator());
        }
        return sb.toString();
    }
}
```

Давайте ненадолго вернемся назад и посмотрим, почему большинство методов этого класса существует в двух версиях. Из определения класса мы знаем, что список `vertices` содержит элементы типа `V`, который может быть любым классом Java. Таким образом, у нас есть вершины типа `V`, которые хранятся в списке `vertices`. Но если мы хотим получить эти вершины и впоследствии ими манипулировать, то нужно знать, где они хранятся в данном списке. Следовательно, с каждой вершиной в этом массиве связан индекс (целое число). Если индекс вершины

неизвестен, то его нужно найти, просматривая `vertices`. Именно для этого нужны две версии каждого метода: одна оперирует внутренними индексами, другая — самим `V`. Методы, которые работают с `V`, ищут соответствующие индексы и вызывают основанную на индексе функцию. Поэтому их можно считать удобными.

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

```
public List<V> neighborsOf(int index) {
    return edges.get(index).stream()
            .map(edge -> vertexAt(edge.v))
            .collect(Collectors.toList());
}
```

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

Теперь, когда у нас есть базовая функциональность графа, реализованная в абстрактном классе `Graph`, мы можем определить подкласс. Помимо того что графы могут быть направленными или ненаправленными, они могут быть также взвешенными или невзвешенными. Взвешенный граф — это такой граф, у которого каждому из ребер сопоставлено некое значение, обычно числовое. В потенциальной сети Hyperloop мы могли бы представить вес ребер как расстояния между станциями. Однако сейчас будем иметь дело с невзвешенной версией графа. Невзвешенное ребро — это просто связь между двумя вершинами, следовательно, класс `Edge` невзвешенный. Другой способ показать это таков: о невзвешенном графе известно только то, какие вершины связаны, а о взвешенном — то, какие вершины связаны, и кое-какая дополнительная информация об этих связях. `UnweightedGraph` представляет граф, у которого нет значений, связанных с его ребрами. Другими словами, это комбинация `Graph` с `Edge`.

Листинг 4.4. UnweightedGraph.java

```
package chapter4;

import java.util.List;

import chapter2.GenericSearch;
import chapter2.GenericSearch.Node;
```

```
public class UnweightedGraph<V> extends Graph<V, Edge> {

    public UnweightedGraph(List<V> vertices) {
        super(vertices);
    }

    // Это неориентированный граф, поэтому мы всегда
    // добавляем ребра в обоих направлениях.
    public void addEdge(Edge edge) {
        edges.get(edge.u).add(edge);
        edges.get(edge.v).add(edge.reversed());
    }

    // Добавление ребра с помощью индексов вершин (удобный метод)
    public void addEdge(int u, int v) {
        addEdge(new Edge(u, v));
    }

    // Добавление ребра путем просмотра индексов вершин (удобный метод)
    public void addEdge(V first, V second) {
        addEdge(new Edge(indexOf(first), indexOf(second)));
    }
```

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

```
public void addEdge(Edge edge) {
    edges.get(edge.u).add(edge);
    edges.get(edge.v).add(edge.reversed());
}
```

Как уже упоминалось, в этой главе мы рассматриваем именно ненаправленные графы.

4.2.1. Работа с Edge и UnweightedGraph

Теперь, когда у нас есть конкретные реализации `Edge` и `Graph`, можем создать представление потенциальной сети Hyperloop. Вершины и ребра в `cityGraph` соответствуют вершинам и ребрам, представленным на рис. 4.2. Используя параметризацию, мы можем указать, что вершины будут иметь тип `String` (`UnweightedGraph<String>`). Другими словами, тип `String` заменяет переменную типа `V` (листинг 4.5).

Листинг 4.5. UnweightedGraph.java (продолжение)

```
    public static void main(String[] args) {
        // Представляет 15 крупнейших MSA в США.
        UnweightedGraph<String> cityGraph = new UnweightedGraph<>(
                List.of("Seattle", "San Francisco", "Los Angeles", "Riverside",
                        "Phoenix", "Chicago", "Boston", "New York", "Atlanta",
                        "Miami", "Dallas", "Houston", "Detroit", "Philadelphia",
                        "Washington"));

        cityGraph.addEdge("Seattle", "Chicago");
        cityGraph.addEdge("Seattle", "San Francisco");
        cityGraph.addEdge("San Francisco", "Riverside");
        cityGraph.addEdge("San Francisco", "Los Angeles");
        cityGraph.addEdge("Los Angeles", "Riverside");
        cityGraph.addEdge("Los Angeles", "Phoenix");
        cityGraph.addEdge("Riverside", "Phoenix");
        cityGraph.addEdge("Riverside", "Chicago");
        cityGraph.addEdge("Phoenix", "Dallas");
        cityGraph.addEdge("Phoenix", "Houston");
        cityGraph.addEdge("Dallas", "Chicago");
        cityGraph.addEdge("Dallas", "Atlanta");
        cityGraph.addEdge("Dallas", "Houston");
        cityGraph.addEdge("Houston", "Atlanta");
        cityGraph.addEdge("Houston", "Miami");
        cityGraph.addEdge("Atlanta", "Chicago");
        cityGraph.addEdge("Atlanta", "Washington");
        cityGraph.addEdge("Atlanta", "Miami");
        cityGraph.addEdge("Miami", "Washington");
        cityGraph.addEdge("Chicago", "Detroit");
        cityGraph.addEdge("Detroit", "Boston");
        cityGraph.addEdge("Detroit", "Washington");
        cityGraph.addEdge("Detroit", "New York");
        cityGraph.addEdge("Boston", "New York");
        cityGraph.addEdge("New York", "Philadelphia");
        cityGraph.addEdge("Philadelphia", "Washington");
        System.out.println(cityGraph.toString());
    }
}
```

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

```
Seattle -> [Chicago, San Francisco]
San Francisco -> [Seattle, Riverside, Los Angeles]
Los Angeles -> [San Francisco, Riverside, Phoenix]
Riverside -> [San Francisco, Los Angeles, Phoenix, Chicago]
Phoenix -> [Los Angeles, Riverside, Dallas, Houston]
Chicago -> [Seattle, Riverside, Dallas, Atlanta, Detroit]
```

```
Boston -> [Detroit, New York]
New York -> [Detroit, Boston, Philadelphia]
Atlanta -> [Dallas, Houston, Chicago, Washington, Miami]
Miami -> [Houston, Atlanta, Washington]
Dallas -> [Phoenix, Chicago, Atlanta, Houston]
Houston -> [Phoenix, Dallas, Atlanta, Miami]
Detroit -> [Chicago, Boston, Washington, New York]
Philadelphia -> [New York, Washington]
Washington -> [Atlanta, Miami, Detroit, Philadelphia]
```

4.3. ПОИСК КРАТЧАЙШЕГО ПУТИ

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

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

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

4.3.1. Пересмотр алгоритма поиска в ширину

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

СОВЕТ

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

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

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

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

Учитывая этот план, мы можем добавить соответствующий код в конец основного раздела `main()` файла `UnweightedGraph.java`, чтобы найти на `cityGraph` кратчайший маршрут между Бостоном и Майами (листинг 4.6).

ПРИМЕЧАНИЕ

В листинг 4.4 (ранее в главе, где впервые был определен UnweightedGraph) включены операции импорта (например, chapter2.GenericSearch, chapter2.genericSearch.Node). Это работает, только если пакет chapter2 доступен из пакета chapter4. Если среда разработки не настроена таким образом, вы сможете скопировать класс GenericSearch непосредственно в пакет chapter4 и исключить импорт.

Листинг 4.6. UnweightedGraph.java (продолжение)

```
        Node<String> bfsResult = GenericSearch.bfs("Boston",
                v -> v.equals("Miami"),
                cityGraph::neighborsOf);
        if (bfsResult == null) {
            System.out.println("No solution found using breadth-first search!");
        } else {
            List<String> path = GenericSearch.nodeToPath(bfsResult);
            System.out.println("Path from Boston to Miami:");
            System.out.println(path);
        }
```

Результат должен выглядеть примерно так:

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

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

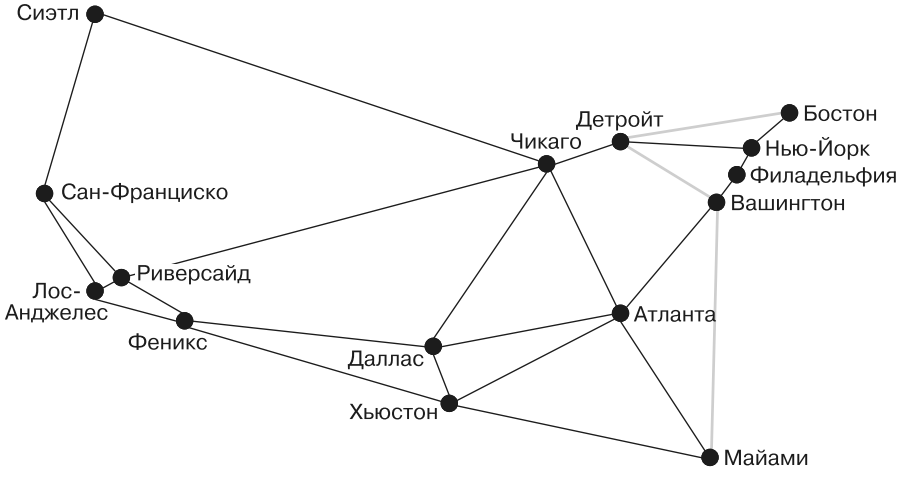


Рис. 4.4. Здесь выделен кратчайший маршрут между Бостоном и Майами с точки зрения количества ребер

4.4. МИНИМИЗАЦИЯ ЗАТРАТ НА ПОСТРОЕНИЕ СЕТИ

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

4.4.1. Работа с весами

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

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

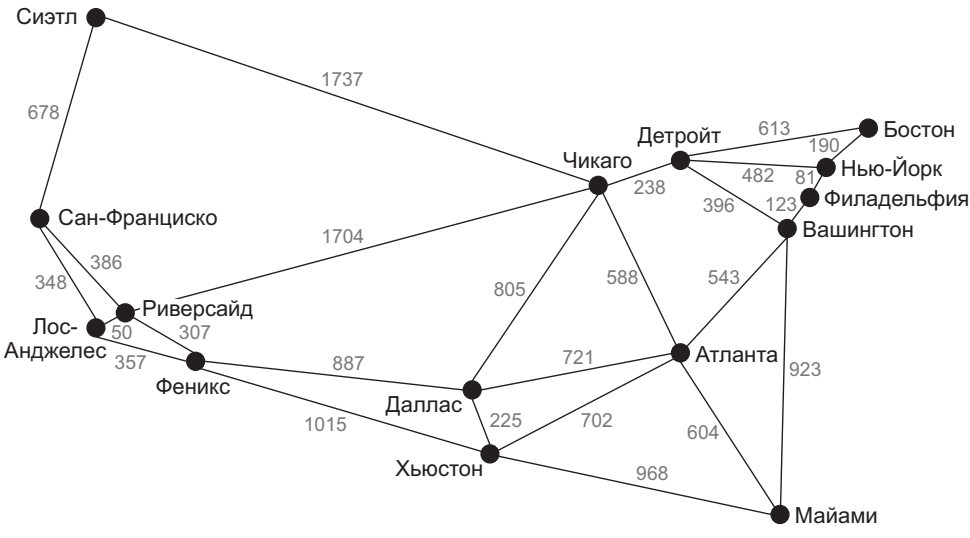


Рис. 4.5. Взвешенный граф с 15 самыми большими MSA Соединенных Штатов, вес каждого ребра представляет собой расстояние между двумя MSA в милях

Алгоритму Ярника, о котором мы вскоре расскажем, требуется возможность сравнивать ребра между собой, чтобы выбрать из них ребро с меньшим весом. Это легко реализовать посредством числовых весов (листинг 4.7).

Листинг 4.7. WeightedEdge.java

```
package chapter4;

public class WeightedEdge extends Edge implements Comparable<WeightedEdge> {
    public final double weight;

    public WeightedEdge(int u, int v, double weight) {
        super(u, v);
        this.weight = weight;
    }

    @Override
    public WeightedEdge reversed() {
        return new WeightedEdge(v, u, weight);
    }

    // так можно упорядочить ребра по весу и найти ребро с минимальным весом
    @Override
    public int compareTo(WeightedEdge other) {
        Double mine = weight;
        Double theirs = other.weight;
        return mine.compareTo(theirs);
```

```
    }

    @Override
    public String toString() {
        return u + " " + weight + "> " + v;
    }
}
```

Реализация `WeightedEdge` не сильно отличается от `Edge`. Разница только в том, что добавлено свойство `weight` и реализован интерфейс `Comparable` с помощью `compareTo()`, благодаря чему два объекта `WeightedEdge` можно сравнивать. Метод `compareTo()` просматривает только веса ребер (а не унаследованные свойства `u` и `v`), так как алгоритм Ярника осуществляет поиск наименьшего по весу ребра.

Класс `WeightedGraph` наследует бóльшую часть своей функциональности от `Graph`. Помимо этого, у данного класса есть методы инициализации и удобные методы сложения объектов `WeightedEdge`. К тому же в нем реализована собственная версия `toString()` (листинг 4.8).

Листинг 4.8. WeightedGraph.java

```
package chapter4;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.function.IntConsumer;

public class WeightedGraph<V> extends Graph<V, WeightedEdge> {

    public WeightedGraph(List<V> vertices) {
        super(vertices);
    }

    // Это невзвешенный граф, поэтому мы всегда
    // добавляем ребра в обоих направлениях
    public void addEdge(WeightedEdge edge) {
        edges.get(edge.u).add(edge);
        edges.get(edge.v).add(edge.reversed());
    }

    public void addEdge(int u, int v, float weight) {
        addEdge(new WeightedEdge(u, v, weight));
    }

    public void addEdge(V first, V second, float weight) {
```

```
        addEdge(indexOf(first), indexOf(second), weight);
    }

    // упрощенная печать графика
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < getVertexCount(); i++) {
            sb.append(vertexAt(i));
            sb.append(" -> ");
            sb.append(Arrays.toString(edgesOf(i).stream()
                    .map(we -> "(" + vertexAt(we.v) + ", " + we.weight +
                               ")").toArray()));
            sb.append(System.lineSeparator());
    }
    return sb.toString();
}
```

Теперь можно наконец определить взвешенный граф. Взвешенный граф, с которым мы будем работать (см. рис. 4.5), называется `cityGraph2` (листинг 4.9).

Листинг 4.9. WeightedGraph.java (продолжение)

```
    public static void main(String[] args) {
        // представляет 15 крупнейших MSA в США.
        WeightedGraph<String> cityGraph2 = new WeightedGraph<>(
            List.of("Seattle", "San Francisco", "Los Angeles", "Riverside",
            "Phoenix", "Chicago", "Boston", "New York", "Atlanta", "Miami",
            "Dallas", "Houston", "Detroit", "Philadelphia", "Washington"));

        cityGraph2.addEdge("Seattle", "Chicago", 1737);
        cityGraph2.addEdge("Seattle", "San Francisco", 678);
        cityGraph2.addEdge("San Francisco", "Riverside", 386);
        cityGraph2.addEdge("San Francisco", "Los Angeles", 348);
        cityGraph2.addEdge("Los Angeles", "Riverside", 50);
        cityGraph2.addEdge("Los Angeles", "Phoenix", 357);
        cityGraph2.addEdge("Riverside", "Phoenix", 307);
        cityGraph2.addEdge("Riverside", "Chicago", 1704);
        cityGraph2.addEdge("Phoenix", "Dallas", 887);
        cityGraph2.addEdge("Phoenix", "Houston", 1015);
        cityGraph2.addEdge("Dallas", "Chicago", 805);
        cityGraph2.addEdge("Dallas", "Atlanta", 721);
        cityGraph2.addEdge("Dallas", "Houston", 225);
        cityGraph2.addEdge("Houston", "Atlanta", 702);
        cityGraph2.addEdge("Houston", "Miami", 968);
        cityGraph2.addEdge("Atlanta", "Chicago", 588);
        cityGraph2.addEdge("Atlanta", "Washington", 543);
        cityGraph2.addEdge("Atlanta", "Miami", 604);
        cityGraph2.addEdge("Miami", "Washington", 923);
        cityGraph2.addEdge("Chicago", "Detroit", 238);
        cityGraph2.addEdge("Detroit", "Boston", 613);
        cityGraph2.addEdge("Detroit", "Washington", 396);
```

```
        cityGraph2.addEdge("Detroit", "New York", 482);
        cityGraph2.addEdge("Boston", "New York", 190);
        cityGraph2.addEdge("New York", "Philadelphia", 81);
        cityGraph2.addEdge("Philadelphia", "Washington", 123);

        System.out.println(cityGraph2);
    }
}
```

Поскольку в `WeightedGraph` реализован метод `toString()`, мы можем структурно распечатать `cityGraph2`. При выводе вы увидите и вершины, с которыми соединена данная вершина, и веса этих соединений:

```
Seattle -> [(Chicago, 1737.0), (San Francisco, 678.0)]
San Francisco -> [(Seattle, 678.0), (Riverside, 386.0), (Los Angeles, 348.0)]
Los Angeles -> [(San Francisco, 348.0), (Riverside, 50.0), (Phoenix, 357.0)]
Riverside -> [(San Francisco, 386.0), (Los Angeles, 50.0), (Phoenix, 307.0),
     (Chicago, 1704.0)]
Phoenix -> [(Los Angeles, 357.0), (Riverside, 307.0), (Dallas, 887.0), (Houston,
     1015.0)]
Chicago -> [(Seattle, 1737.0), (Riverside, 1704.0), (Dallas, 805.0), (Atlanta,
     588.0), (Detroit, 238.0)]
Boston -> [(Detroit, 613.0), (New York, 190.0)]
New York -> [(Detroit, 482.0), (Boston, 190.0), (Philadelphia, 81.0)]
Atlanta -> [(Dallas, 721.0), (Houston, 702.0), (Chicago, 588.0), (Washington,
     543.0), (Miami, 604.0)]
Miami -> [(Houston, 968.0), (Atlanta, 604.0), (Washington, 923.0)]
Dallas -> [(Phoenix, 887.0), (Chicago, 805.0), (Atlanta, 721.0), (Houston,
     225.0)]
Houston -> [(Phoenix, 1015.0), (Dallas, 225.0), (Atlanta, 702.0), (Miami, 968.0)]
Detroit -> [(Chicago, 238.0), (Boston, 613.0), (Washington, 396.0), (New York,
     482.0)]
Philadelphia -> [(New York, 81.0), (Washington, 123.0)]
Washington -> [(Atlanta, 543.0), (Miami, 923.0), (Detroit, 396.0), (Philadelphia,
     123.0)]
```

4.4.2. Поиск минимального связующего дерева

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

Связный граф — это граф, для которого существует способ добраться из любой вершины в любую другую вершину (все графы, которые мы рассматриваем в этой

главе, связные). *Связующее дерево* — это дерево, которое соединяет все вершины графа. *Минимальное связующее дерево* — это дерево, которое соединяет все вершины во взвешенном графе и имеет минимальный общий вес по сравнению с другими связующими деревьями. Для каждого взвешенного графа можно найти минимальное связующее дерево.

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

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

Вычисление общего веса взвешенного пути

Прежде чем разработать метод поиска минимального связующего дерева, создадим функцию, которую сможем использовать для проверки суммарного веса решения. Найденное минимальное связующее дерево будет представлять собой список взвешенных ребер, из которых оно состоит. Сначала мы определим взвешенный путь как список `WeightedEdge`. Затем определим функцию `totalWeight()`, которая принимает список `WeightedEdges` и находит общий вес, получаемый в результате сложения всех весов ее ребер. Обратите внимание на то, что этот метод и остальные в данной главе будут добавлены к существующему классу `WeightedGraph` (листинг 4.10).

Листинг 4.10. WeightedGraph.java (продолжение)

```
public static double totalWeight(List<WeightedEdge> path) {
    return path.stream().mapToDouble(we -> we.weight).sum();
}
```

Алгоритм Ярника

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

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

ПРИМЕЧАНИЕ

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

Для эффективного выполнения алгоритма Ярника используется очередь с приоритетом (см. главу 2). Каждый раз, когда новая вершина добавляется в минимальное связующее дерево, все ее исходящие ребра, которые связываются с вершинами вне дерева, добавляются в очередь с приоритетом. Ребро с наименьшим весом всегда первым извлекается из очереди с приоритетом, и алгоритм продолжает выполняться до тех пор, пока очередь с приоритетом не опустеет. Это гарантирует, что ребра с наименьшим весом всегда будут первыми добавляться в дерево. Ребра, которые соединяются с вершинами, уже относящимися к дереву, игнорируются при извлечении из очереди.

Далее представлен код функции `mst()` — полная реализация алгоритма Ярника, а также вспомогательная функция для печати взвешенного пути (листинг 4.11).

ПРЕДУПРЕЖДЕНИЕ

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

---

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

Листинг 4.11. WeightedGraph.java (продолжение)

```
    public List<WeightedEdge> mst(int start) {
        LinkedList<WeightedEdge> result = new LinkedList<>(); // здесь мы уже
                                                              // были
        if (start < 0 || start > (getVertexCount() – 1)) {
            return result;
        }
        PriorityQueue<WeightedEdge> pq = new PriorityQueue<>();
        boolean[] visited = new boolean[getVertexCount()];

        // это похоже на внутреннюю функцию "visit"
        IntConsumer visit = index -> {
            visited[index] = true; // пометить как прочитанное
            for (WeightedEdge edge : edgesOf(index)) {
                // добавляем все ребра отсюда в pq
                if (!visited[edge.v]) {
                    pq.offer(edge);
                }
            }
        };

        visit.accept(start); // первая вершина, с которой все начинается
        while (!pq.isEmpty()) { // продолжаем, пока остаются необработанные
                                // вершины
            WeightedEdge edge = pq.poll();
            if (visited[edge.v]) {
                continue; // никогда не просматриваем дважды
            }
            // на данный момент это минимальный вес, так что добавляем в дерево
            result.add(edge);
            visit.accept(edge.v); // посетите места соединений
        }
        return result;
    }

    public void printWeightedPath(List<WeightedEdge> wp) {
    for (WeightedEdge edge : wp) {
        System.out.println(vertexAt(edge.u) + " "
        + edge.weight + "> " + vertexAt(edge.v));
    }
    System.out.println("Total Weight: " + totalWeight(wp));
}
```

Рассмотрим функцию `mst()` построчно:

```
public List<WeightedEdge> mst(int start) {
    LinkedList<WeightedEdge> result = new LinkedList<>(); // здесь мы уже были
    if (start < 0 || start > (getVertexCount() – 1)) {
        return result;
    }
```

Алгоритм возвращает взвешенный путь (`List<WeightedEdge>`), представляющий собой минимальное связующее дерево. Если значение `start` оказалось неверным,

то `mst()` возвращает `None`. В итоге `result` будет содержать взвешенный путь, куда входит минимальное связующее дерево. Именно здесь добавляются объекты `WeightedEdges`, так как ребро с наименьшим весом извлекается из очереди и переводит нас к новой части графа.

```
PriorityQueue<WeightedEdge> pq = new PriorityQueue<>();
boolean[] visited = new boolean[getVertexCount()]; // здесь мы уже были
```

Алгоритм Ярника считается *жадным алгоритмом*, потому что всегда выбирает ребро с наименьшим весом. В `pq` хранятся новые обнаруженные ребра, и отсюда извлекается следующее ребро с наименьшим весом. В `visited` отслеживаются индексы вершин, которые мы уже просмотрели. Это можно сделать также с помощью `Set`, аналогичного `explored` в `bfs()`:

```
IntConsumer visit = index -> {
    visited[index] = true; // пометить как прочитанное
    for (WeightedEdge edge : edgesOf(index)) {
        // добавляем все ребра отсюда в pq
        if (!visited[edge.v]) {
            pq.offer(edge);
        }
    }
};
```

`visit()` — это внутренняя вспомогательная функция, которая отмечает вершину как просмотренную и добавляет в `pq` все ее ребра, которые соединяются с еще не просмотренными вершинами. Функция `visit` реализована как `IntConsumer`, представляющая собой просто функцию, которая принимает `int` в качестве единственного параметра. В этом случае `int` будет индексом посещаемой вершины. Обратите внимание на то, как легко модель списка смежности облегчает поиск ребер, принадлежащих определенной вершине:

```
visit.accept(start); // первая вершина, с которой все начинается
```

`accept()` — это метод `IntConsumer`, вызывающий связанную с ним функцию с предоставленным параметром `int`. Неважно, какая вершина будет посещена первой, если граф несвязный. Если несвязный граф состоит из разъединенных компонентов, то `mst()` вернет дерево, охватывающее тот *компонент*, которому принадлежит начальная вершина:

```
while (!pq.isEmpty()) { // продолжаем, пока остаются необработанные вершины
    WeightedEdge edge = pq.poll();
    if (visited[edge.v]) {
        continue; // никогда не просматриваем дважды
    }
    // на данный момент это минимальный вес, так что добавляем в дерево
        result.add(edge);
```

```
        visit.accept(edge.v); // посетите места соединений
}

return result;
```

Пока в очереди с приоритетом остаются ребра, мы извлекаем их оттуда и проверяем, ведут ли они к вершинам, которых еще нет в дереве. Поскольку очередь с приоритетом возрастающая, из нее сначала извлекаются ребра с наименьшим весом. Это гарантирует, что результат действительно имеет минимальный общий вес. Если ребро, извлеченное из очереди, не приводит к неисследованной вершине, то оно игнорируется. В противном случае, поскольку ребро имеет минимальный из всех видимых весов, оно добавляется к результирующему набору и исследуется новая вершина, к которой ведет это ребро. Когда неисследованных ребер не остается, возвращается результат.

Теперь мы наконец вернемся к задаче соединения всех 15 крупнейших MSA Соединенных Штатов в сеть Hyperloop минимальным количеством трасс. Маршрут, выполняющий эту задачу, — это просто минимальное связующее дерево для `cityGraph2`. Попробуем запустить `mst()` для `cityGraph2` с помощью `main()` (листинг 4.12).

Листинг 4.12. WeightedGraph.java (продолжение)

```
List<WeightedEdge> mst = cityGraph2.mst(0);
cityGraph2.printWeightedPath(mst);
```

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

```
Seattle 678.0> San Francisco
San Francisco 348.0> Los Angeles
Los Angeles 50.0> Riverside
Riverside 307.0> Phoenix
Phoenix 887.0> Dallas
Dallas 225.0> Houston
Houston 702.0> Atlanta
Atlanta 543.0> Washington
Washington 123.0> Philadelphia
Philadelphia 81.0> New York
New York 190.0> Boston
Washington 396.0> Detroit
Detroit 238.0> Chicago
Atlanta 604.0> Miami
Total Weight: 5372.0
```

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

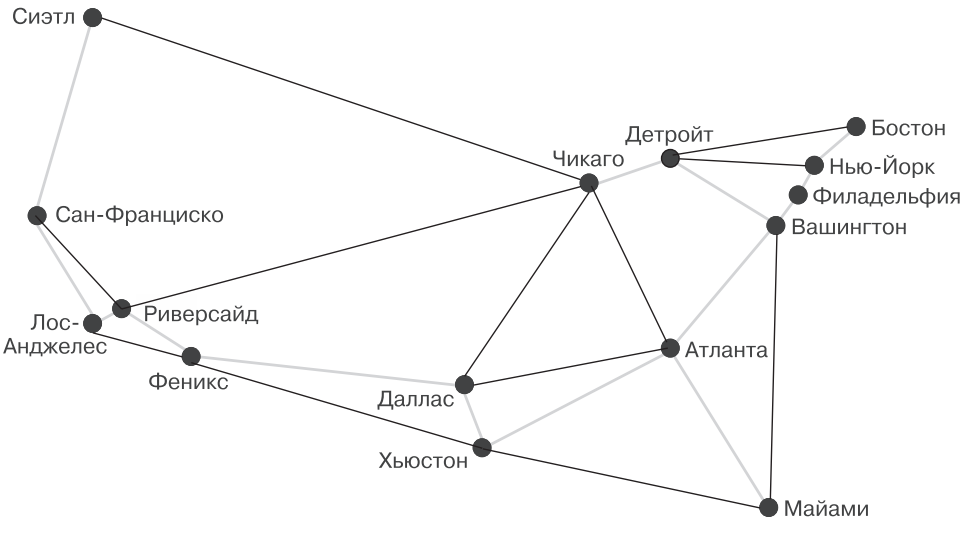


Рис. 4.7. Выделенные ребра образуют минимальное связующее дерево, которое соединяет все 15 MSA

4.5. ПОИСК КРАТЧАЙШИХ ПУТЕЙ ВО ВЗВЕШЕННОМ ГРАФЕ

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

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

4.5.1. Алгоритм Дейкстры

Алгоритм Дейкстры решает задачу поиска кратчайшего пути из одной вершины. Задается начальная вершина, и алгоритм возвращает путь с наименьшим весом к любой другой вершине во взвешенном графе. Он также возвращает минимальный общий вес для пути из начальной вершины к каждой из оставшихся. Алгоритм Дейкстры начинается из одной исходной вершины, а затем постоянно исследует

ближайшие к ней вершины. По этой причине алгоритм Дейкстры, как и алгоритм Ярника, является жадным. Когда алгоритм Дейкстры исследует новую вершину, он проверяет, как далеко она находится от начальной вершины, и обновляет это значение, если находит более короткий путь. Подобно алгоритму поиска в ширину, он отслеживает, какие ребра ведут к каждой вершине.

Алгоритм Дейкстры выполняет следующие шаги.

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

Код алгоритма Дейкстры включает в себя `DijkstraNode` — простую структуру данных для отслеживания затрат, связанных с каждой исследованной вершиной, и сравнения вершин. Это похоже на класс `Node`, описанный в главе 2. Он также включает в себя вспомогательные функции для преобразования возвращенного массива расстояний во что-то более простое, удобное для поиска по вершине и вычисления кратчайшего пути до конкретной конечной вершины из словаря путей, возвращаемого `dijkstra()`.

Чем долго рассуждать, лучше сразу приведем код алгоритма Дейкстры (листинг 4.13). Разберем его построчно. Весь код находится в файле `WeightedGraph.java`.

Листинг 4.13. WeightedGraph.java (продолжение)

```
    public static final class DijkstraNode implements Comparable<DijkstraNode> {
        public final int vertex;
        public final double distance;

        public DijkstraNode(int vertex, double distance) {
            this.vertex = vertex;
            this.distance = distance;
        }

        @Override
        public int compareTo(DijkstraNode other) {
            Double mine = distance;
            Double theirs = other.distance;
```

```
        return mine.compareTo(theirs);
    }
}

public static final class DijkstraResult {
    public final double[] distances;
    public final Map<Integer, WeightedEdge> pathMap;
    public DijkstraResult(double[] distances, Map<Integer, WeightedEdge>
      pathMap) {
        this.distances = distances;
        this.pathMap = pathMap;
    }
}

public DijkstraResult dijkstra(V root) {
    int first = indexOf(root); // найти начальный индекс
    // вначале расстояния неизвестны
    double[] distances = new double[getVertexCount()];
    distances[first] = 0; // корневая вершина равна 0
    boolean[] visited = new boolean[getVertexCount()];
    visited[first] = true;
    // как добраться до каждой вершины?
    HashMap<Integer, WeightedEdge> pathMap = new HashMap<>();
    PriorityQueue<DijkstraNode> pq = new PriorityQueue<>();
    pq.offer(new DijkstraNode(first, 0));

    while (!pq.isEmpty()) {
        int u = pq.poll().vertex; // исследовать ближайшую вершину
        double distU = distances[u]; // это мы уже, должно быть, видели
        // все ребра и вершины для данной вершины
        for (WeightedEdge we : edgesOf(u)) {
            // старое расстояние до этой вершины
            double distV = distances[we.v];
            // новое расстояние до этой вершины
            double pathWeight = we.weight + distU;
            // новая вершина или найден более короткий путь?
            if (!visited[we.v] || (distV > pathWeight)) {
                visited[we.v] = true;
                // изменить расстояние до этой вершины
                distances[we.v] = pathWeight;
                // заменить ребро на более короткий путь к этой вершине
                pathMap.put(we.v, we);
                // вскоре мы это проверим
                pq.offer(new DijkstraNode(we.v, pathWeight));
            }
        }
    }

    return new DijkstraResult(distances, pathMap);
}

// Вспомогательная функция для получения удобного доступа
// к результатам алгоритма Дейкстры
public Map<V, Double> distanceArrayToDistanceMap(double[] distances) {
```

```
        HashMap<V, Double> distanceMap = new HashMap<>();
        for (int i = 0; i < distances.length; i++) {
            distanceMap.put(vertexAt(i), distances[i]);
        }
        return distanceMap;
    }

    // Принимает словарь ребер, позволяющих достичь каждого узла,
    // и возвращает список ребер от start до end
    public static List<WeightedEdge> pathMapToPath(int start, int end,
      Map<Integer, WeightedEdge> pathMap) {
        if (pathMap.size() == 0) {
            return List.of();
        }
        LinkedList<WeightedEdge> path = new LinkedList<>();
        WeightedEdge edge = pathMap.get(end);
        path.add(edge);
        while (edge.u != start) {
            edge = pathMap.get(edge.u);
            path.add(edge);
        }
        Collections.reverse(path);
        return path;
    }
```

В первых строках `dijkstra()` используются структуры данных, с которыми вы уже знакомы, за исключением `distances` — заполнителя для расстояний до каждой вершины графа от `root`. Первоначально все эти расстояния равны `0`, так как мы еще не знаем значения каждого из них, — именно для их определения и применяем алгоритм Дейкстры!

```
    public DijkstraResult dijkstra(V root) {
        int first = indexOf(root); // найти начальную вершину
        // поначалу расстояния неизвестны
        double[] distances = new double[getVertexCount()];
        distances[first] = 0; // корневая вершина равна 0
        boolean[] visited = new boolean[getVertexCount()];
        visited[first] = true;
        // как добраться до каждой вершины
        HashMap<Integer, WeightedEdge> pathMap = new HashMap<>();
        PriorityQueue<DijkstraNode> pq = new PriorityQueue<>();
        pq.offer(new DijkstraNode(first, 0));
```

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

```
        while (!pq.isEmpty()) {
            int u = pq.poll().vertex; // исследовать ближайшую вершину
            double distU = distances[u]; // это мы уже, должно быть, видели
```

Мы будем выполнять алгоритм Дейкстры до тех пор, пока очередь с приоритетом не опустеет. `U` — текущая вершина, с которой начинается поиск,

а distU — сохраненное расстояние, позволяющее добраться до U по известным маршрутам. Все вершины, исследованные на этом этапе, уже найдены, поэтому для них расстояния уже известны.

```
// рассмотреть все ребра и вершины для данной вершины
for (WeightedEdge we : edgesOf(u)) {
    // старое расстояние до этой вершины
    double distV = distances[we.v];
    // новое расстояние до этой вершины
    double pathWeight = we.weight + distU;
```

Затем исследуются все ребра, связанные с u. distV — это расстояние до всех известных вершин, соединенных ребром с u. pathWeight — это расстояние, на котором используется новый исследуемый маршрут.

```
// новая вершина или найден более короткий путь
if (!visited[we.v] || (distV > pathWeight)) {
    visited[we.v] = true;
    // изменить расстояние до этой вершины
    distances[we.v] = pathWeight;
    // заменить ребро на более короткий путь к этой вершине
    pathMap.put(we.v, we);
    // вскоре мы это проверим
    pq.offer(new DijkstraNode(we.v, pathWeight));
}
```

Если мы обнаружили вершину, которая еще не была исследована (!visited[we.v]), или нашли новый, более короткий путь к ней (distV>pathWeight), то записываем новое кратчайшее расстояние до v и ребро, которое привело нас туда. Наконец, помещаем все вершины с новыми путями в очередь с приоритетом.

```
return new DijkstraResult(distances, pathMap);
```

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

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

Листинг 4.14. WeightedGraph.java (продолжение)

```
System.out.println(); // пустая строка

DijkstraResult dijkstraResult = cityGraph2.dijkstra("Los Angeles");
```

```
Map<String, Double> nameDistance =
     cityGraph2.distanceArrayToDistanceMap(dijkstraResult.distances);
System.out.println("Distances from Los Angeles:");
nameDistance.forEach((name, distance) -> System.out.println(name + " : " +
     distance));

System.out.println(); // пустая строка

System.out.println("Shortest path from Los Angeles to Boston:");
List<WeightedEdge> path = pathMapToPath(cityGraph2.indexOf("Los Angeles"),
cityGraph2.indexOf("Boston"), dijkstraResult.pathMap);
cityGraph2.printWeightedPath(path);
```

Результат должен выглядеть примерно так:

```
Distances from Los Angeles:
New York : 2474.0
Detroit : 1992.0
Seattle : 1026.0
Chicago : 1754.0
Washington : 2388.0
Miami : 2340.0
San Francisco : 348.0
Atlanta : 1965.0
Phoenix : 357.0
Los Angeles : 0.0
Dallas : 1244.0
Philadelphia : 2511.0
Riverside : 50.0
Boston : 2605.0
Houston : 1372.0

Shortest path from Los Angeles to Boston:
Los Angeles 50.0> Riverside
Riverside 1704.0> Chicago
Chicago 238.0> Detroit
Detroit 613.0> Boston
Total Weight: 2605.0
```

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

ПРИМЕЧАНИЕ

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

4.6. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

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

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

Одни из самых очевидных приложений графовых алгоритмов — это социальные сети и картографические приложения. В социальной сети люди играют роль вершин, а связи между ними (такие, как дружба в Facebook) — ребер. Один из известнейших инструментов разработчика Facebook так и называется — Graph API (https://developers.facebook.com/docs/graph-api). В картографических приложениях, таких как Apple Maps и Google Maps, графовые алгоритмы задействуются для указания направлений и расчета времени в пути.

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

4.7. УПРАЖНЕНИЯ

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

5 Генетические алгоритмы

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

5.1. НЕМНОГО БИОЛОГИЧЕСКОЙ ТЕОРИИ

В биологии теория эволюции объясняет, каким образом генетическая мутация в сочетании с ограничениями окружающей среды с течением времени приводит к изменениям в организме, включая видообразование — создание новых видов. Механизм, обусловливающий то, что хорошо адаптированные организмы процветают, а хуже адаптированные погибают, известен как *естественный отбор*. В каждое поколение определенного вида входят особи с различными, а иногда и новыми чертами, которые возникают в результате генетической мутации. Все особи соревнуются за ограниченные ресурсы, чтобы выжить, и поскольку живых особей больше, чем ресурсов, то некоторые из них должны умереть.

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

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

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

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

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

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

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

для которых не существует быстрых детерминированных алгоритмов. В таких случаях генетические алгоритмы — хороший выбор.

5.2. ОБОБЩЕННЫЙ ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

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

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

- определять собственную жизнеспособность;
- реализовывать кроссинговер (объединяться с другим объектом того же типа, чтобы создавать потомков), другими словами, скрещиваться с другой хромосомой;
- мутировать — вносить в себя небольшое случайное изменение;
- копировать саму себя;
- сравнивать себя с другими хромосомами того же типа.

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

Листинг 5.1. Chromosome.java

```
package chapter5;

import java.util.List;

public abstract class Chromosome<T extends Chromosome<T>> implements
Comparable<T> {
    public abstract double fitness();

    public abstract List<T> crossover(T other);

    public abstract void mutate();

    public abstract T copy();

    @Override
    public int compareTo(T other) {
        Double mine = this.fitness();
```

```
        Double theirs = other.fitness();
        return mine.compareTo(theirs);
    }
}
```

ПРИМЕЧАНИЕ

Вы заметите, что общий тип T хромосомы привязан к самой хромосоме (Chromosome<T extends Chromosome <T>>). Это означает, что все, что заменяет тип T, должно быть подклассом Chromosome. Это будет полезно применять с методами crossover(), copy() и compareTo(), поскольку мы хотим, чтобы реализации этих методов работали по отношению к другим хромосомам того же типа.

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

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

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

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

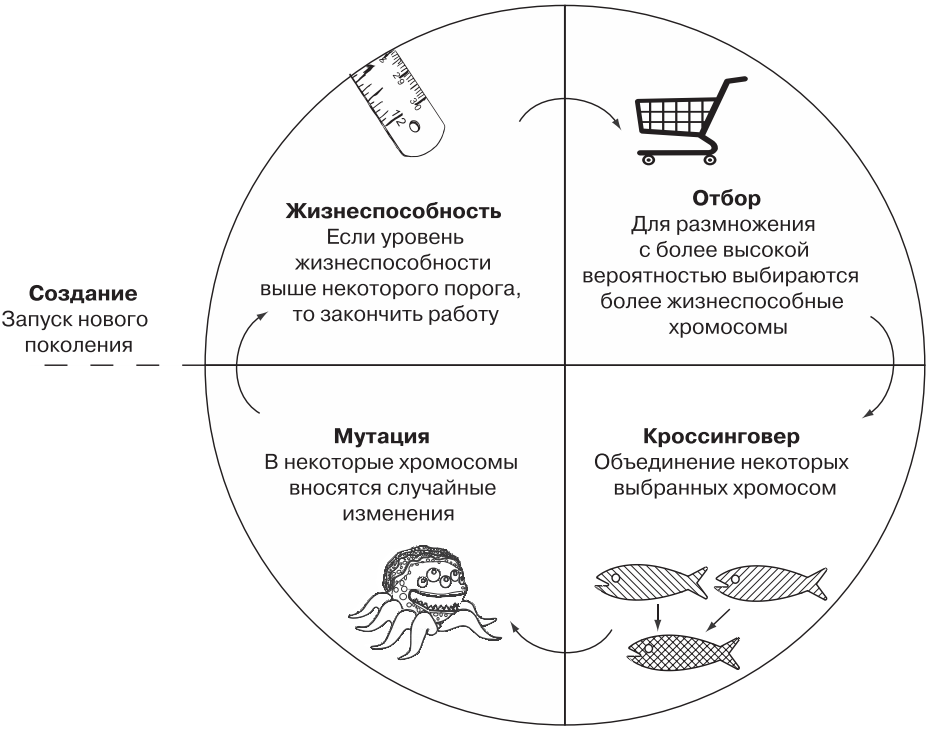


Рис. 5.1. Обобщенная схема генетического алгоритма

Листинг 5.2. GeneticAlgorithm.java

```
package chapter5;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class GeneticAlgorithm<C extends Chromosome<C>> {

    public enum SelectionType {
        ROULETTE, TOURNAMENT;
    }
```

`GeneticAlgorithm` принимает параметризованный тип, который соответствует `Chromosome` и называется `C`. Перечисление `SelectionType` является внутренним типом, используемым для определения метода отбора, применяемого в алгоритме. Существует два наиболее распространенных метода отбора для генетического алгоритма — *отбор методом рулетки* (иногда называемый *пропорциональным отбором по жизнеспособности*) и *турнирный отбор*. Первый дает каждой хромосоме

шанс быть отобранной пропорционально ее жизнеспособности. При турнирном отборе определенное количество случайно выбранных хромосом борются друг с другом и выбираются обладающие лучшей жизнеспособностью (листинг 5.3).

Листинг 5.3. GeneticAlgorithm.java (продолжение)

```
    private ArrayList<C> population;
    private double mutationChance;
    private double crossoverChance;
    private SelectionType selectionType;
    private Random random;

    public GeneticAlgorithm(List<C> initialPopulation, double mutationChance,
      double crossoverChance, SelectionType selectionType) {
        this.population = new ArrayList<>(initialPopulation);
        this.mutationChance = mutationChance;
        this.crossoverChance = crossoverChance;
        this.selectionType = selectionType;
        this.random = new Random();
    }
```

В приведенном коде представлены некоторые свойства генетического алгоритма, которые будут настроены в момент создания посредством `initialPopulation` — хромосомы первого поколения алгоритма. `mutationChance` — это вероятность мутации каждой хромосомы в каждом поколении. `crossoverChance` — вероятность того, что у двух родителей, отобранных для размножения, появятся потомки, представляющие собой смесь родительских генов, в противном случае они будут просто дубликатами родителей. Наконец, `selectionType` — это тип метода отбора, описанный в перечислении `SelectionType`.

В наших примерах `population` инициализируется случайным набором хромосом. Другими словами, первое поколение хромосом состоит из случайных особей. Это точка потенциальной оптимизации для более сложного генетического алгоритма. Вместо того чтобы начинать с чисто случайных особей, благодаря некоторому знанию конкретной задачи можно взять первое поколение, содержащее особей, находящихся ближе к решению. Это называется *посевом*.

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

Листинг 5.4. GeneticAlgorithm.java (продолжение)

```
    // используем метод рулетки чтобы выбрать родителей
    private List<C> pickRoulette(double[] wheel, int numPicks) {
        List<C> picks = new ArrayList<>();
        for (int i = 0; i < numPicks; i++) {
            double pick = random.nextDouble();
            for (int j = 0; j < wheel.length; j++) {
                pick -= wheel[j];
```

```
                if (pick <= 0) { // не работает при отрицательных
                                 // значениях жизнеспособности
                    picks.add(population.get(j));
                    break;
                }
            }
        }
        return picks;
    }
```

Выбор методом рулетки основан на отношении жизнеспособности каждой хромосомы к суммарной жизнеспособности всех хромосом данного поколения. Хромосомы с самой высокой жизнеспособностью имеют больше шансов быть отобранными. Значения, которые соответствуют процентной доле от общей жизнеспособности хромосомы, указаны в параметре `wheel`. Эти проценты представлены значениями с плавающей запятой от 0 до 1. Случайное число (`pick`) от 0 до 1 можно использовать для вычисления того, какую хромосому отобрать. Алгоритм будет работать, последовательно уменьшая `pick` на величину пропорциональной жизнеспособности каждой хромосомы. Когда `pick` станет меньше 0, это и будет хромосома для отбора.

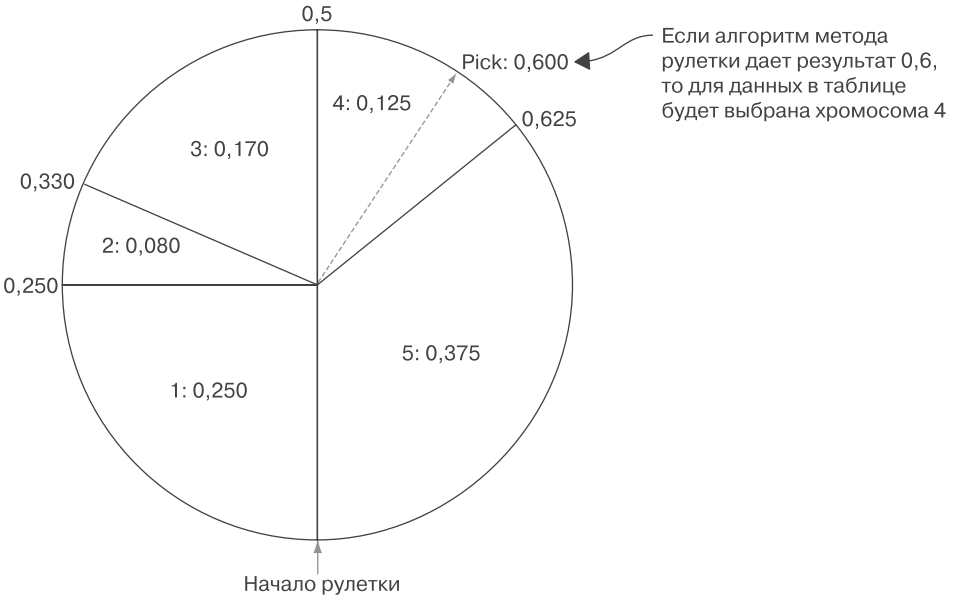
Понимаете ли вы, почему этот процесс приводит к тому, что каждая хромосома выбирается по ее пропорциональной жизнеспособности? Если нет, подумайте об этом с карандашом и бумагой. Попробуйте нарисовать метод пропорционального отбора, как показано на рис. 5.2.

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

Листинг 5.5. GeneticAlgorithm.java (продолжение)

```
    // Посредством турнирного отбора выбираем определенное число хромосом
    private List<C> pickTournament(int numParticipants, int numPicks) {
        // Находим случайным образом numParticipants для участия в турнирном
        // отборе
        Collections.shuffle(population);
        List<C> tournament = population.subList(0, numParticipants);
        // Выбираем numPicks хромосом с наилучшей жизнеспособностью
        Collections.sort(tournament, Collections.reverseOrder());
        return tournament.subList(0, numPicks);
    }
```

В коде для `pickTournament()` вначале используется `shuffle()`, чтобы случайным образом выбрать `numParticipants` из `population`. Это простой способ получить случайные хромосомы `numParticipants`. Затем выполним сортировку этих хромосом по их пригодности и получим `numPicks` наиболее подходящих хромосом.



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

Рис. 5.2. Пример отбора методом рулетки

Каково правильное количество для `numParticipants`? Как и во многих других параметрах генетического алгоритма, наилучшим способом определения этого значения может быть метод проб и ошибок. Следует иметь в виду, что чем больше участников турнира, тем меньше разнообразие в популяции, потому что хромосомы с плохой жизнеспособностью с большей вероятностью будут устранены в процессе поединков между особями[1]. Более сложные формы турнирного отбора могут выбирать особей не самых лучших, а лишь вторых или третьих по жизнеспособности, основываясь на некоторой модели убывающей вероятности.

Методы `pickRoulette()` и `pickTournament()` используются для отбора, выполняемого в процессе размножения. Воспроизводство реализовано в методе

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

`reproduceAndReplace()`, который заботится о том, чтобы на смену хромосомам последнего поколения пришло то же количество хромосом новой популяции (листинг 5.6).

Листинг 5.6. GeneticAlgorithm.java (продолжение)

```
// Замена популяции новым поколением особей
private void reproduceAndReplace() {
    ArrayList<C> nextPopulation = new ArrayList<>();
    // продолжаем, пока не заполним особями все новое поколение
    while (nextPopulation.size() < population.size()) {
        // выбор двух родителей
        List<C> parents;
        if (selectionType == SelectionType.ROULETTE) {
            // создание метода рулетки
            double totalFitness = population.stream()
                    .mapToDouble(C::fitness).sum();
            double[] wheel = population.stream()
                    .mapToDouble(C -> C.fitness()
                        / totalFitness).toArray();
            parents = pickRoulette(wheel, 2);
        } else { // турнирный отбор
            parents = pickTournament(population.size() / 2, 2);
        }
        // потенциальное скрещивание двух родителей
        if (random.nextDouble() < crossoverChance) {
            C parent1 = parents.get(0);
            C parent2 = parents.get(1);
            nextPopulation.addAll(parent1.crossover(parent2));
        } else { // добавление двух родителей
            nextPopulation.addAll(parents);
        }
    }
    if (nextPopulation.size() > population.size()) {
        nextPopulation.remove(0);
    }
    // заменяем ссылку/поколение
    population = nextPopulation;
}
```

В `reproduceAndReplace()` выполняются следующие основные операции.

1. Две хромосомы, называемые родителями (`parents`), отбираются для воспроизводства посредством одного из двух методов отбора. При турнирном отборе всегда проводится турнир среди половины популяции, но этот способ можно настраивать в конфигурации.
2. Существует вероятность `crossoverChance` того, что для получения двух новых хромосом два родителя будут объединены. В этом случае они добавляются в новую популяцию (`nextPopulation`). Если потомков нет, то два родителя просто добавляются в `nextPopulation`.

3. Если новая популяция `nextPopulation` содержит столько же хромосом, сколько и старая `population`, то она ее заменяет. В противном случае возвращаемся к шагу 1.

Метод `mutate()`, который реализует мутацию, очень прост, подробная реализация мутации предоставляется отдельным хромосомам (листинг 5.7). Каждая из наших хромосом знает, как мутироваться.

Листинг 5.7. GeneticAlgorithm.java (продолжение)

```
    // Каждая особь мутирует с вероятностью mutationChance
    private void mutate() {
        for (C individual : population) {
            if (random.nextDouble() < mutationChance) {
                individual.mutate();
            }
        }
    }
```

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

Листинг 5.8. GeneticAlgorithm.java (продолжение)

```
    // Выполнение генетического алгоритма для maxGenerations итераций
    // и возвращение лучшей из найденных особей
    public C run(int maxGenerations, double threshold) {
        C best = Collections.max(population).copy();
        for (int generation = 0; generation < maxGenerations; generation++) {
            // досрочный выход при достижении порога
            if (best.fitness() >= threshold) {
                return best;
            }
            // печать результатов отладки
            System.out.println("Generation " + generation +
                    " Best " + best.fitness() +
                    " Avg " + population.stream()
                .mapToDouble(C::fitness).average().orElse(0.0));
            reproduceAndReplace();
            mutate();
            C highest = Collections.max(population);
            if (highest.fitness() > best.fitness()) {
                best = highest.copy();
            }
        }
        return best;
    }
}
```

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

5.3. ПРИМИТИВНЫЙ ТЕСТ

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

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

Листинг 5.9. SimpleEquation.java

```
package chapter5;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class SimpleEquation extends Chromosome<SimpleEquation> {
    private int x, y;

    private static final int MAX_START = 100;

    public SimpleEquation(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public static SimpleEquation randomInstance() {
        Random random = new Random();
        return new SimpleEquation(random.nextInt(MAX_START),
            random.nextInt(MAX_START));
```

```
}

// 6x − x^2 + 4y − y^2
@Override
public double fitness() {
    return 6 * x − x * x + 4 * y − y * y;
}

@Override
public List<SimpleEquation> crossover(SimpleEquation other) {
    SimpleEquation child1 = new SimpleEquation(x, other.y);
    SimpleEquation child2 = new SimpleEquation(other.x, y);
    return List.of(child1, child2);
}

@Override
public void mutate() {
    Random random = new Random();
    if (random.nextDouble() > 0.5) { // мутация x
        if (random.nextDouble() > 0.5) {
            x += 1;
        } else {
            x -= 1;
        }
    } else { // иначе мутация y
        if (random.nextDouble() > 0.5) {
            y += 1;
        } else {
            y -= 1;
        }
    }

}

@Override
public SimpleEquation copy() {
    return new SimpleEquation(x, y);
}

@Override
public String toString() {
    return "X: " + x + " Y: " + y + " Fitness: " + fitness();
}
```

`SimpleEquation` соответствует `Chromosome` и согласно своему названию делает это максимально просто,. Гены хромосомы `SimpleEquation` можно рассматривать как x и y.

Метод `fitness()` оценивает жизнеспособность x и y с помощью уравнения $6x - x^2 + 4y - y^2$. Чем больше значение, тем выше жизнеспособность данной хромосомы в соответствии с `GeneticAlgorithm`. При использовании случайного экземпляра x и y изначально присваиваются случайные целые числа из диапазона 0...100, поэтому

`randomInstance()` необходимо только создать новый экземпляр `SimpleEquation` с этими значениями. Чтобы скрестить два `SimpleEquation` в `crossover()`, значения y этих экземпляров просто меняются местами для создания двух дочерних элементов. `mutate()` случайным образом увеличивает или уменьшает x или y. В общих чертах это все.

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

Листинг 5.10. SimpleEquation.java (продолжение)

```
    public static void main(String[] args) {
        ArrayList<SimpleEquation> initialPopulation = new ArrayList<>();
        final int POPULATION_SIZE = 20;
        final int GENERATIONS = 100;
        final double THRESHOLD = 13.0;
        for (int i = 0; i < POPULATION_SIZE; i++) {
            initialPopulation.add(SimpleEquation.randomInstance());
        }
        GeneticAlgorithm<SimpleEquation> ga = new GeneticAlgorithm<>(
                initialPopulation,
                0.1, 0.7, GeneticAlgorithm.SelectionType.TOURNAMENT);
                SimpleEquation result = ga.run(100, 13.0);
                System.out.println(GENERATIONS, THRESHOLD);
        }
    }
```

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

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

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

```
Generation 0 Best -72.0 Avg -4436.95
Generation 1 Best 9.0 Avg -579.0
Generation 2 Best 9.0 Avg -38.15
Generation 3 Best 12.0 Avg 9.0
Generation 4 Best 12.0 Avg 9.2
Generation 5 Best 12.0 Avg 11.25
Generation 6 Best 12.0 Avg 11.95
X: 3 Y: 2 Fitness: 13.0
```

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

Следует принять во внимание, что для поиска решения генетический алгоритм потребовал больше вычислительных ресурсов, чем другие методы. В реальном мире применение генетического алгоритма для столь простой задачи, как поиск максимума, едва ли будет хорошей идеей. Но этой простой реализации достаточно, чтобы показать: наш генетический алгоритм работает.

5.4. SEND + MORE = MONEY, УЛУЧШЕННЫЙ ВАРИАНТ

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

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

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

Листинг 5.11. SendMoreMoney2.java

```
package chapter5;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

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

```
import java.util.Random;

public class SendMoreMoney2 extends Chromosome<SendMoreMoney2> {

    private List<Character> letters;
    private Random random;

    public SendMoreMoney2(List<Character> letters) {
        this.letters = letters;
        random = new Random();
    }

    public static SendMoreMoney2 randomInstance() {
        List<Character> letters = new ArrayList<>(
                List.of('S', 'E', 'N', 'D', 'M', 'O', 'R', 'Y', ' ', ' '));
        Collections.shuffle(letters);
        return new SendMoreMoney2(letters);
    }

    @Override
    public double fitness() {
        int s = letters.indexOf('S');
        int e = letters.indexOf('E');
        int n = letters.indexOf('N');
        int d = letters.indexOf('D');
        int m = letters.indexOf('M');
        int o = letters.indexOf('O');
        int r = letters.indexOf('R');
        int y = letters.indexOf('Y');
        int send = s * 1000 + e * 100 + n * 10 + d;
        int more = m * 1000 + o * 100 + r * 10 + e;
        int money = m * 10000 + o * 1000 + n * 100 + e * 10 + y;
        int difference = Math.abs(money - (send + more));
        return 1.0 / (difference + 1.0);
    }

    @Override
    public List<SendMoreMoney2> crossover(SendMoreMoney2 other) {
        SendMoreMoney2 child1 = new SendMoreMoney2(new ArrayList<>(letters));
        SendMoreMoney2 child2 = new SendMoreMoney2(new ArrayList<>
     (other.letters));
        int idx1 = random.nextInt(letters.size());
        int idx2 = random.nextInt(other.letters.size());
        Character l1 = letters.get(idx1);
        Character l2 = other.letters.get(idx2);
        int idx3 = letters.indexOf(l2);
        int idx4 = other.letters.indexOf(l1);
        Collections.swap(child1.letters, idx1, idx3);
        Collections.swap(child2.letters, idx2, idx4);
        return List.of(child1, child2);
    }

    @Override
    public void mutate() {
```

```
        int idx1 = random.nextInt(letters.size());
        int idx2 = random.nextInt(letters.size());
        Collections.swap(letters, idx1, idx2);
    }

    @Override
    public SendMoreMoney2 copy() {
        return new SendMoreMoney2(new ArrayList<>(letters));
    }

    @Override
    public String toString() {
        int s = letters.indexOf('S');
        int e = letters.indexOf('E');
        int n = letters.indexOf('N');
        int d = letters.indexOf('D');
        int m = letters.indexOf('M');
        int o = letters.indexOf('O');
        int r = letters.indexOf('R');
        int y = letters.indexOf('Y');
        int send = s * 1000 + e * 100 + n * 10 + d;
        int more = m * 1000 + o * 100 + r * 10 + e;
        int money = m * 10000 + o * 1000 + n * 100 + e * 10 + y;
        int difference = Math.abs(money - (send + more));
        return (send + " + " + more + " = " + money + " Difference: " +
            difference);
}
```

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

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

difference	difference + 1	fitness (1 / (difference + 1))
0	1	1
1	2	0,5
2	3	0,33
3	4	0,25

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

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

`SendMoreMoney2` подключается к `GeneticAlgorithm` так же легко, как и `SimpleEquation` (листинг 5.12). Имейте в виду: это довольно сложная задача и ее выполнение займет много времени, если неудачно настроить исходные параметры. Но даже при правильной настройке существует определенный элемент случайности! Задача может быть решена в течение нескольких секунд или нескольких минут. К сожалению, такова природа генетических алгоритмов.

Листинг 5.12. SendMoreMoney2.java (продолжение)

```
    public static void main(String[] args) {
        ArrayList<SendMoreMoney2> initialPopulation = new ArrayList<>();
        final int POPULATION_SIZE = 1000;
        final int GENERATIONS = 1000;
        final double THRESHOLD = 1.0;
        for (int i = 0; i < POPULATION_SIZE; i++) {
            initialPopulation.add(SendMoreMoney2.randomInstance());
        }
        GeneticAlgorithm<SendMoreMoney2> ga = new GeneticAlgorithm<>(
                initialPopulation,
                0.2, 0.7, GeneticAlgorithm.SelectionType.ROULETTE);
                SendMoreMoney2 result = ga.run(GENERATIONS, THRESHOLD);
                System.out.println(result);
        }
    }
```

Следующие результаты были получены при запуске алгоритма, при котором задача была решена за три поколения с использованием 1000 особей в каждом

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

поколении, созданном ранее. Проверьте, удастся ли вам настроить параметры `GeneticAlgorithm` так, чтобы получить аналогичный результат с меньшим количеством особей. Вам не кажется, что этот алгоритм лучше работает с отбором по методу рулетки, чем с турнирным отбором?

```
Generation 0 Best 0.07142857142857142 Avg 2.588160841027962E-4
Generation 1 Best 0.16666666666666666 Avg 0.005418719421172926
Generation 2 Best 0.5 Avg 0.022271971406414452
8324 + 913 = 9237 Difference: 0
```

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

5.5. ОПТИМИЗАЦИЯ СЖАТИЯ СПИСКА

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

Ответ будет зависеть от используемого алгоритма сжатия. В этом примере применим класс `GZIPOutputStream` из модуля `java.util.zip`. Далее показано полное решение для списка из 12 имен (листинг 5.13). Если не брать генетический алгоритм, а просто запустить `compress()` для 12 имен в том порядке, в каком они были представлены, то получим сжатые данные объемом 164 байта.

Листинг 5.13. ListCompression.java

```
package chapter5;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.zip.GZIPOutputStream;

public class ListCompression extends Chromosome<ListCompression> {
    private static final List<String> ORIGINAL_LIST = List.of("Michael",
      "Sarah", "Joshua", "Narine", "David", "Sajid", "Melanie", "Daniel",
      "Wei", "Dean", "Brian", "Murat", "Lisa");
    private List<String> myList;
```

```
    private Random random;

    public ListCompression(List<String> list) {
        myList = new ArrayList<>(list);
        random = new Random();
    }

    public static ListCompression randomInstance() {
        ArrayList<String> tempList = new ArrayList<>(ORIGINAL_LIST);
        Collections.shuffle(tempList);
        return new ListCompression(tempList);
    }

    private int bytesCompressed() {
        try {
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            GZIPOutputStream gos = new GZIPOutputStream(baos);
            ObjectOutputStream oos = new ObjectOutputStream(gos);
            oos.writeObject(myList);
            oos.close();
            return baos.size();
        } catch (IOException ioe) {
            System.out.println("Could not compress list!");
            ioe.printStackTrace();
            return 0;
        }
    }

    @Override
    public double fitness() {
        return 1.0 / bytesCompressed();
    }

    @Override
    public List<ListCompression> crossover(ListCompression other) {
        ListCompression child1 = new ListCompression(new ArrayList<>(myList));
        ListCompression child2 = new ListCompression(new ArrayList<>(myList));
        int idx1 = random.nextInt(myList.size());
        int idx2 = random.nextInt(other.myList.size());
        String s1 = myList.get(idx1);
        String s2 = other.myList.get(idx2);
        int idx3 = myList.indexOf(s2);
        int idx4 = other.myList.indexOf(s1);
        Collections.swap(child1.myList, idx1, idx3);
        Collections.swap(child2.myList, idx2, idx4);
        return List.of(child1, child2);
    }

    @Override
    public void mutate() {
        int idx1 = random.nextInt(myList.size());
        int idx2 = random.nextInt(myList.size());
        Collections.swap(myList, idx1, idx2);
    }
```

```
    @Override
    public ListCompression copy() {
        return new ListCompression(new ArrayList<>(myList));
    }

    @Override
    public String toString() {
        return "Order: " + myList + " Bytes: " + bytesCompressed();
    }

    public static void main(String[] args) {
        ListCompression originalOrder = new ListCompression(ORIGINAL_LIST);
        System.out.println(originalOrder);
        ArrayList<ListCompression> initialPopulation = new ArrayList<>();
        final int POPULATION_SIZE = 100;
        final int GENERATIONS = 100;
        final double THRESHOLD = 1.0;
        for (int i = 0; i < POPULATION_SIZE; i++) {
            initialPopulation.add(ListCompression.randomInstance());
        }
        GeneticAlgorithm<ListCompression> ga = new GeneticAlgorithm<>(
                initialPopulation,
                0.2, 0.7, GeneticAlgorithm.SelectionType.TOURNAMENT);
        ListCompression result = ga.run(GENERATIONS, THRESHOLD);
        System.out.println(result);
    }
}
```

Обратите внимание на то, как эта реализация похожа на реализацию из SEND + MORE = MONEY (см. раздел 5.4). Функции `crossover()` и `mutate()` практически одинаковы. В решениях обеих задач мы берем список элементов, постоянно перестраиваем его и проверяем перестановки. Можно написать общий суперкласс для решения обеих задач, который работал бы с широким спектром задач. Любая задача, если ее можно представить в виде списка элементов, для которого требуется найти оптимальный порядок, может быть решена аналогичным образом. Единственная реальная точка настройки для этих подклассов — соответствующая функция жизнеспособности.

Выполнение `ListCompression.java` может занять очень много времени. Так происходит потому, что, в отличие от двух предыдущих задач, мы не знаем заранее, каков правильный ответ, поэтому у нас нет реального порога, к которому следовало бы стремиться. Вместо этого мы устанавливаем число поколений и число особей в каждом поколении как произвольное большое число и надеемся на лучшее. Каково минимальное количество байтов, которое удастся получить при сжатии в результате перестановки 12 имен? Честно говоря, мы не знаем ответа на этот вопрос. У меня в лучшем случае, используя конфигурацию из предыдущего решения, после 100 поколений генетический алгоритм нашел последовательность из 12 имен, которая дала сжатие размером 158 байт.

Всего лишь 6 байт по сравнению с первоначальным порядком — экономия составляет примерно 4 %. Можно сказать, что 4 % не имеют значения, но если бы это был гораздо больший список, который многократно передавался бы по сети, то экономия выросла бы многократно. Представьте, что получилось бы, если бы это был список размером 1 Мбайт, который передавался бы через интернет 10 000 000 раз. Если бы генетический алгоритм мог оптимизировать порядок списка для сжатия, чтобы сэкономить 4 %, то он сэкономил бы примерно 40 Кбайт на каждую передачу и в итоге 400 Гбайт пропускной способности для всех передач. Это не очень большая экономия, но она может оказаться значительной настолько, что будет иметь смысл один раз запустить алгоритм и получить почти оптимальный порядок сжатия.

Однако подумайте вот о чем: в действительности неизвестно, нашли ли мы оптимальную последовательность для 12 имен, не говоря уже о гипотетическом списке размером 1 Мбайт. Как об этом узнать? Поскольку мы не обладаем глубоким пониманием алгоритма сжатия, следует проверить степень сжатия для всех возможных последовательностей в списке. Но даже для списка всего лишь из 12 пунктов это трудновыполнимые 479 001 600 возможных вариантов (12!, где знак ! означает факториал). Использование генетического алгоритма, который пытается найти оптимальный вариант, возможно, будет более целесообразным, даже если мы не знаем, действительно ли его окончательное решение оптимально.

5.6. ПРОБЛЕМЫ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

Генетические алгоритмы — это не панацея. В сущности, для большинства задач они не подходят. Для любой задачи, для которой существует быстрый детерминированный алгоритм, подход с использованием генетического алгоритма не имеет смысла. Стохастическая природа генетических алгоритмов не позволяет предсказать время их выполнения. Чтобы решить эту проблему, можно прерывать работу алгоритма через определенное количество поколений. Но тогда остается неясным, было ли найдено действительно оптимальное решение.

Стивен Скиена, автор одного из самых популярных текстов об алгоритмах, даже написал следующее: *«Я никогда не сталкивался с задачей, для которой генетические алгоритмы показались бы мне подходящим способом решения. Кроме того, мне никогда не встречались вычислительные результаты, полученные с использованием генетических алгоритмов, которые произвели бы на меня благоприятное впечатление»*[1].

Мнение Скиены несколько радикально, но оно свидетельствует о том, что к генетическим алгоритмам следует прибегать, только когда вы совершенно уверены:

[1] *Skiena S.* The Algorithm Design Manual. — 2nd ed. — Springer, 2009.

лучшего решения не существует — или решаете неизвестную вам задачу. Еще одна проблема генетических алгоритмов — определить, как представить потенциальное решение задачи в виде хромосомы. Традиционная практика заключается в представлении большинства задач в виде двоичных строк (последовательности 1 и 0, необработанные биты). Зачастую это оптимально с точки зрения использования пространства и позволяет легко выполнять функции кроссинговера. Но большинство сложных задач не так легко представить в виде кратных битовых строк.

Еще одна, более специфичная особенность, на которую стоит обратить внимание, — это проблемы, связанные с отбором методом рулетки, описанным ранее. Отбор методом рулетки, иногда называемый *пропорциональным отбором по жизнеспособности*, может привести к отсутствию разнообразия в популяции из-за преобладания довольно хорошо подходящих особей при каждом отборе. В то же время, если значения жизнеспособности близки, отбор методом рулетки может привести к отсутствию давления отбора[1]. Кроме того, этот метод не работает в задачах, в которых жизнеспособность может принимать отрицательные значения, как в простом уравнении в разделе 5.3.

Короче говоря, для большинства задач, достаточно больших, чтобы для них стоило использовать генетические алгоритмы, последние не могут гарантировать поиск оптимального решения за предсказуемое время. По этой причине такие алгоритмы лучше всего применять в ситуациях, которые требуют найти не оптимальное, а лишь приемлемое решение. Эти алгоритмы довольно легко реализовать, но настройка их параметров может потребовать множества проб и ошибок.

5.7. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

Несмотря на негативное мнение Скиена, генетические алгоритмы часто и эффективно применяются в многочисленных пространствах задач. Их нередко используют для сложных задач, которые не требуют абсолютно оптимальных решений, таких как задачи с ограничениями, если они слишком велики, чтобы их можно было решить с помощью традиционных методов. Один из примеров таких задач — сложные проблемы планирования.

Генетические алгоритмы нашли широкое применение в вычислительной биологии. Они были успешно использованы для соединения белка-лиганда, при котором требовался поиск конфигурации маленькой молекулы, связанной с реципиентом. Эти алгоритмы используются в фармацевтических исследованиях и для того, чтобы лучше понять механизмы природных явлений.

---

[1] *Eiben A. E., Smith J. E.* Introduction to Evolutionary Computation. — 2nd ed. — Springer, 2015.

Одна из самых известных задач в области компьютерных наук — это задача коммивояжера, к которой мы еще вернемся в главе 9. Коммивояжер желает найти по карте кратчайший маршрут, чтобы посетить каждый город ровно один раз и вернуться в исходную точку. Эта формулировка может напомнить о минимальных связующих деревьях из главы 4, но в действительности это не так. Решение задачи коммивояжера представляет собой гигантский цикл, который сводит к минимуму затраты на его прохождение, тогда как минимальное связующее дерево минимизирует затраты на подключение каждого города. Человеку, путешествующему через минимальное связующее дерево городов, возможно, придется дважды посетить один и тот же город, чтобы добраться до каждого города. Несмотря на то что задачи выглядят похожими, не существует разумно рассчитанного алгоритма для поиска решения задачи коммивояжера для произвольного числа городов. Как было показано, генетические алгоритмы находят неоптимальные, но довольно хорошие решения за короткий промежуток времени. Эта задача широко применяется для эффективного распределения товаров. Например, диспетчеры грузовых автомобилей служб FedEx и UPS используют программное обеспечение для решения задачи коммивояжера в повседневной деятельности. Алгоритмы, помогающие решить эту задачу, позволяют сократить расходы в самых разных отраслях.

В компьютерном искусстве генетические алгоритмы иногда применяются для имитации фотографий с помощью стохастических методов. Представьте себе 50 многоугольников, случайным образом размещенных на экране и постепенно скручиваемых, поворачиваемых, перемещаемых, изменяющих размеры и цвет до тех пор, пока они не будут как можно точнее соответствовать фотографии. Результат выглядит как работа художника-абстракциониста или, если задействовать более угловатые формы, как витраж.

Генетические алгоритмы — это часть более широкой области, называемой эволюционными вычислениями. Одной из областей эволюционных вычислений, тесно связанной с генетическими алгоритмами, является *генетическое программирование*, при котором программы задействуют операции отбора, кроссинговера и мутации, чтобы, изменяя себя, найти неочевидные решения задач программирования. Генетическое программирование — не слишком широко используемая технология, но представьте себе будущее, в котором программы пишут себя сами.

Преимущество генетических алгоритмов состоит в том, что они легко поддаются распараллеливанию. В наиболее очевидной форме каждая популяция может быть смоделирована на отдельном процессоре. В более детализированном варианте для каждого индивида выделяется особый поток, в котором он может мутировать и участвовать в кроссинговере и где вычисляется его жизнеспособность. Существует также множество промежуточных возможностей.

5.8. УПРАЖНЕНИЯ

1. Добавьте в `GeneticAlgorithm` поддержку расширенной формы турнирного отбора, которая может иногда выбирать вторую или третью по жизнеспособности хромосому, основываясь на уменьшающейся вероятности.
2. Добавьте к структуре с ограничениями, описанной в главе 3, новую функцию, которая решает любой произвольный CSP с использованием генетического алгоритма. Возможной мерой жизнеспособности является количество ограничений, которые разрешаются данной хромосомой.
3. Создайте класс `BitString`, который реализует `Chromosome`. Для того чтобы вспомнить, что представляет собой битовая строка, обратитесь к главе 1. Затем примените новый класс для решения задачи простого уравнения из раздела 5.3. Как можно представить эту задачу в виде битовой строки?

6 Кластеризация методом k-средних

Человечество еще никогда не имело так много данных о столь многочисленных аспектах общественной жизни, как сегодня. Компьютеры отлично справляются с хранением наборов данных, но последние мало что значат для общества, пока не будут проанализированы людьми. Вычислительные методы способны направлять людей по пути извлечения смысла из наборов данных.

Кластеризация — это вычислительная технология, которая делит все единицы данных из набора на группы. Успешная кластеризация приводит к созданию групп, которые содержат единицы данных, связанные между собой. Для того чтобы выяснить, значимы ли эти отношения, обычно требуется проверка человеком.

При кластеризации группа, называемая *кластером*, к которой принадлежит единица данных, не предопределяется, а определяется во время выполнения алгоритма кластеризации. В сущности, целью алгоритма не является размещение какой-либо конкретной единицы данных в каком-либо конкретном кластере на основании некоей заранее известной информации. По этой причине кластеризация считается *неконтролируемым* методом при машинном обучении. *Неконтролируемость* можно представить как *независимость от чего-то, что известно заранее*.

Кластеризация — это полезная технология, если требуется изучить структуру набора данных, но заранее ничего не известно о ее составных частях. Например, представьте, что у вас есть продуктовый магазин и вы собираете данные о клиентах и их покупках. Чтобы привлечь клиентов в магазин, вы намерены рассылать мобильную рекламу о специальных предложениях в соответствующие дни недели. Можете попробовать кластеризовать данные по дням недели и демографической информации. Возможно, вы найдете кластер, указывающий на то, что молодые

покупатели предпочитают делать покупки по вторникам, и тогда сможете использовать данную информацию для показа в этот день рекламы, специально ориентированной на них.

6.1. ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

Алгоритм кластеризации потребует некоторых статистических примитивов (среднее значение, стандартное отклонение и т. д.). Начиная с Java версии 8 в стандартную библиотеку языка в модуль `util` входит несколько полезных статистических примитивов по классу `DoubleSummaryStatistics`. Мы будем использовать эти примитивы для разработки более сложной статистики. Следует отметить: хотя в книге мы и придерживаемся стандартной библиотеки, существуют более эффективные сторонние библиотеки, которые следует применять в приложениях, для которых важна производительность, особенно для работы с большими объемами данных. Встроенная библиотека всегда будет лучшим выбором с точки зрения производительности и возможностей, чем разработанная вами. Однако в этой книге мы учимся на собственном опыте.

Для простоты все наборы данных, с которыми мы будем работать в этой главе, можно выразить с помощью типа `double` или его объектного эквивалента `Double`. В следующем классе `Statistics` статистические примитивы `sum()`, `mean()`, `max()` и `min()` реализуются через `DoubleSummaryStatistics`. `variance()`, `std()` (стандартное отклонение) и `zscored()` построены на основе этих примитивов. Их определения вытекают из формул, которые вы найдете в любом учебнике по статистике (листинг 6.1).

Листинг 6.1. Statistics.java

```
package chapter6;

import java.util.DoubleSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;

public final class Statistics {
    private List<Double> list;
    private DoubleSummaryStatistics dss;

    public Statistics(List<Double> list) {
        this.list = list;
        dss = list.stream().collect(Collectors.summarizingDouble(d -> d));
    }

    public double sum() {
        return dss.getSum();
    }
```

```
    // Найдите среднее значение (mean)
    public double mean() {
        return dss.getAverage();
    }

    // Найдите сумму дисперсии ((Xi – среднее) ^ 2) / N
    public double variance() {
        double mean = mean();
        return list.stream().mapToDouble(x -> Math.pow((x – mean), 2))
                .average().getAsDouble();
    }

    // Найдите стандартное отклонение sqrt (дисперсия)
    public double std() {
        return Math.sqrt(variance());
    }

    // Преобразование элементов в соответствующие z-значения
    // (формула z-score = (x-mean) / std)
    public List<Double> zscored() {
        double mean = mean();
        double std = std();
        return list.stream()
                .map(x -> std != 0 ? ((x – mean) / std) : 0.0)
                .collect(Collectors.toList());
    }

    public double max() {
        return dss.getMax();
    }

    public double min() {
        return dss.getMin();
    }
}
```

СОВЕТ

Функция variance() находит дисперсию популяции. Несколько иная формула, которую мы не используем, находит дисперсию отбора. Мы всегда будем оценивать все данные одновременно.

Функция `zscores()` преобразует последовательность чисел в список чисел с соответствующими *z*-оценками исходных чисел относительно всех чисел в исходной последовательности. Подробнее о *z*-оценках будет рассказано позже в этой главе.

ПРИМЕЧАНИЕ

Обучение элементарной статистике выходит за рамки данной книги, но чтобы разобраться в содержании оставшейся части главы, вам понадобятся лишь элементарные знания о среднем и стандартном отклонениях. Если вы успели забыть, что это такое, и хотите освежить знания или же никогда ранее не изучали эти термины, возможно, стоит бегло просмотреть какой-нибудь ресурс, посвященный статистике, где объясняются эти фундаментальные понятия.

Все алгоритмы кластеризации работают с единицами данных, и наша реализация метода *k*-средних (*k*-means) не станет исключением. Мы определим общий класс под названием `DataPoint`. Из соображений ясности кода определим его в отдельном файле (листинг 6.2).

Листинг 6.2. DataPoint.java

```
package chapter6;

import java.util.ArrayList;
import java.util.List;

public class DataPoint {
    public final int numDimensions;
    private List<Double> originals;
    public List<Double> dimensions;
    public DataPoint(List<Double> initials) {
        originals = initials;
        dimensions = new ArrayList<>(initials);
        numDimensions = dimensions.size();
    }

    public double distance(DataPoint other) {
        double differences = 0.0;
        for (int i = 0; i < numDimensions; i++) {
            double difference = dimensions.get(i) – other.dimensions.get(i);
            differences += Math.pow(difference, 2);
        }
        return Math.sqrt(differences);
    }

    @Override
    public String toString() {
        return originals.toString();
    }
}
```

Каждая единица данных должна быть удобочитаемой для вывода при отладке (`toString()`). Каждый тип единиц данных имеет определенное количество измерений (`numDimensions`). В кортеже `dimensions` хранятся фактические значения для каждого из этих измерений в виде чисел типа `doubles`. Конструктор принимает список исходных значений. Впоследствии эти измерения могут быть заменены с помощью *k*-средних на *z*-оценки, поэтому мы также сохраняем в `originals` копию исходных данных для последующего вывода.

И последнее, что нам нужно, прежде чем углубиться в алгоритм *k*-средних, — это способ вычисления расстояния между любыми двумя единицами данных одного типа. Существует много способов вычисления расстояния, но наиболее распространенная форма, используемая для *k*-средних, — евклидово расстояние. Это формула вычисления расстояния по теореме Пифагора, которую большинство из нас знает из школьных уроков геометрии. В сущности, мы уже обсуждали эту

формулу и вывели ее версию для двумерных пространств в главе 2, где с ее помощью искали расстояние между любыми двумя точками в лабиринте. Версия для `DataPoint` должна быть более сложной, поскольку `DataPoint` может включать в себя любое количество измерений.

Квадраты каждой из разностей суммируются, и окончательное значение, возвращаемое функцией `distance()`, является квадратным корнем из данной суммы.

6.2. АЛГОРИТМ КЛАСТЕРИЗАЦИИ МЕТОДОМ k-СРЕДНИХ

Метод *k*-средних — это алгоритм кластеризации, который стремится сгруппировать единицы данных в некое заранее определенное количество кластеров. В каждом периоде *k*-средних вычисляется расстояние между каждой единицей данных и каждым центром кластера — единицей данных, известной как *центроид*. Единицы данных присваиваются кластеру, к центроиду которого они ближе всего. Затем алгоритм пересчитывает все центроиды, находя среднее значение единиц данных, назначенных каждому кластеру, и заменяя старый центроид новым средним. Процесс назначения единиц данных и пересчета центроидов продолжается до тех пор, пока центроиды не перестанут передвигаться или не будет выполнено определенное количество итераций.

Все измерения начальных точек, представленных *k*-средними, должны быть сопоставимыми по величине. Иначе *k*-средние будут отклоняться в сторону кластеризации на основе измерений с наибольшим отличием. Процесс сопоставления разных типов данных (в нашем случае разных измерений) называется *нормализацией*. Одним из распространенных способов нормализации данных является приближенная оценка каждого значения на основе его *z-оценки*, известной также как *стандартная оценка*, относительно других значений того же типа. *Z*-оценка рассчитывается путем вычитания среднего всех значений из данного значения и деления результата на стандартное отклонение всех значений. Функция `zscores()`, разработанная в начале предыдущего раздела, делает именно это для каждого значения итерируемого объекта, состоящего из значений типа `doubles`.

Основная сложность, связанная с алгоритмом *k*-средних, заключается в определении способа выбора начальных центроидов. В простейшей форме алгоритма, которую мы будем реализовывать, начальные центроиды размещаются случайным образом в пределах диапазона данных. Сложно также решить, на сколько кластеров разделить данные (*k* в *k*-средних). В классическом алгоритме это число определяет пользователь, но он может не знать правильного числа, и его определение потребует некоторого количества экспериментов. Мы позволим пользователю определить *k*.

Объединяя все эти этапы и соображения, получим следующий алгоритм кластеризации методом *k*-средних.

1. Инициализировать все единицы данных и *k* пустых кластеров.
2. Нормализовать все единицы данных.
3. Создать случайные центроиды, связанные с каждым кластером.
4. Назначить каждую единицу данных кластеру того центроида, к которому она расположена ближе всего.
5. Пересчитать каждый центроид, чтобы он был центром (средним) кластера, с которым связан.
6. Повторять пункты 4 и 5 до тех пор, пока не будет выполнено максимально допустимое количество итераций или пока центроиды не перестанут передвигаться (сходиться).

Концептуально метод *k*-средних, в сущности, очень прост: на каждой итерации каждая единица данных связана с тем кластером, к центру которого она расположена ближе всего. По мере того как в кластер вносятся новые единицы данных, этот центр перемещается (рис. 6.1).

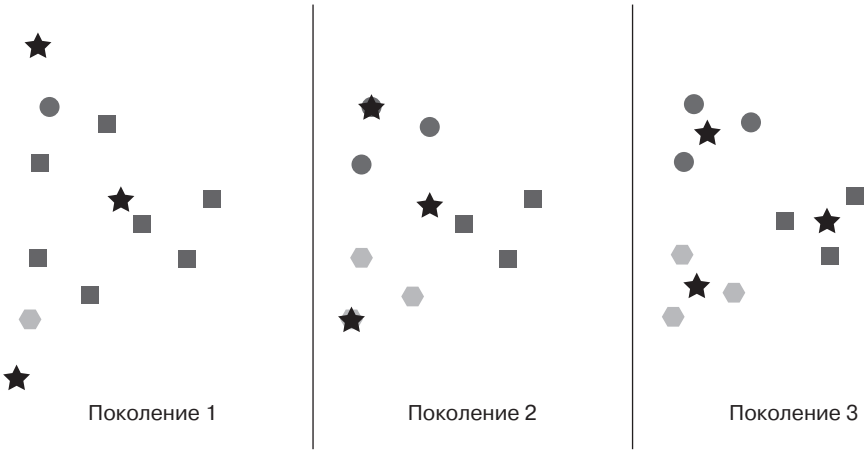


Рис. 6.1. Пример метода *k*-средних, выполняемого на протяжении трех поколений для произвольного набора данных. Звездочками отмечены центроиды. Оттенком и формой обозначено текущее членство в кластере (оно изменяется)

Далее реализован класс для поддержания состояния и выполнения алгоритма, аналогичный `GeneticAlgorithm`, описанному в главе 5. Теперь мы начнем с внутреннего класса, представляющего кластер (листинг 6.3).

Листинг 6.3. KMeans.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;

public class KMeans<Point extends DataPoint> {

    public class Cluster {
        public List<Point> points;
        public DataPoint centroid;
        public Cluster(List<Point> points, DataPoint randPoint) {
            this.points = points;
            this.centroid = randPoint;
        }
    }
```

`KMeans` — это параметризованный класс. Он работает с `DataPoint` или с любым подклассом `DataPoint`, как определено значением `bound` типа `Point` (`Point extends DataPoint`). У него есть внутренний класс `Cluster`, который отслеживает отдельные кластеры в процессе выполнения операции. У каждого `Cluster` есть связанные с ним единицы данных и центроид.

ПРИМЕЧАНИЕ

Чтобы уменьшить размер кода и сделать его более читабельным, мы позволяем некоторым переменным экземпляра быть общедоступными, к ним обычно можно получить доступ через геттеры/сеттеры.

Продолжим работу с конструктором для внешнего класса (листинг 6.4).

Листинг 6.4. KMeans.java (продолжение)

```
private List<Point> points;
private List<Cluster> clusters;

public KMeans(int k, List<Point> points) {
    if (k < 1) { // число кластеров не может быть
                 // отрицательным или равным нулю
        throw new IllegalArgumentException("k must be >= 1");
    }
    this.points = points;
    zScoreNormalize();
    // инициализируем пустые кластеры случайными центроидами
        clusters = new ArrayList<>();
        for (int i = 0; i < k; i++) {
            DataPoint randPoint = randomPoint();
            Cluster cluster = new Cluster(new ArrayList<Point>(), randPoint);
            clusters.add(cluster);
        }
    }
```

```
private List<DataPoint> centroids() {
    return clusters.stream().map(cluster -> cluster.centroid)
            .collect(Collectors.toList());
}
```

С классом `KMeans` связан массив `points`. Это все единицы данных из набора данных. Затем единицы данных делятся между кластерами, которые хранятся в соответствующей переменной `clusters`. Когда создается экземпляр `KMeans`, он должен знать, сколько кластеров требуется создать (`k`). Каждый кластер изначально имеет случайный центроид. Все единицы данных, которые будут использоваться в алгоритме, нормализуются по z-оценке. Вычисляемое свойство `centroids` возвращает все центроиды, связанные с кластерами, создаваемыми алгоритмом (листинг 6.5).

Листинг 6.5. KMeans.java (продолжение)

```
private List<Double> dimensionSlice(int dimension) {
    return points.stream().map(x -> x.dimensions.get(dimension))
            .collect(Collectors.toList());
}
```

`dimensionSlice()` — это удобный метод, который можно представить как метод, возвращающий столбец данных. Он возвращает список, состоящий из всех значений для определенного индекса каждой единицы данных. Например, если бы единицы данных имели тип `DataPoint`, то `dimensionSlice(0)` вернул бы список значений первого измерения каждой единицы данных. Это будет полезно в следующем методе нормализации (листинг 6.6).

Листинг 6.6. KMeans.java (продолжение)

```
private void zScoreNormalize() {
    List<List<Double>> zscored = new ArrayList<>();
    for (Point point : points) {
        zscored.add(new ArrayList<Double>());
    }
    for (int dimension = 0; dimension <
    points.get(0).numDimensions; dimension++) {
        List<Double> dimensionSlice = dimensionSlice(dimension);
        Statistics stats = new Statistics(dimensionSlice);
        List<Double> zscores = stats.zscored();
        for (int index = 0; index < zscores.size(); index++) {
            zscored.get(index).add(zscores.get(index));
        }
    }
    for (int i = 0; i < points.size(); i++) {
        points.get(i).dimensions = zscored.get(i);
    }
}
```

`zScoreNormalize()` заменяет значения в кортеже измерений каждой единицы данных эквивалентом ее *z*-оценки. При этом применяется функция `zscores()`,

которую мы определили ранее для последовательностей данных типа `double`. Значения в кортеже `dimensions` заменены, но кортеж `originals` в `DataPoint` не изменился. Это полезно: если значения хранятся в двух местах, то после выполнения алгоритма пользователь может получить исходные значения измерений, какими они были до нормализации (листинг 6.7).

Листинг 6.7. KMeans.java (продолжение)

```
    private DataPoint randomPoint() {
        List<Double> randDimensions = new ArrayList<>();
        Random random = new Random();
        for (int dimension = 0; dimension < points.get(0).numDimensions;
      dimension++) {
            List<Double> values = dimensionSlice(dimension);
            Statistics stats = new Statistics(values);
            Double randValue = random.doubles(stats.min(),
                stats.max()).findFirst().getAsDouble();
            randDimensions.add(randValue);
        }
        return new DataPoint(randDimensions);
    }
```

Описанный ранее метод `randomPoint()` применяется в конструкторе для создания начальных случайных центроидов для каждого кластера. Это ограничивает случайные значения каждой единицы данных в пределах диапазона значений существующих единиц данных. Метод использует конструктор, описанный ранее в `DataPoint`, чтобы создать новую единицу данных из итерируемого объекта значений.

Теперь рассмотрим метод поиска подходящего кластера для единицы данных (листинг 6.8).

Листинг 6.8. KMeans.java (продолжение)

```
    // Найти ближайший центроид кластера для каждой единицы данных
    // и назначить единицу данных этому кластеру
    private void assignClusters() {
        for (Point point : points) {
            double lowestDistance = Double.MAX_VALUE;
            Cluster closestCluster = clusters.get(0);
            for (Cluster cluster : clusters) {
                double centroidDistance =
                    point.distance(cluster.centroid);
                if (centroidDistance < lowestDistance) {
                    lowestDistance = centroidDistance;
                    closestCluster = cluster;
                }
            }
            closestCluster.points.add(point);
        }
    }
```

На протяжении книги мы уже создали несколько функций, которые находят минимум или максимум в заданном списке. Эта функция того же рода. В данном случае мы ищем кластерный центроид с минимальным расстоянием до каждой отдельной единицы данных. Затем эта единица данных присваивается кластеру (листинг 6.9).

Листинг 6.9. KMeans.java (продолжение)

```
// найти центр каждого кластера и переместить центроид туда
private void generateCentroids() {
    for (Cluster cluster : clusters) {
        // игнорировать, если кластер пуст
        if (cluster.points.isEmpty()) {
            continue;
        }
        List<Double> means = new ArrayList<>();
        for (int i = 0; i < cluster.points.get(0).numDimensions; i++) {
            // необходимо использовать в рамках закрытия
            int dimension = i;
            Double dimensionMean = cluster.points.stream()
                    .mapToDouble(x ->
            x.dimensions.get(dimension)).average().getAsDouble();
            means.add(dimensionMean);
        }
        cluster.centroid = new DataPoint(means);
    }
}
```

После того как все единицы данных назначены кластерам, вычисляются новые центроиды. Сюда входит вычисление среднего для каждого измерения каждой единицы данных в кластере. Затем средние значения для каждого измерения объединяются, чтобы найти среднюю точку в кластере, которая становится новым центроидом. Обратите внимание на то, что здесь нельзя использовать `dimensionSlice()`, так как рассматриваемые единицы данных являются подмножеством всех единиц данных (те, что принадлежат конкретному кластеру). Как можно переписать метод `dimensionSlice()`, чтобы он был более универсальным? Далее попробуйте самостоятельно изучить эту тему.

Теперь рассмотрим метод, который фактически выполняет алгоритм (листинг 6.10).

Листинг 6.10. KMeans.java (продолжение)

```
// проверить, являются ли два списка DataPoints эквивалентными
private boolean listsEqual(List<DataPoint> first, List<DataPoint> second)
{
    if (first.size() != second.size()) {
        return false;
    }
    for (int i = 0; i < first.size(); i++) {
        for (int j = 0; j < first.get(0).numDimensions; j++) {
```

```
                if (first.get(i).dimensions.get(j).doubleValue() !=
                  second.get(i).dimensions.get(j).doubleValue()) {
                    return false;
                }
            }
        }
        return true;
    }

    public List<Cluster> run(int maxIterations) {
        for (int iteration = 0; iteration < maxIterations; iteration++) {
            for (Cluster cluster : clusters) {  // очистка всех кластеров
                cluster.points.clear();
            }
            assignClusters();
            List<DataPoint> oldCentroids = new ArrayList<>(centroids());
            generateCentroids(); // поиск новых центроидов
            if (listsEqual(oldCentroids, centroids())) {
                System.out.println("Converged after " + iteration + "
                                   iterations.");
                return clusters;
            }
        }
        return clusters;
    }
```

`run()` — наиболее чистое выражение из всего исходного алгоритма. Единственное изменение алгоритма, которое вы можете посчитать неожиданным, — это удаление всех единиц данных в начале каждой итерации. Если бы этого не произошло, метод `assignClusters()` в том виде, как он написан, в итоге поместил бы в каждый кластер дубликаты единиц данных. `ListEqual()` — это помощник, который проверяет, содержат ли два списка `DataPoints` одинаковые данные. Это полезно для проверки наличия изменений центроидов между поколениями (это означает, что движение прекратилось и алгоритм должен остановиться).

Мы можем быстро протестировать `DataPoint`, присвоив `k` значение `2` (листинг 6.11).

Листинг 6.11. KMeans.java (продолжение)

```
    public static void main(String[] args) {
        DataPoint point1 = new DataPoint(List.of(2.0, 1.0, 1.0));
        DataPoint point2 = new DataPoint(List.of(2.0, 2.0, 5.0));
        DataPoint point3 = new DataPoint(List.of(3.0, 1.5, 2.5));
        KMeans<DataPoint> kmeansTest = new KMeans<>(2, List.of(point1, point2,
            point3));
        List<KMeans<DataPoint>.Cluster> testClusters = kmeansTest.run(100);
        for (int clusterIndex = 0; clusterIndex < testClusters.size();
            clusterIndex++) {
            System.out.println("Cluster " + clusterIndex + ": " +
              testClusters.get(clusterIndex).points);
        }
    }
}
```

Из-за элемента случайности наши результаты могут различаться. Должно получиться что-то вроде следующего:

```
Converged after 1 iterations.
Cluster 0: [[2.0, 1.0, 1.0], [3.0, 1.5, 2.5]]
Cluster 1: [[2.0, 2.0, 5.0]]
```

6.3. КЛАСТЕРИЗАЦИЯ ГУБЕРНАТОРОВ ПО ВОЗРАСТУ И ДОЛГОТЕ ШТАТА

В каждом американском штате есть губернатор. В июне 2017 года возраст этих чиновников находился в диапазоне от 42 до 79 лет. Если рассмотреть Соединенные Штаты с востока на запад, перебирая все штаты по долготе, то, возможно, удастся сформировать группы штатов с близкой долготой и близким возрастом губернаторов. На рис. 6.2 представлена диаграмма распределения всех 50 губернаторов. По оси X откладывается долгота штата, а по оси Y — возраст губернатора.

Существуют ли на рис. 6.2 какие-либо очевидные кластеры? Оси здесь не нормированы. Напротив, мы рассматриваем необработанные данные. Если бы кластеры всегда были очевидными, то не было бы необходимости в алгоритмах кластеризации.

Попробуем пропустить этот набор данных через алгоритм k-средних. Для этого в первую очередь понадобится способ представления отдельной единицы данных (листинг 6.12).

Листинг 6.12. Governor.java

```
package chapter6;

import java.util.ArrayList;
import java.util.List;

public class Governor extends DataPoint {
    private double longitude;
    private double age;
    private String state;

    public Governor(double longitude, double age, String state) {
        super(List.of(longitude, age));
        this.longitude = longitude;
        this.age = age;
        this.state = state;
    }

    @Override
    public String toString() {
        return state + ": (longitude: " + longitude + ", age: " + age + ")";
    }
```

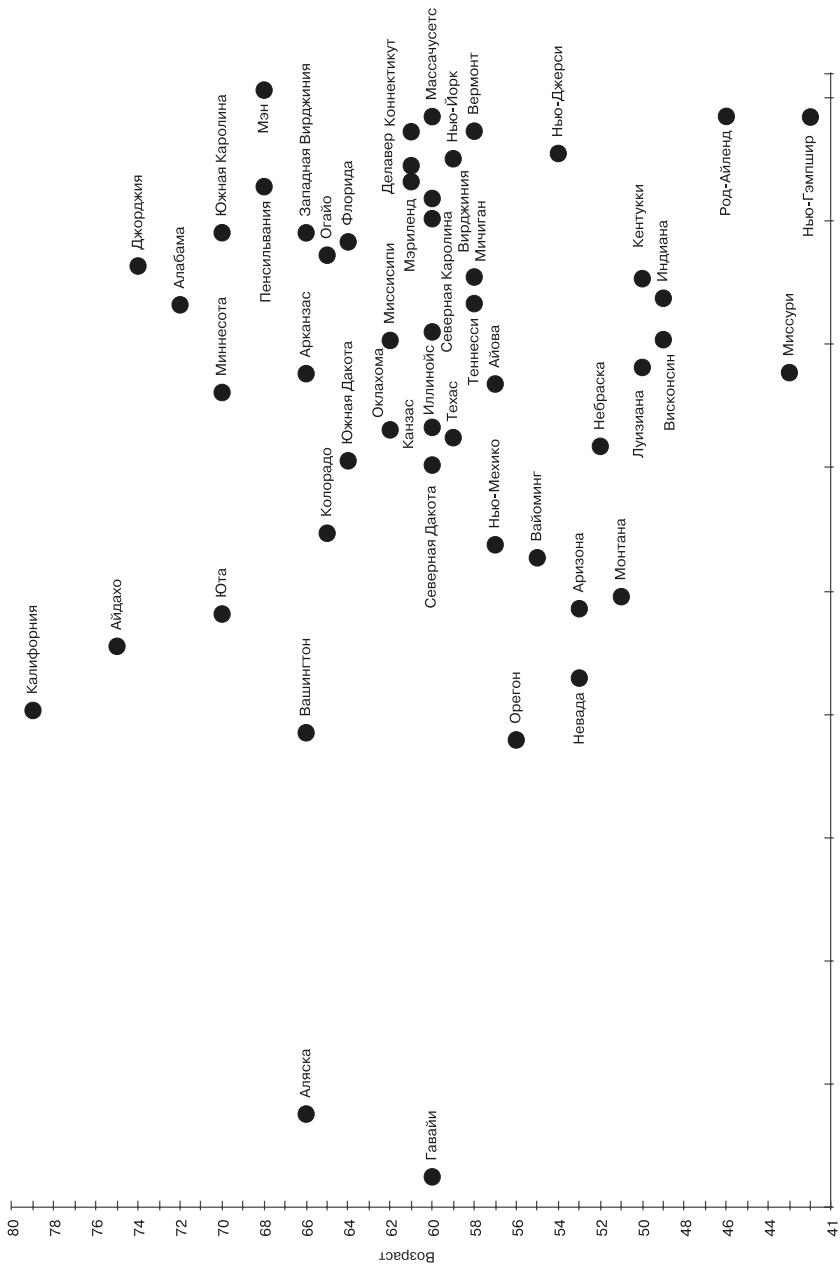


Рис. 6.2. Губернаторы штатов расположены по географической долготе возглавляемых ими штатов и возрасту (по состоянию на июнь 2017 года)

У класса `Governor` есть два именованных и сохраненных измерения: `longitude` и `age`. Кроме того, `Governor` не вносит никаких других изменений в механизм своего суперкласса `DataPoint`, кроме переопределенного `toString()` для структурной печати (листинг 6.13). Едва ли было бы разумно вводить следующие данные вручную, поэтому обратитесь в репозиторий исходного кода, прилагаемый к этой книге.

Листинг 6.13. Governor.java (продолжение)

```
    public static void main(String[] args) {
        List<Governor> governors = new ArrayList<>();
            governors.add(new Governor(-86.79113, 72, "Alabama"));
            governors.add(new Governor(-152.404419, 66, "Alaska"));
            governors.add(new Governor(-111.431221, 53, "Arizona"));
            governors.add(new Governor(-92.373123, 66, "Arkansas"));
            governors.add(new Governor(-119.681564, 79, "California"));
            governors.add(new Governor(-105.311104, 65, "Colorado"));
            governors.add(new Governor(-72.755371, 61, "Connecticut"));
            governors.add(new Governor(-75.507141, 61, "Delaware"));
            governors.add(new Governor(-81.686783, 64, "Florida"));
            governors.add(new Governor(-83.643074, 74, "Georgia"));
            governors.add(new Governor(-157.498337, 60, "Hawaii"));
            governors.add(new Governor(-114.478828, 75, "Idaho"));
            governors.add(new Governor(-88.986137, 60, "Illinois"));
            governors.add(new Governor(-86.258278, 49, "Indiana"));
            governors.add(new Governor(-93.210526, 57, "Iowa"));
            governors.add(new Governor(-96.726486, 60, "Kansas"));
            governors.add(new Governor(-84.670067, 50, "Kentucky"));
            governors.add(new Governor(-91.867805, 50, "Louisiana"));
            governors.add(new Governor(-69.381927, 68, "Maine"));
            governors.add(new Governor(-76.802101, 61, "Maryland"));
            governors.add(new Governor(-71.530106, 60, "Massachusetts"));
            governors.add(new Governor(-84.536095, 58, "Michigan"));
            governors.add(new Governor(-93.900192, 70, "Minnesota"));
            governors.add(new Governor(-89.678696, 62, "Mississippi"));
            governors.add(new Governor(-92.288368, 43, "Missouri"));
            governors.add(new Governor(-110.454353, 51, "Montana"));
            governors.add(new Governor(-98.268082, 52, "Nebraska"));
            governors.add(new Governor(-117.055374, 53, "Nevada"));
            governors.add(new Governor(-71.563896, 42, "New Hampshire"));
            governors.add(new Governor(-74.521011, 54, "New Jersey"));
            governors.add(new Governor(-106.248482, 57, "New Mexico"));
            governors.add(new Governor(-74.948051, 59, "New York"));
            governors.add(new Governor(-79.806419, 60, "North Carolina"));
            governors.add(new Governor(-99.784012, 60, "North Dakota"));
            governors.add(new Governor(-82.764915, 65, "Ohio"));
            governors.add(new Governor(-96.928917, 62, "Oklahoma"));
            governors.add(new Governor(-122.070938, 56, "Oregon"));
            governors.add(new Governor(-77.209755, 68, "Pennsylvania"));
            governors.add(new Governor(-71.51178, 46, "Rhode Island"));
            governors.add(new Governor(-80.945007, 70, "South Carolina"));
            governors.add(new Governor(-99.438828, 64, "South Dakota"));
```

```
            governors.add(new Governor(-86.692345, 58, "Tennessee"));
            governors.add(new Governor(-97.563461, 59, "Texas"));
            governors.add(new Governor(-111.862434, 70, "Utah"));
            governors.add(new Governor(-72.710686, 58, "Vermont"));
            governors.add(new Governor(-78.169968, 60, "Virginia"));
            governors.add(new Governor(-121.490494, 66, "Washington"));
            governors.add(new Governor(-80.954453, 66, "West Virginia"));
            governors.add(new Governor(-89.616508, 49, "Wisconsin"));
            governors.add(new Governor(-107.30249, 55, "Wyoming"));
```

Запустим алгоритм *k*-средних с k, равным 2 (листинг 6.14).

Листинг 6.14. Governor.java (продолжение)

```
        KMeans<Governor> kmeans = new KMeans<>(2, governors);
        List<KMeans<Governor>.Cluster> govClusters = kmeans.run(100);
        for (int clusterIndex = 0; clusterIndex < govClusters.size();
        clusterIndex++) {
            System.out.printf("Cluster %d: %s%n", clusterIndex,
              govClusters.get(clusterIndex).points);
        }
    }
}
```

Поскольку выполнение алгоритма начинается со случайно выбранных центроидов, то каждый запуск `KMeans` потенциально может возвращать разные кластеры. Чтобы увидеть, действительно ли это правильно выбранные кластеры, требуется, чтобы результаты проанализировал человек. Следующий результат получен после запуска, который дал действительно интересный кластер:

```
Converged after 3 iterations.
Cluster 0: [Alabama: (longitude: -86.79113, age: 72.0), Arizona: (longitude:
-111.431221, age: 53.0), Arkansas: (longitude: -92.373123, age: 66.0), Colorado:
(longitude: -105.311104, age: 65.0), Connecticut: (longitude: -72.755371, age:
61.0), Delaware: (longitude: -75.507141, age: 61.0), Florida: (longitude:
-81.686783, age: 64.0), Georgia: (longitude: -83.643074, age: 74.0), Illinois:
(longitude: -88.986137, age: 60.0), Indiana: (longitude: -86.258278, age: 49.0),
Iowa: (longitude: -93.210526, age: 57.0), Kansas: (longitude: -96.726486, age:
60.0), Kentucky: (longitude: -84.670067, age: 50.0), Louisiana: (longitude:
-91.867805, age: 50.0), Maine: (longitude: -69.381927, age: 68.0), Maryland:
(longitude: -76.802101, age: 61.0), Massachusetts: (longitude: -71.530106, age:
60.0), Michigan: (longitude: -84.536095, age: 58.0), Minnesota: (longitude:
-93.900192, age: 70.0), Mississippi: (longitude: -89.678696, age: 62.0), Missouri:
(longitude: -92.288368, age: 43.0), Montana: (longitude: -110.454353, age: 51.0),
Nebraska: (longitude: -98.268082, age: 52.0), Nevada: (longitude: -117.055374,
age: 53.0), New Hampshire: (longitude: -71.563896, age: 42.0), New Jersey:
(longitude: -74.521011, age: 54.0), New Mexico: (longitude: -106.248482, age:
57.0), New York: (longitude: -74.948051, age: 59.0), North Carolina: (longitude:
-79.806419, age: 60.0), North Dakota: (longitude: -99.784012, age: 60.0), Ohio:
(longitude: -82.764915, age: 65.0), Oklahoma: (longitude: -96.928917, age: 62.0),
Pennsylvania: (longitude: -77.209755, age: 68.0), Rhode Island: (longitude:
-71.51178, age: 46.0), South Carolina: (longitude: -80.945007, age: 70.0), South
```

```
Dakota: (longitude: -99.438828, age: 64.0), Tennessee: (longitude: -86.692345,
age: 58.0), Texas: (longitude: -97.563461, age: 59.0), Vermont: (longitude:
-72.710686, age: 58.0), Virginia: (longitude: -78.169968, age: 60.0), West
Virginia: (longitude: -80.954453, age: 66.0), Wisconsin: (longitude: -89.616508,
age: 49.0), Wyoming: (longitude: -107.30249, age: 55.0)]
Cluster 1: [Alaska: (longitude: -152.404419, age: 66.0), California: (longitude:
-119.681564, age: 79.0), Hawaii: (longitude: -157.498337, age: 60.0), Idaho:
(longitude: -114.478828, age: 75.0), Oregon: (longitude: -122.070938, age: 56.0),
Utah: (longitude: -111.862434, age: 70.0), Washington: (longitude: -121.490494,
age: 66.0)]
```

Кластер 1 представляет штаты Крайнего Запада, все они географически расположены рядом друг с другом (если считать Аляску и Гавайи штатами Тихоокеанского побережья). Во всех них относительно старые губернаторы, следовательно, эти штаты образуют интересную группу. Население Тихоокеанского побережья предпочитает губернаторов постарше? Мы не можем сделать на основании этих кластеров какие-либо определенные выводы за пределами данной корреляции. Результат показан на рис. 6.3. Квадратики соответствуют кластеру 1, а кружки — кластеру 0.

СОВЕТ

Необходимо еще и еще раз подчеркнуть, что результаты, полученные методом *k*-средних со случайной инициализацией центроидов, всегда будут различаться. Обязательно запустите алгоритм *k*-средних несколько раз независимо от набора данных.

6.4. КЛАСТЕРИЗАЦИЯ АЛЬБОМОВ МАЙКЛА ДЖЕКСОНА ПО ДЛИТЕЛЬНОСТИ

Майкл Джексон выпустил десять сольных студийных альбомов. В следующем примере мы сгруппируем их, рассмотрев два измерения: длительность (в минутах) и количество дорожек. Этот пример хорошо контрастирует с предыдущим — о губернаторах, поскольку в его исходном наборе данных кластеры легко увидеть даже без использования метода *k*-средних. Подобный пример может быть хорошим способом отладки реализации алгоритма кластеризации.

ПРИМЕЧАНИЕ

Оба примера, представленные в этой главе, задействуют двумерные единицы данных, но метод *k*-средних позволяет работать с единицами данных, имеющими любое количество измерений.

Этот пример представлен целиком в виде одного листинга с кодом. Если вы посмотрите на данные об альбомах в листинге 6.15, прежде чем запустить пример, то станет ясно, что самые длинные альбомы Майкл Джексон записал ближе к концу

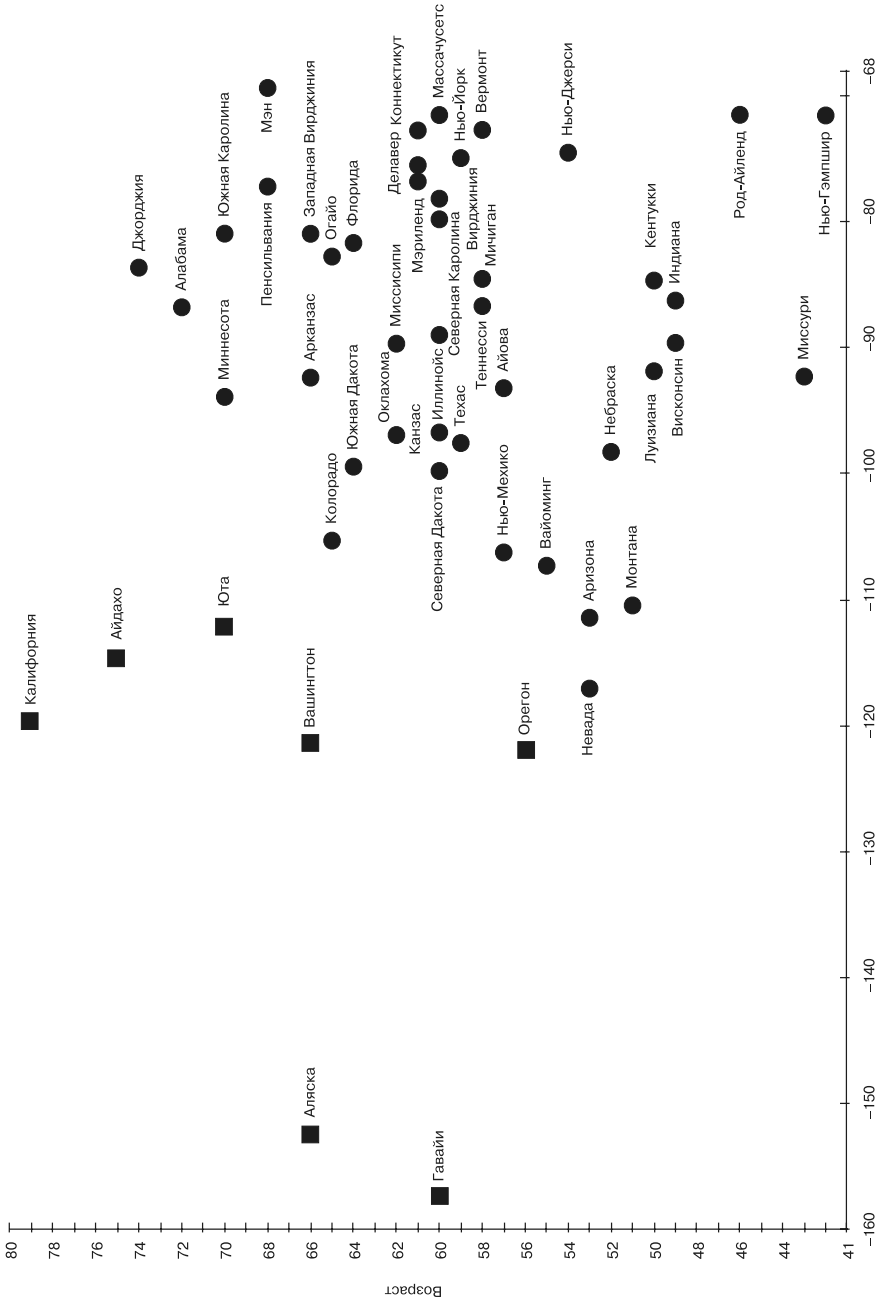


Рис. 6.3. Единицы данных кластера 0 обозначены кружками, а единицы данных кластера 1 — квадратиками

карьеры. Таким образом, вероятно, следует выделить два кластера — более ранние и более поздние альбомы. Альбом *HIStory: Past, Present and Future, Book I* является выбросом и может логически оказаться в отдельном кластере с единственной единицей данных. *Выбросом* называется единица данных, которая выходит за пределы нормальных границ набора данных.

Листинг 6.15. Album.java

```
package chapter6;

import java.util.ArrayList;
import java.util.List;

public class Album extends DataPoint {

    private String name;
    private int year;

    public Album(String name, int year, double length, double tracks) {
        super(List.of(length, tracks));
        this.name = name;
        this.year = year;
    }

    @Override
    public String toString() {
        return "(" + name + ", " + year + ")";
    }

    public static void main(String[] args) {
        List<Album> albums = new ArrayList<>();
        albums.add(new Album("Got to Be There", 1972, 35.45, 10));
        albums.add(new Album("Ben", 1972, 31.31, 10));
        albums.add(new Album("Music & Me", 1973, 32.09, 10));
        albums.add(new Album("Forever, Michael", 1975, 33.36, 10));
        albums.add(new Album("Off the Wall", 1979, 42.28, 10));
        albums.add(new Album("Thriller", 1982, 42.19, 9));
        albums.add(new Album("Bad", 1987, 48.16, 10));
        albums.add(new Album("Dangerous", 1991, 77.03, 14));
        albums.add(new Album("HIStory: Past, Present and Future, Book I", 1995,
            148.58, 30));
        albums.add(new Album("Invincible", 2001, 77.05, 16));
        KMeans<Album> kmeans = new KMeans<>(2, albums);
        List<KMeans<Album>.Cluster> clusters = kmeans.run(100);
        for (int clusterIndex = 0; clusterIndex < clusters.size();
            clusterIndex++) {
            System.out.printf("Cluster %d Avg Length %f Avg Tracks %f: %s%n",
            clusterIndex, clusters.get(clusterIndex).centroid.dimensions.get(0),
            clusters.get(clusterIndex).centroid.dimensions.get(1),
            clusters.get(clusterIndex).points);
        }
    }
}
```

Обратите внимание: атрибуты `name` и `year` записываются только для обозначения альбома и не используются при кластеризации. Вот пример вывода результатов:

```
Converged after 1 iterations.
Cluster 0 Avg Length -0.5458820039179509 Avg Tracks -0.5009878988684237: [(Got
to Be There, 1972), (Ben, 1972), (Music & Me, 1973), (Forever, Michael, 1975),
(Off the Wall, 1979), (Thriller, 1982), (Bad, 1987)] Cluster 1 Avg Length
1.2737246758085525 Avg Tracks 1.168971764026322: [(Dangerous, 1991), (HIStory:
Past, Present and Future, Book I, 1995), (Invincible, 2001)]
```

Представляют интерес вычисленные средние значения кластеров. Обратите внимание на то, что средние значения являются z-оценками. Три альбома из кластера 1 — последние в творчестве Майкла Джексона — оказались примерно на одно стандартное отклонение длиннее, чем в среднем остальные семь его сольных альбомов.

6.5. ПРОБЛЕМЫ И РАСШИРЕНИЯ КЛАСТЕРИЗАЦИИ МЕТОДОМ *k*-СРЕДНИХ

Когда кластеризация методом *k*-средних осуществляется с использованием случайных начальных точек, она может пропустить все полезные точки разделения данных. Это часто требует множества проб и ошибок при выполнении операции. Также определение правильного значения *k* (количества кластеров) сложно и чревато ошибками, если у оператора нет четкого представления о том, сколько должно быть групп данных.

Существуют более сложные версии алгоритма *k*-средних, в которых можно попытаться сделать обоснованные предположения или выполнить несколько автоматических проб (и допустить ошибки) при определении проблемных переменных. Один из распространенных вариантов — это метод *k*-средних++, который пытается решить проблему инициализации, выбирая центроиды не совершенно случайно, а на основе распределения вероятностей расстояния до каждой единицы данных. Еще более удачный вариант для многих приложений — выбор хороших начальных областей для каждого центроида на основе заранее известной информации о данных, иными словами, версия метода *k*-средних, в которой начальные центроиды выбирает пользователь алгоритма.

Время выполнения кластеризации методом *k*-средних пропорционально количеству единиц данных, количеству кластеров и количеству измерений для единиц данных. Если имеется множество единиц данных с большим количеством измерений, то этот алгоритм в своей базовой форме может оказаться непригодным. Существуют расширения, которые пытаются не проводить столько вычислений

для каждой единицы данных и каждого центроида, оценивая, действительно ли эта единица данных может переместиться в другой кластер, прежде чем выполнять вычисление. Другой вариант для наборов с многочисленными единицами данных или с большим количеством измерений — просто сделать выборку единиц данных методом *k*-средних. Это позволит создать приблизительный набор кластеров, которые затем сможет уточнить полный алгоритм *k*-средних.

Выбросы в наборе данных могут привести к странным результатам работы метода *k*-средних. Если первоначальный центроид окажется рядом с выбросом, то он может сформировать кластер из одной единицы данных (как с альбомом *HIStory* в примере с Майклом Джексоном). Метод *k*-средних может работать лучше, если вначале удалить выбросы.

Наконец, среднее значение не всегда считается хорошим показателем для центра. В методе *k*-медиан серединой кластера является медиана каждого измерения, а в методе *k*-медоидов — фактическая единица набора данных. Для выбора каждого из этих методов центрирования существуют статистические причины, выходящие за рамки этой книги, но здравый смысл подсказывает, что для сложной задачи, возможно, стоит попробовать каждый из них и выбрать подходящие результаты. Реализации всех этих методов не так уж и различаются.

6.6. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

Кластеризация — это то, чем часто приходится заниматься исследователям данных и аналитикам статистики. Она широко используется для интерпретации данных в различных областях. В частности, кластеризация методом *k*-средних является полезной технологией в тех случаях, когда о структуре набора данных известно немногое.

Кластеризация — важная методика для анализа данных. Представьте себе полицейское управление, сотрудники которого хотят знать, как расставить полицейские патрули. Или руководителя франшизы предприятия быстрого питания, которому нужно выяснить, где находятся лучшие клиенты, чтобы рассылать рекламные предложения. Или оператора на борту судна, который стремится свести к минимуму количество аварий, анализируя, когда они происходят и кто является их причиной. Теперь подумайте, как все эти люди могли бы решить свои проблемы с помощью кластеризации.

Кластеризация помогает при распознавании образов. Алгоритм кластеризации позволяет обнаруживать шаблоны, которые пропускает человеческий глаз. Например, кластеризация иногда используется в биологии для выявления групп несоответствующих клеток.

При распознавании изображений кластеризация помогает идентифицировать неочевидные функции. Отдельные пикселы можно рассматривать как единицы данных, их отношение друг к другу определяется расстоянием и разницей цветов.

В политологии кластеризация иногда применяется для поиска избирателей. Может ли политическая партия выявить избирателей, лишенных избирательных прав и сконцентрированных в одном округе, чтобы сосредоточить там свои затраты на избирательную кампанию? Какие проблемы могут вызывать опасения у подобных избирателей?

6.7. УПРАЖНЕНИЯ

1. Создайте функцию, которая импортировала бы данные из CSV-файла в объекты `DataPoints`.
2. Создайте функцию, задействуя среду графического интерфейса пользователя (например, AWT, Swing или JavaFX) или графическую библиотеку, которая создает диаграмму разброса с цветовой кодировкой результатов любого запуска `KMeans` на двумерном наборе данных.
3. Создайте новый инициализатор для `KMeans`, который будет принимать начальные позиции центроидов извне, а не назначать их случайным образом.
4. Исследуйте и реализуйте алгоритм *k*-средних++.

7 Простейшие нейронные сети

Достижения в области искусственного интеллекта, о которых мы слышим в наше время, обычно имеют отношение к дисциплине, известной как *машинное обучение* (компьютеры изучают некую новую информацию, не получая на то явной команды). Чаще всего эти достижения появлялись благодаря технологии машинного обучения, известной как *нейронные сети*. Нейронные сети были изобретены пару десятков лет назад, но сейчас переживают что-то вроде возрождения, поскольку усовершенствованное вычислительное оборудование и новые программные технологии, построенные по принципу исследований, позволяют создать новую парадигму, известную как *глубокое обучение*.

Глубокое обучение нашло широкое применение. Оно оказалось полезным в самых разных областях, от алгоритмов хедж-фондов до биоинформатики. Два самых широко известных потребителям способа применения технологии глубокого обучения — это распознавание изображений и распознавание речи. Когда вы спрашиваете своего цифрового помощника о прогнозе погоды или обнаруживаете, что программа обработки фотографий распознала ваше лицо, скорее всего, выполняется некое глубокое обучение.

В методах глубокого обучения используются те же строительные блоки, что и в простейших нейронных сетях. В этой главе мы рассмотрим такие блоки, построив простую нейронную сеть. Это не будет современным решением, но станет основой для понимания глубокого обучения, которое основано на более сложных нейронных сетях, чем построенные нами. На практике при машинном обучении в большинстве случаев нейронные сети не строятся с нуля. Вместо этого применяются популярные высокооптимизированные готовые платформы, которые и выполняют всю тяжелую работу. Эта глава не поможет вам узнать,

как задействовать какую-либо конкретную среду, и сеть, которую мы создадим, будет бесполезной для реального приложения, но она поможет вам понять, как эти платформы работают на низком уровне.

7.1. В ОСНОВЕ — БИОЛОГИЯ?

Человеческий мозг — самое невероятное вычислительное устройство из существующих. Он не в состоянии обрабатывать числа так же быстро, как микропроцессор, но с его способностью адаптироваться к незнакомым ситуациям, изучать новые навыки и проявлять творческий подход не сравнится ни одна из известных машин. С момента появления первых компьютеров ученые были заинтересованы в моделировании механизмов мозга. Нервные клетки мозга называются *нейронами*. Нейроны в мозге связаны между собой соединениями, которые называются *синапсами*. Для питания этих сетей нейронов, также известных как *нейронные сети*, по синапсам подается электричество.

ПРИМЕЧАНИЕ

Приведенное здесь описание биологических нейронов — это грубое упрощение ради аналогии. На самом деле у биологических нейронов есть такие части, как аксоны, дендриты и ядра, о которых вы, возможно, знаете из курса биологии средней школы. А синапсы в действительности являются промежутками между нейронами, в которых выделяются нейротрансмиттеры, обеспечивающие прохождение этих электрических сигналов.

Несмотря на то что ученые выделили части и определили функции нейронов, подробности того, каким образом биологические нейронные сети образуют сложные модели мышления, до сих пор не вполне понятны. Как они обрабатывают информацию? Как формируют оригинальные мысли? Бóльшая часть наших знаний о том, как работает мозг, основана на рассмотрении его на макроуровне. Сканирование мозга методом функциональной магнитно-резонансной томографии (МРТ) показывает, где течет кровь, когда человек выполняет определенную деятельность или обдумывает конкретную мысль (рис. 7.1). Эта и другие макротехнологии могут помочь понять, каким образом связаны различные части мозга, но они не объясняют тайны того, как отдельные нейроны способствуют образованию новых мыслей.

Группы ученых по всему земному шару стремятся раскрыть секреты мозга, но вот что следует учесть: в человеческом мозге приблизительно 100 000 000 000 нейронов, и каждый из них может быть связан с десятками тысяч других нейронов. Даже на компьютере с миллиардами логических элементов и терабайтами памяти современные технологии не позволяют построить модель одного человеческого

мозга. Очевидно, в обозримом будущем человек по-прежнему будет оставаться самым сложным универсальным объектом изучения.

Общественное достояние США.
Национальный институт психического здоровья

Рис. 7.1. Исследователь изучает изображения головного мозга, полученные посредством МРТ. МРТ-изображения мало что говорят нам о том, как функционируют отдельные нейроны или как организованы нейронные сети (источник — Национальный институт психического здоровья)

ПРИМЕЧАНИЕ

Универсальная обучающая машина со способностями, сопоставимыми с человеческими, является целью так называемого сильного искусственного интеллекта (ИИ), также известного как общий искусственный интеллект. В данный исторический момент это все еще относится к области научной фантастики. Слабый ИИ — это тип ИИ, который нам встречается ежедневно: компьютеры, интеллектуально решающие конкретные задачи, на которые они были предварительно настроены.

Если биологические нейронные сети все еще не полностью изучены, то каким образом их моделирование стало эффективным вычислительным методом? Действительно, цифровые нейронные сети, известные как *искусственные нейронные сети*, построены под впечатлением от биологических нейронных сетей, но на этом их сходство заканчивается. Современные искусственные нейронные сети не претендуют на то, чтобы работать подобно своим биологическим аналогам. На самом деле это было бы невозможно прежде всего потому, что мы не до конца понимаем принципы работы биологических нейронных сетей.

7.2. ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

В этом разделе мы изучим то, что, пожалуй, является самым распространенным типом искусственной нейронной сети, — *сеть прямой связи с обратным распространением*. Именно такую сеть мы позже разработаем. *Прямая связь* означает, что сигнал обычно движется по сети в одном направлении. *Обратное распространение* подразумевает, что мы будем обнаруживать ошибки в конце прохождения каждого сигнала по сети и пытаться распространять исправления этих ошибок в обратном направлении, особенно значительно воздействуя на нейроны, в наибольшей степени ответственные за эти ошибки. Существует много других типов искусственных нейронных сетей, и возможно, эта глава пробудит в вас интерес к дальнейшим исследованиям.

7.2.1. Нейроны

Наименьшая единица искусственной нейронной сети — это нейрон. Он хранит вектор весов, которые представляют собой обычные числа с плавающей точкой. Нейрону передается вектор входных данных, которые также являются обычными числами с плавающей точкой. Нейрон объединяет входные данные с их весами посредством скалярного произведения. Затем запускает *функцию активации* для этого произведения и выдает результат на выходе. Можно считать, что это действие аналогично поведению настоящих нейронов.

Функция активации — это преобразователь выходного сигнала нейрона. Она почти всегда нелинейна, что позволяет нейронным сетям представлять решения нелинейных задач. Если бы не было функций активации, то вся нейронная сеть была бы просто линейным преобразованием. На рис. 7.2 показана работа одного нейрона.

ПРИМЕЧАНИЕ

В этом разделе используются несколько математических терминов, которые вам, возможно, не встречались со времен изучения курсов вычислительной математики или линейной алгебры. Объяснение того, что такое векторы или скалярное произведение, выходит за рамки этой главы, но вы, вероятно, все равно получите представление о том, что делает нейронная сеть, просто читая текст, даже если не вполне понимаете математическую часть. Позже в этой главе будут применены некоторые численные методы, такие как вычисление обычных и частных производных, но даже если вы не понимаете математику, то все равно сможете проследить код. В сущности, здесь не объясняется, как можно получить формулы, по которым выполняются вычисления. Вместо этого мы сосредоточимся на самих вычислениях.

Рис. 7.2. В нейроне его веса объединяются с входными сигналами для получения результирующего сигнала, который модифицируется функцией активации

7.2.2. Слои

В типичной искусственной нейронной сети с прямой связью нейроны образуют слои. Каждый слой состоит из определенного количества нейронов, выстроенных в строку или в столбец в зависимости от диаграммы (варианты эквивалентны). В сети с прямой связью, которую мы будем создавать, сигналы всегда передаются в одном направлении от одного слоя к другому. Нейроны в каждом слое посылают выходной сигнал, который является входным для нейронов следующего слоя. Каждый нейрон в каждом слое связан с каждым нейроном в следующем слое.

Первый слой называется *входным*, он получает сигналы от некоторого внешнего объекта. Последний слой известен как *выходной*, и его выходные сигналы обычно должен интерпретировать внешний субъект, чтобы получить осмысленный результат. Слои, расположенные между входным и выходным слоями, называются *скрытыми*. В простых нейронных сетях, таких как та, которую мы будем строить в этой главе, есть только один скрытый слой, но в сетях глубокого обучения слоев много. На рис. 7.3 показаны слои, образующие простую сеть. Обратите внимание

на то, как выходные сигналы одного слоя используются в качестве входных для каждого нейрона следующего слоя.

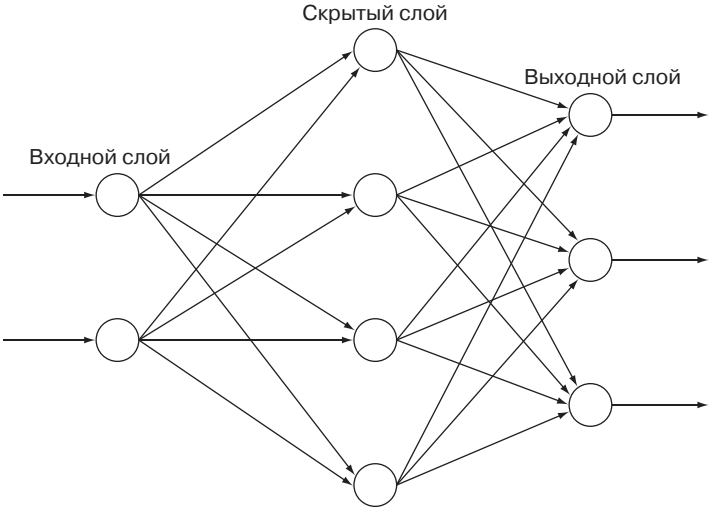


Рис. 7.3. Простая нейронная сеть с одним входным слоем, состоящим из двух нейронов, одним скрытым слоем из четырех нейронов и одним выходным слоем из трех нейронов. Количество нейронов в каждом слое произвольно

Эти слои просто манипулируют числами с плавающей точкой. Входные сигналы для входного слоя и выходные сигналы выходного слоя — это числа с плавающей запятой.

Очевидно, что эти числа должны представлять что-то значимое. Представим сеть, разработанную для классификации небольших черно-белых изображений животных. Возможно, ее входной слой имеет 100 нейронов, представляющих интенсивность оттенков серого для каждого пиксела в изображении животного размером 10 × 10 пикселов, а выходной слой имеет пять нейронов, представляющих вероятность того, что это изображение млекопитающего, рептилии, амфибии, рыбы или птицы. Окончательная классификация может быть определена выходным нейроном с самым высоким выходным сигналом с плавающей точкой. Если бы выходные числа были равны 0,24, 0,65, 0,70, 0,12 и 0,21, то изображение можно было бы классифицировать как амфибию.

7.2.3. Обратное распространение

Последняя и самая сложная часть головоломки — это обратное распространение. Обратное распространение позволяет обнаружить ошибку в выходных данных

нейронной сети и задействовать ее для изменения весов нейронов. Сильнее всего изменяются нейроны, больше других ответственные за ошибку. Но откуда берется ошибка? Как ее распознать? Ошибки возникают на этапе использования нейронной сети, называемом *обучением*.

СОВЕТ

В этом разделе обычным языком описаны этапы, которые можно представить несколькими математическими формулами. На рисунках приводятся псевдоформулы (без принятых в математике обозначений). Благодаря такому подходу формулы становятся удобочитаемыми для тех, кто не связан с математикой или не применяет математические обозначения. Если вас интересует более формальная запись и вывод формул, ознакомьтесь с главой 18 книги «Искусственный интеллект» Рассела и Норвига.

Прежде чем с нейронной сетью можно будет работать, ее, как правило, необходимо обучить. Но мы должны знать правильные выходные данные для определенных входных данных, чтобы можно было, используя разницу между ожидаемыми и фактическими выходными данными, находить ошибки и изменять веса. Другими словами, нейронные сети ничего не знают, пока им не сообщат правильные ответы для определенного набора входных данных, чтобы потом они могли подготовиться к другим входным данным. Обратное распространение происходит только во время обучения.

ПРИМЕЧАНИЕ

Поскольку большинство нейронных сетей нуждается в обучении, их относят к типу контролируемого машинного обучения. Напомню: как было показано в главе 6, алгоритм k-средних и другие кластерные алгоритмы считаются формой неконтролируемого машинного обучения, поскольку после их запуска внешнее вмешательство не требуется. Существуют и другие типы нейронных сетей, отличные от описанных в этой главе, которые не требуют предварительной подготовки и считаются формой неконтролируемого обучения.

Первый шаг при обратном распространении — вычисление ошибки между выходным сигналом нейронной сети для некоторого входного сигнала и ожидаемым выходным сигналом. Эта ошибка распространяется на все нейроны в выходном слое. (Для каждого нейрона есть ожидаемый и фактический выходной сигнал.) Затем к тому, что было бы выходным сигналом нейрона до того, как была задействована его функция активации, применяется производная функции активации выходного нейрона. (Мы кэшируем результат его функции перед активацией.) Этот результат умножают на ошибку нейрона, чтобы найти его *дельту*. Формула вычисления дельты использует частную производную (нахождение этой производной выходит за рамки данной книги). Главное, что мы получаем в результате, — то, за какое количество ошибок отвечает каждый выходной нейрон. Диаграмма вычисления показана на рис. 7.4.

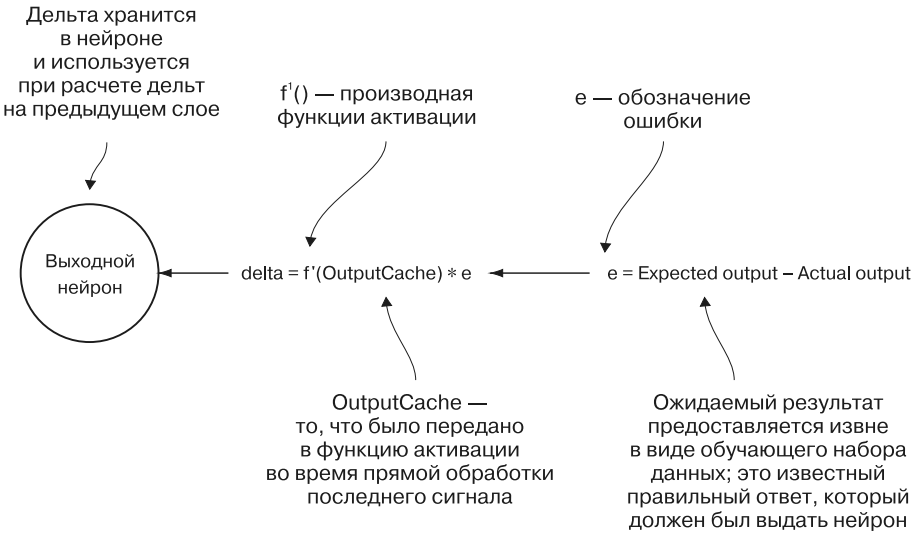


Рис. 7.4. Механизм, с помощью которого вычисляется дельта выходного нейрона на этапе обучения посредством обратного распространения

Затем необходимо вычислить дельты для всех нейронов скрытого слоя (слоев) данной сети. Нужно определить, насколько каждый нейрон ответствен за неправильный выходной сигнал в выходном слое. Дельты выходного слоя используются для вычисления дельт в предыдущем скрытом слое. Для каждого предыдущего слоя дельты вычисляются определением скалярного произведения весов следующего слоя по отношению к конкретному рассматриваемому нейрону и уже вычисленных дельт в следующем слое. Чтобы получить дельту нейрона, это значение умножается на производную от функции активации, применяемой к последнему выходному сигналу нейрона, кэшированному перед задействованием функции активации. Эта формула также получена с помощью частной производной, подробнее о которой вы можете прочитать в более специализированных математических текстах.

На рис. 7.5 показан фактический расчет дельт для нейронов в скрытых слоях. В сети с несколькими скрытыми слоями нейроны O1, O2 и O3 могут быть нейронами следующего скрытого слоя, а не нейронами выходного слоя.

И последнее, но самое главное: все веса для каждого нейрона в сети должны быть обновлены путем умножения последнего входного значения каждого отдельного веса на дельту нейрона и нечто, называемое *скоростью обучения*, и прибавления этого значения к существующему весу. Этот метод изменения веса нейрона называется *градиентным спуском*. Он похож на спуск с холма, где холм — это

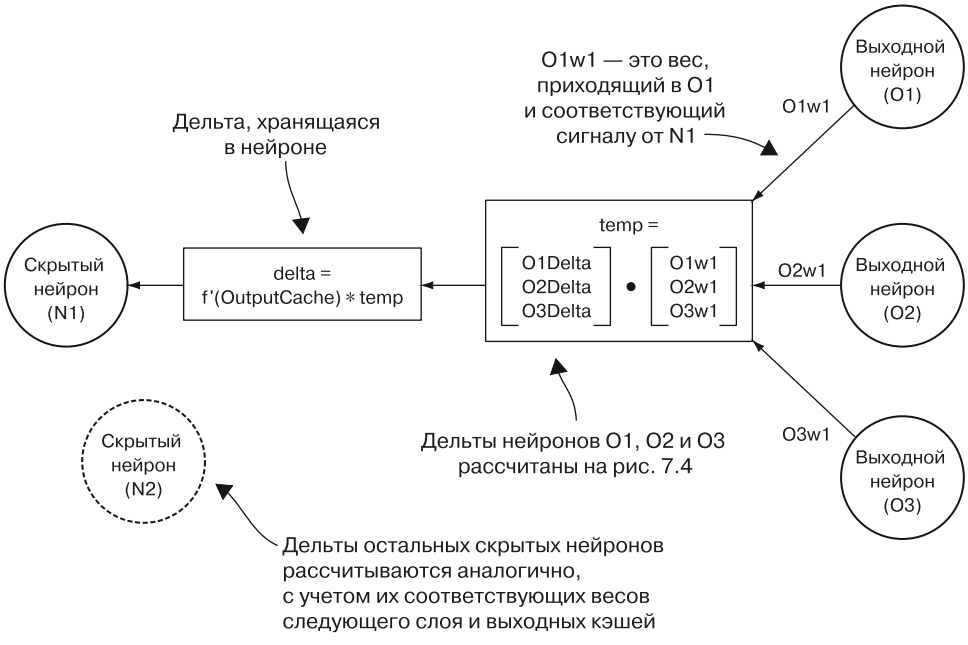


Рис. 7.5. Вычисление дельты для нейрона в скрытом слое

график функции ошибки нейрона, к точке минимальной ошибки. Дельта — это направление, в котором мы хотим спускаться, а скорость обучения определяет то, как быстро мы будем это делать. Трудно определить хорошую скорость обучения для неизвестной задачи без проб и ошибок. На рис. 7.6 показано, как изменяются веса в скрытом и выходном слоях.

После изменения весов нейронная сеть готова к повторному обучению с другим набором входных и ожидаемых выходных данных. Процесс повторяется до тех пор, пока пользователь нейронной сети не сочтет, что сеть хорошо обучена. Это можно определить, проверив сеть по входным данным с известными правильными выходными данными.

Обратное распространение — сложный процесс. Не беспокойтесь, если вы еще не поняли все его детали. Объяснения, представленного в этом разделе, может оказаться недостаточно. В идеале реализация обратного распространения выведет ваше понимание на новый уровень. При реализации нашей нейронной сети и обратного распространения помните главное: обратное распространение — это способ корректировки каждого отдельного веса в сети в соответствии с тем, насколько верно он определяет неправильные выходные данные.

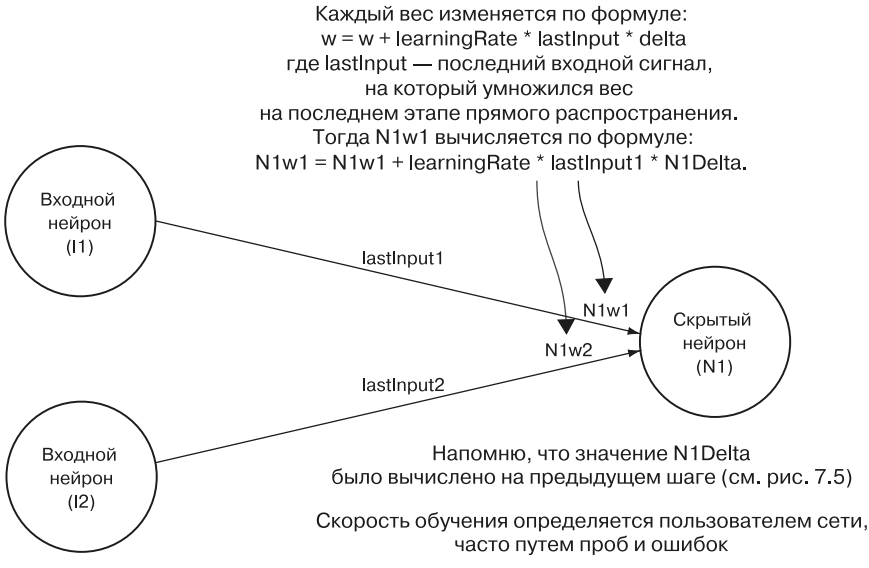


Рис. 7.6. Веса каждого скрытого слоя и нейрона выходного слоя обновляются с помощью дельт, рассчитанных на предыдущих шагах, предыдущих весов, предыдущих входных данных и заданной пользователем скорости обучения

7.2.4. Ситуация в целом

В этом разделе мы рассмотрели множество вопросов. Даже если вам все еще непонятны некоторые детали, важно помнить об основных свойствах сети прямой связи с обратным распространением.

1. Сигналы (числа с плавающей точкой) проходят через нейроны, расположенные слоями, в одном направлении. Каждый нейрон каждого слоя связан с каждым нейроном следующего слоя.
2. Каждый нейрон (кроме нейронов входного слоя) обрабатывает получаемые сигналы, комбинируя их с весами (которые также являются числами с плавающей точкой) и применяя к ним функцию активации.
3. Во время процесса, называемого обучением, результаты сети сравниваются с ожидаемыми результатами, чтобы вычислить ошибки.
4. Ошибки распространяются по сети в обратном направлении (туда, откуда они пришли) для изменения весов, чтобы в результате они с большей вероятностью приводили к выработке правильных выходных данных.

Кроме описанных, существует множество других методов обучения нейронных сетей. Есть также много иных способов передачи сигналов внутри нейронных

сетей. Описанный здесь метод, который мы и будем реализовывать, является лишь очень распространенной формой, которая служит достойным введением в предмет. В приложении Б приводятся дополнительные источники для получения расширенной информации по нейронным сетям (включая другие типы сетей) и по математике.

7.3. ПРЕДВАРИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

В нейронных сетях применяются математические механизмы, которые требуют множества операций с плавающей точкой. Прежде чем мы разработаем фактические структуры простой нейронной сети, нам понадобятся некоторые математические примитивы. Эти простые примитивы широко используются в приводимом далее коде, поэтому если вы сможете найти способы ускорения их работы, это существенно повысит производительность нейронной сети.

ПРЕДУПРЕЖДЕНИЕ

Сложность кода в данной главе, вероятно, выше, чем в любой другой главе книги. Будет много наслоений, а реальные результаты мы увидим лишь в самом конце. Есть множество ресурсов, посвященных нейронным сетям, которые помогут вам создать такую сеть, написав всего несколько строк кода, но цель этого примера — изучение механизма и того, как организовать взаимодействие различных компонентов в удобочитаемой и расширяемой форме. Пусть даже код получится довольно длинным и многословным.

7.3.1. Скалярное произведение

Как мы помним, скалярное произведение требуется и на этапе прямой связи, и на этапе обратного распространения. Будем хранить вспомогательные функции в файле `Util` (листинг 7.1). Как и весь код в этой главе, написанный лишь для демонстрации, это очень наивная реализация, не учитывающая вопросов производительности. В производственной библиотеке будут использоваться векторные инструкции, как описано в разделе 7.6.

Листинг 7.1. Util.java

```
package chapter7;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
```

```
import java.util.List;
import java.util.stream.Collectors;

public final class Util {

    public static double dotProduct(double[] xs, double[] ys) {
        double sum = 0.0;
        for (int i = 0; i < xs.length; i++) {
            sum += xs[i] * ys[i];
        }
        return sum;
    }
```

7.3.2. Функция активации

Напомню, что функция активации преобразует выходные данные нейрона перед тем, как сигнал перейдет на следующий слой (см. рис. 7.2). Функция активации имеет две цели: она позволяет нейронной сети представлять решения, которые не являются всего лишь линейными преобразованиями (если сама функция активации — это что-то большее, нежели линейное преобразование), и может делать так, чтобы выходные данные каждого нейрона не выходили за пределы заданного диапазона. Функция активации должна иметь вычислимую производную, чтобы ее можно было использовать для обратного распространения.

Популярным множеством функций активации являются *сигмоидные* функции. Одна из самых широко распространенных сигмоидных функций, часто называемая просто сигмоидной функцией, показана на рис. 7.7 (обозначена $S(x)$). Там же приводится ее уравнение и производная $S'(x)$. Результатом сигмоидной функции всегда будет значение в диапазоне 0...1. Как мы вскоре узнаем, то, что это число всегда находится в пределах от 0 до 1, полезно для сети. В листинге 7.2 формулы, показанные на этом рисунке, реализованы в виде кода.

Существуют и другие функции активации, но мы будем использовать сигмоидную функцию. Вот прямое преобразование формул, показанных на рис. 7.7, в код.

Листинг 7.2. Util.java (продолжение)

```
    // классическая сигмоидная функция активации
    public static double sigmoid(double x) {
        return 1.0 / (1.0 + Math.exp(-x));
    }

    public static double derivativeSigmoid(double x) {
        double sig = sigmoid(x);
        return sig * (1.0 – sig);
    }
```

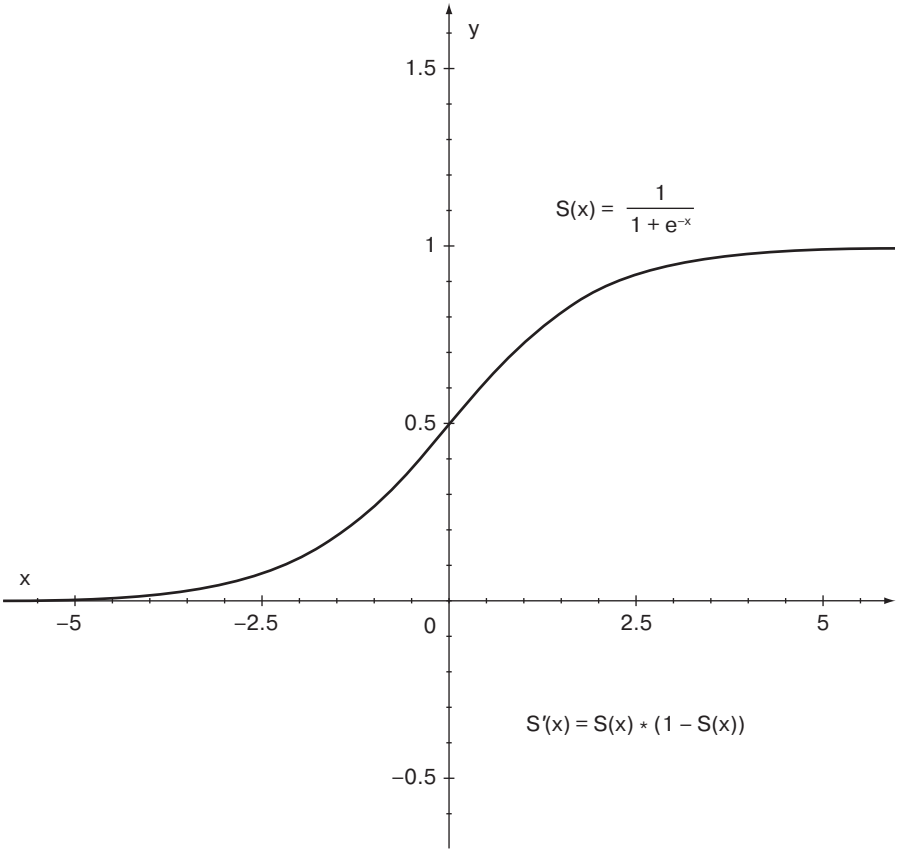


Рис. 7.7. Сигмоидная функция активации *S*(*x*) всегда возвращает значение в диапазоне 0…1. Обратите внимание на то, что ее производную *S′*(*x*) также легко вычислить

7.4. ПОСТРОЕНИЕ СЕТИ

Создадим классы для моделирования всех трех организационных единиц сети: нейронов, слоев и самой сети. Для простоты начнем с самого маленького компонента — нейронов, перейдем к центральному организующему компоненту — слоям, а затем к самому большому — ко всей сети. Переходя от наименьшего компонента к наибольшему, мы инкапсулируем предыдущий уровень. Нейроны знают только о самих себе, слои — о нейронах, которые они содержат, и других слоях, а сеть — обо всех слоях.

ПРИМЕЧАНИЕ

В этой главе приводится много длинных строк кода, которые не помещаются по ширине на печатных страницах книги. Я настоятельно рекомендую при чтении загрузить прилагаемый к ней исходный код из репозитория исходных кодов книги и отслеживать его на экране компьютера: https://github.com/davecom/ClassicComputerScienceProblemsInJava.

7.4.1. Реализация нейронов

Начнем с нейронов. Каждый нейрон должен хранить много элементов состояния, включая вес, дельту, скорость обучения, кэш последних выходных данных, функцию активации, а также производную от нее. Некоторые из этих элементов эффективнее было бы хранить в слое (в будущем классе `Layer`), но они включены в представленный далее класс `Neuron` из соображений наглядности (листинг 7.3).

Листинг 7.3. Neuron.java

```
package chapter7;

import java.util.function.DoubleUnaryOperator;

public class Neuron {
    public double[] weights;
    public final double learningRate;
    public double outputCache;
    public double delta;
    public final DoubleUnaryOperator activationFunction;
    public final DoubleUnaryOperator derivativeActivationFunction;

    public Neuron(double[] weights, double learningRate, DoubleUnaryOperator
      activationFunction, DoubleUnaryOperator derivativeActivationFunction) {
        this.weights = weights;
        this.learningRate = learningRate;
        outputCache = 0.0;
        delta = 0.0;
        this.activationFunction = activationFunction;
        this.derivativeActivationFunction = derivativeActivationFunction;
    }

    public double output(double[] inputs) {
        outputCache = Util.dotProduct(inputs, weights);
        return activationFunction.applyAsDouble(outputCache);
    }
}
```

Большинство этих параметров инициализируется в конструкторе. Поскольку `delta` и `outputCache` неизвестны при первом создании `Neuron`, они просто инициализируются нулем. Некоторые из этих переменных (`learningRate`,

activationFunction, derivativeActivationFunction) выглядят предустановленными, но все же есть причина сделать их изменяемыми. Если класс Neuron будет использоваться для других типов нейронных сетей, то, возможно, некоторые из этих значений будут изменяться в процессе выполнения программы, поэтому их можно настраивать для максимальной гибкости. Существуют даже нейронные сети, которые изменяют скорость обучения по мере приближения к решению и автоматически пробуют разные функции активации. Поскольку наши переменные — final, они не могут быть изменены в середине потока, но чтобы сделать их неокончательными, можно просто изменить код.

Кроме конструктора, у класса есть только один метод — output(). output() принимает входные сигналы (входные данные), поступающие в нейрон, и применяет к ним формулу, рассмотренную ранее в этой главе (см. рис. 7.2). Входные сигналы объединяются с весами посредством скалярного произведения, и результат кэшируется в outputCache. Напомню, что это значение, полученное до того, как была задействована функция активации, используется для вычисления дельты (см. раздел об обратном распространении). Наконец, прежде чем сигнал будет отправлен на следующий слой (будучи возвращенным из output()), к нему применяется функция активации.

Вот и все! Отдельный нейрон в этой сети довольно прост. Он не может сделать ничего иного, кроме как принять входной сигнал, преобразовать его и передать для дальнейшей обработки. Нейрон поддерживает несколько элементов состояния, которые используются другими классами.

7.4.2. Реализация слоев

Слой в нашей сети должен поддерживать три элемента состояния: свои нейроны, предшествующий слой и выходной кэш. Выходной кэш похож на кэш нейрона, но на один слой выше. Он кэширует выходные данные каждого нейрона в данном слое после применения функций активации.

В момент создания слоя основной задачей является инициализация его нейронов. Поэтому конструктору класса Layer необходимо знать, сколько нейронов требуется инициализировать, какими должны быть их функции активации и какова скорость обучения. В нашей простой сети у всех нейронов слоя функция активации и скорость обучения одинаковы (листинг 7.4).

Листинг 7.4. Layer.java

```
package chapter7;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
```

```
import java.util.Random;
import java.util.function.DoubleUnaryOperator;

public class Layer {
    public Optional<Layer> previousLayer;
    public List<Neuron> neurons = new ArrayList<>();
    public double[] outputCache;

    public Layer(Optional<Layer> previousLayer, int numNeurons,
      double learningRate, DoubleUnaryOperator activationFunction,
      DoubleUnaryOperator derivativeActivationFunction) {
        this.previousLayer = previousLayer;
        Random random = new Random();
        for (int i = 0; i < numNeurons; i++) {
            double[] randomWeights = null;
            if (previousLayer.isPresent()) {
                randomWeights = random.doubles(previousLayer.get().
                                neurons.size()).toArray();
            }
            Neuron neuron = new Neuron(randomWeights, learningRate,
              activationFunction, derivativeActivationFunction);
            neurons.add(neuron);
        }
        outputCache = new double[numNeurons];
    }
```

По мере того как сигналы передаются через сеть, их должен обрабатывать каждый нейрон `Layer`. (Помните, что каждый нейрон в слое получает сигналы от каждого нейрона предыдущего слоя.) Именно это делает метод `outputs()`. Он также возвращает результат обработки для передачи по сети на следующий слой и кэширует выходные данные. Если предыдущего слоя нет, значит, данный слой входной и он просто передает сигналы на следующий слой (листинг 7.5).

Листинг 7.5. Layer.java (продолжение)

```
    public double[] outputs(double[] inputs) {
        if (previousLayer.isPresent()) {
            outputCache = neurons.stream().mapToDouble(n ->
              n.output(inputs)).toArray();
        } else {
            outputCache = inputs;
        }
        return outputCache;
    }
```

Существует два типа дельт для вычисления в обратном распространении: дельты для нейронов в выходном слое и дельты для нейронов в скрытых слоях. Их формулы показаны на рис. 7.4 и 7.5, и следующие два метода представляют собой механический перевод этих формул на Java (листинг 7.6). Впоследствии эти методы будут вызываться сетью во время обратного распространения.

Листинг 7.6. Layer.java (продолжение)

```
    // вызывается только для выходного слоя
    public void calculateDeltasForOutputLayer(double[] expected) {
        for (int n = 0; n < neurons.size(); n++) {
            neurons.get(n).delta = neurons.get(n).derivativeActivationFunction.
                applyAsDouble(neurons.get(n). outputCache)
                    * (expected[n] – outputCache[n]);
        }
    }

    // не вызывается для выходного слоя
    public void calculateDeltasForHiddenLayer(Layer nextLayer) {
        for (int i = 0; i < neurons.size(); i++) {
            int index = i;
            double[] nextWeights = nextLayer.neurons.stream().mapToDouble(n ->
                n.weights[index]).toArray();
            double[] nextDeltas = nextLayer.neurons.stream().mapToDouble(n ->
                n.delta).toArray();
            double sumWeightsAndDeltas = Util.dotProduct(nextWeights,
                nextDeltas);
            neurons.get(i).delta = neurons.get(i).derivativeActivationFunction
            .applyAsDouble(neurons.get(i).outputCache) * sumWeightsAndDeltas;
        }
    }
}
```

7.4.3. Реализация сети

Сама сеть хранит только один элемент состояния — слои, которыми она управляет. Класс `Network` отвечает за инициализацию составляющих его слоев.

Конструктор принимает список элементов типа `int`, описывающий структуру сети. Например, список `{2, 4, 3}` описывает сеть, имеющую два нейрона во входном слое, четыре нейрона — в скрытом и три нейрона — в выходном. Работая над этой простой сетью, предполагаем, что все слои сети используют одну и ту же функцию активации для своих нейронов и имеют одинаковую скорость обучения (листинг 7.7).

Листинг 7.7. Network.java

```
package chapter7;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.function.DoubleUnaryOperator;
import java.util.function.Function;
```

```
public class Network<T> {
    private List<Layer> layers = new ArrayList<>();

    public Network(int[] layerStructure, double learningRate,
            DoubleUnaryOperator activationFunction, DoubleUnaryOperator
     derivativeActivationFunction) {
        if (layerStructure.length < 3) {
            throw new IllegalArgumentException("Error: Should be at least 3
                layers (1 input, 1 hidden, 1 output).");
        }
        // входной слой
        Layer inputLayer = new Layer(Optional.empty(), layerStructure[0],
            learningRate, activationFunction, derivativeActivationFunction);
        layers.add(inputLayer);
        // скрытые слои и выходной слой
        for (int i = 1; i < layerStructure.length; i++) {
            Layer nextLayer = new Layer(Optional.of(layers.get(i – 1)),
            layerStructure[i], learningRate, activationFunction,
            derivativeActivationFunction);
            layers.add(nextLayer);
        }
    }
```

ПРИМЕЧАНИЕ

Общий тип T связывает сеть с типом окончательных категорий классификации из набора данных. Он используется только в последнем методе класса validate().

Выходные данные нейронной сети — это результат обработки сигналов, проходящих через все ее слои (листинг 7.8).

Листинг 7.8. Network.java (продолжение)

```
    // Помещает входные данные на первый слой, затем выводит их
    // с первого слоя и подает на второй слой в качестве входных данных,
    // со второго – на третий и т. д.
    private double[] outputs(double[] input) {
        double[] result = input;
        for (Layer layer : layers) {
            result = layer.outputs(result);
        }
        return result;
    }
```

Метод `backpropagate()` отвечает за вычисление дельт для каждого нейрона в сети. В этом методе последовательно задействуются методы `calculateDeltasForOutputLayer()` и `calculateDeltasForHiddenLayer()`. (Напомню, что при обратном распространении дельты вычисляются в обратном порядке.) Метод `backpropagate()` передает ожидаемые значения выходных данных для заданного набора входных данных в функцию `calculateDeltasForOutputLayer()`. Этот метод

использует ожидаемые значения, чтобы найти ошибку, с помощью которой вычисляется дельта (листинг 7.9).

Листинг 7.9. Network.java (продолжение)

```
// Определяет изменения каждого нейрона на основании ошибок
// выходных данных по сравнению с ожидаемым выходом
private void backpropagate(double[] expected) {
    // вычисление дельты для нейронов выходного слоя
    int lastLayer = layers.size() – 1;
    layers.get(lastLayer).calculateDeltasForOutputLayer(expected);
    // вычисление дельты для скрытых слоев в обратном порядке
    for (int i = lastLayer – 1; i >= 0; i--) {
        layers.get(i).calculateDeltasForHiddenLayer(layers.get(i + 1));
    }
}
```

Метод `backpropagate()` отвечает за вычисление всех дельт, но не изменяет веса элементов сети. Для этого после `backpropagate()` должна вызываться функция `updateWeights()`, поскольку изменение веса зависит от дельт (листинг 7.10). Этот метод вытекает непосредственно из формулы, представленной на рис. 7.6.

Листинг 7.10. Network.java (продолжение)

```
// Сама функция backpropagate() не изменяет веса
// Функция update_weights использует дельты, вычисленные в backpropagate(),
// чтобы действительно изменить веса
private void updateWeights() {
    for (Layer layer : layers.subList(1, layers.size())) {
        for (Neuron neuron : layer.neurons) {
            for (int w = 0; w < neuron.weights.length; w++) {
                neuron.weights[w] = neuron.weights[w] + (neuron.learningRate
                * layer.previousLayer.get().outputCache[w] * neuron.delta);
            }
        }
    }
}
```

Веса нейронов изменяются в конце каждого этапа обучения. Для этого в сеть должны быть поданы обучающие наборы данных (входные данные и ожидаемые результаты). Метод `train()` принимает список массивов входных данных и список массивов ожидаемых выходных данных. Каждый набор входных данных пропускается через сеть, после чего ее веса обновляются посредством вызова `backpropagate()` для ожидаемого результата и последующего вызова `updateWeights()`. Попробуйте добавить сюда код, который позволит вывести на печать частоту ошибок, когда через сеть проходит обучающий набор данных. Так вы сможете увидеть, как постепенно уменьшается частота ошибок сети по мере ее продвижения вниз по склону в процессе градиентного спуска (листинг 7.11).

Листинг 7.11. Network.java (продолжение)

```
// Функция train() использует результаты выполнения функции outputs()
// для нескольких входных данных, сравнивает их с ожидаемыми результатами
// и передает полученное функциям backpropagate() и updateWeights()
public void train(List<double[]> inputs, List<double[]> expecteds) {
    for (int i = 0; i < inputs.size(); i++) {
        double[] xs = inputs.get(i);
        double[] ys = expecteds.get(i);
        outputs(xs);
        backpropagate(ys);
        updateWeights();
    }
}
```

Наконец, после обучения сеть необходимо протестировать. Функция `validate()` принимает входные данные и ожидаемые выходные данные так же, как `train()`, но, в отличие от `train()`, использует их не для обучения, а для вычисления процента точности, так как предполагается, что сеть уже обучена. Функция `validate()` принимает также функцию `interpretOutput()`, с помощью которой интерпретируются выходные данные нейронной сети для сравнения с ожидаемыми выходными данными. (Возможно, ожидаемый вывод — это не набор чисел с плавающей запятой, а строка типа `"Amphibian"`.) Функция `interpretOutput()` должна принимать числа с плавающей точкой, полученные в сети, и преобразовывать их в нечто сопоставимое с ожидаемыми выходными данными. Это специальная функция, созданная для конкретного набора данных. Функция `validate()` возвращает количество правильных классификаций, общее количество протестированных образцов и процент правильных классификаций (листинг 7.12).

Листинг 7.12. Network.java (продолжение)

```
public class Results {
    public final int correct;
    public final int trials;
    public final double percentage;

    public Results(int correct, int trials, double percentage) {
        this.correct = correct;
        this.trials = trials;
        this.percentage = percentage;
    }
}

// Для параметризованных результатов, которые требуют классификации,
// эта функция возвращает правильное количество попыток
// и процентное отношение по сравнению с общим количеством
public Results validate(List<double[]> inputs, List<T> expecteds,
    Function<double[], T> interpret) {
    int correct = 0;
    for (int i = 0; i < inputs.size(); i++) {
```

```
            double[] input = inputs.get(i);
            T expected = expecteds.get(i);
            T result = interpret.apply(outputs(input));
            if (result.equals(expected)) {
                correct++;
            }
        }
        double percentage = (double) correct / (double) inputs.size();
        return new Results(correct, inputs.size(), percentage);
    }
}
```

Нейронная сеть готова! Ее можно протестировать на нескольких настоящих задачах. Построенная архитектура достаточно универсальна для того, чтобы с ее помощью можно было решать различные задачи, но мы сосредоточимся на популярной задаче — классификации.

7.5. ЗАДАЧИ КЛАССИФИКАЦИИ

В главе 6 мы классифицировали набор данных посредством кластеризации с помощью метода *k*-средних, не используя заранее известных представлений о том, к какой категории принадлежит каждый элемент данных. При кластеризации мы знаем, что хотим найти категории данных, но заранее не знаем, что это за категории. При решении задачи классификации мы тоже пытаемся классифицировать набор данных, но в этом случае существуют заранее определенные категории. Например, если бы мы пытались классифицировать набор изображений животных, то могли бы перед этим выбрать такие категории, как млекопитающие, рептилии, амфибии, рыбы и птицы.

Существует множество методов машинного обучения, которые можно задействовать в задачах классификации. Возможно, вы слышали о методах опорных векторов, деревьях принятия решений и наивных классификаторах Байеса. Существуют и другие методы. В последнее время нейронные сети стали широко применяться в области классификации. Они требуют больше вычислительных ресурсов, чем некоторые другие алгоритмы классификации, но их способность классифицировать казалось бы произвольные виды данных делает их эффективной технологией. Классификаторы нейронных сетей лежат в основе многих интересных методов классификаций изображений, которые применяются в современном программном обеспечении для обработки фотографий.

Почему вновь возник интерес к использованию нейронных сетей для задач классификации? Аппаратное обеспечение стало достаточно быстрым для того, чтобы окупилась необходимость дополнительных вычислений по сравнению с другими алгоритмами.

7.5.1. Нормализация данных

Наборы данных, с которыми мы предполагаем работать, обычно требуют некоторой очистки, прежде чем их можно будет ввести в алгоритмы. Очистка может означать удаление посторонних символов и дубликатов, исправление ошибок и другие вспомогательные операции. Вид очистки, которую нужно выполнить для двух наборов данных, с которыми предстоит работать, — это нормализация. В главе 6 она сделана с помощью метода `zScoreNormalize()` в классе `KMeans`. Нормализация — это приведение атрибутов, записанных в разных масштабах, к единому масштабу.

Благодаря сигмоидной функции активации каждый нейрон в сети выводит значения в диапазоне 0...1. Логично, что шкала от 0 до 1 будет иметь смысл и для атрибутов в нашем наборе входных данных. Преобразовать шкалу из некоторого диапазона в диапазон 0...1 не составляет труда. Для любого значения `V` в определенном диапазоне атрибутов с максимальным `max` и минимальным `min` значениями формула имеет вид `newV = (oldV – min)/(max – min)`. Эта операция называется *масштабированием объектов*. Далее представлены реализация формулы на Java, добавленная в класс `Util`, а также два служебных метода для загрузки данных из файлов CSV и нахождения удобного максимального значения в массиве (листинг 7.13).

Листинг 7.13. Util.java (продолжение)

```
// Будем считать, что все строки одинаковой длины,
// а каждый столбец масштабирован в диапазоне 0...1
public static void normalizeByFeatureScaling(List<double[]> dataset) {
    for (int colNum = 0; colNum < dataset.get(0).length; colNum++) {
        List<Double> column = new ArrayList<>();
        for (double[] row : dataset) {
            column.add(row[colNum]);
        }
        double maximum = Collections.max(column);
        double minimum = Collections.min(column);
        double difference = maximum – minimum;
        for (double[] row : dataset) {
            row[colNum] = (row[colNum] – minimum) / difference;
        }
    }
}

// Загрузка файла CSV в список массивов строк
public static List<String[]> loadCSV(String filename) {
    try (InputStream inputStream = Util.class.getResourceAsStream(filename)) {
        InputStreamReader inputStreamReader = new
            InputStreamReader(inputStream);
        BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
        return bufferedReader.lines().map(line -> line.split(","))
```

```
                    .collect(Collectors.toList());
        }
        catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException(e.getMessage(), e);
        }
    }

    // Нахождение максимума в массиве
    public static double max(double[] numbers) {
        return Arrays.stream(numbers)
                .max()
                .orElse(Double.MIN_VALUE);
    }
}
```

Обратите внимание на параметр `dataset` в функции `normalizeByFeatureScaling()`. Он указывает на список массивов, который будет изменен в этой функции. Другими словами, функция `normalizeByFeatureScaling()` получает не копию набора данных, а ссылку на исходный набор данных. В этом случае мы хотим внести изменения в значение, а не возвратить его измененную копию. Java передает все по значению, но в этом случае мы передаем ссылку по значению, поэтому получаем копию ссылки на тот же список.

Обратите внимание также на то, что программа предполагает, что наборы данных являются двумерными списками данных типа `double`.

7.5.2. Классический набор данных радужной оболочки

Подобно существованию классических задач информатики, существуют и классические наборы данных для машинного обучения. Они используются для проверки новых методов и сравнения их с существующими. А также служат хорошей отправной точкой для тех, кто изучает машинное обучение. Возможно, самый известный — это набор данных об ирисах. Собранный в 1930-х годах, этот набор данных состоит из 150 образцов ирисов (красивые цветы), разделенных на три вида по 50 растений в каждом. У всех растений измерены четыре атрибута: длина чашелистика, ширина чашелистика, длина лепестка и ширина лепестка.

Стоит отметить: нейронной сети безразлично, что представляют собой атрибуты. С точки зрения важности ее модель обучения не делает различий между длиной чашелистика и длиной лепестка. Если такое различие должно быть сделано, то пользователю нейронной сети следует выполнить соответствующую корректировку.

В репозитории исходного кода, прилагаемого к этой книге, есть файл с данными, разделенными запятыми (в формате CSV), который содержит набор данных об ирисах[1]. Этот набор получен из репозитория машинного обучения UCI Калифорнийского университета[2]. CSV-файл — это просто текстовый файл со значениями, разделенными запятыми. Это общий формат обмена табличными данными, включая электронные таблицы.

Вот несколько строк из файла `iris.csv`:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

Каждая строка описывает отдельную единицу данных. Четыре числа представляют собой четыре атрибута (длина и ширина чашелистика, длина и ширина лепестка), которые, с нашей точки зрения, являются произвольными в отношении того, что они на самом деле представляют. Название в конце каждой строки соответствует определенному виду ириса. Все пять строк относятся к одному и тому же виду — данный образец был взят из верхней части файла, и три вида сгруппированы вместе по 50 строк в каждом.

Для того чтобы прочитать с диска CSV-файл, используем несколько функций из стандартной библиотеки Java. Они содержатся в методе `loadCSV()`, который мы ранее определили в классе `Util`. Помимо этих нескольких строк, остальная часть конструктора `IrisTest`, класса для фактического выполнения классификации, просто переупорядочивает данные из CSV-файла, чтобы подготовить их к применению нашей сетью для обучения и проверки (листинг 7.14).

Листинг 7.14. IrisTest.java

```
package chapter7;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class IrisTest {
    public static final String IRIS_SETOSA = "Iris-setosa";
    public static final String IRIS_VERSICOLOR = "Iris-versicolor";
```

[1] Этот репозиторий доступен на GitHub по адресу https://github.com/davecom/ClassicComputerScienceProblemsInJava.

[2] *Lichman M.* UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science, 2013. http://archive.ics.uci.edu/ml.

```
public static final String IRIS_VIRGINICA = "Iris-virginica";

private List<double[]> irisParameters = new ArrayList<>();
private List<double[]> irisClassifications = new ArrayList<>();
private List<String> irisSpecies = new ArrayList<>();

public IrisTest() {
    // убедитесь, что iris.csv расположен правильно
    List<String[]> irisDataset =
      Util.loadCSV("/chapter7/data/iris.csv");
    // получаем наши строки данных в случайном порядке
    Collections.shuffle(irisDataset);
    for (String[] iris : irisDataset) {
        // первые четыре пункта — параметры (doubles)
        double[] parameters = Arrays.stream(iris)
                .limit(4)
                .mapToDouble(Double::parseDouble)
                .toArray();
        irisParameters.add(parameters);
        // последний пункт — это виды
        String species = iris[4];
        switch (species) {
            case IRIS_SETOSA :
                irisClassifications.add(new double[] { 1.0, 0.0, 0.0 });
                break;
            case IRIS_VERSICOLOR :
                irisClassifications.add(new double[] { 0.0, 1.0, 0.0 });
                break;
            default :
                irisClassifications.add(new double[] { 0.0, 0.0, 1.0 });
                break;
        }
        irisSpecies.add(species);
    }
    Util.normalizeByFeatureScaling(irisParameters);
}
```

`irisParameters` представляет собой коллекцию из четырех атрибутов для каждого образца. Применим эту коллекцию для классификации каждого ириса. `irisClassifications` — это фактическая классификация любого образца. В нашей нейронной сети будет три выходных нейрона, каждый из которых соответствует одному из возможных видов. Например, окончательный набор выходных данных `{0.9, 0.3, 0.1}` будет означать классификацию `Iris-setosa`, поскольку этому виду соответствует первый нейрон и он выдал наибольшее число.

Мы уже знаем правильные ответы для обучения, поэтому у каждого ириса есть готовый ответ. Для цветка, который должен иметь тип `Iris-setosa`, запись в `irisClassifications` будет иметь вид `{1.0, 0.0, 0.0}`. Эти значения станут использоваться для расчета ошибки после каждого этапа обучения. В `IrisSpecies` хранятся непосредственные соответствия классификаций для каждого цветка

на английском языке. Ирис вида `Iris-setosa` будет отмечен в наборе данных как `"Iris-setosa"`.

ВНИМАНИЕ

Отсутствие блока проверки ошибок делает этот код весьма опасным. Он не подходит для практической работы, но вполне пригоден для тестовых целей.

Теперь определим саму нейронную сеть (листинг 7.15).

Листинг 7.15. IrisTest.java (продолжение)

```
public String irisInterpretOutput(double[] output) {
    double max = Util.max(output);
    if (max == output[0]) {
        return IRIS_SETOSA;
    } else if (max == output[1]) {
        return IRIS_VERSICOLOR;
    } else {
        return IRIS_VIRGINICA;
    }
}
```

`irisInterpretOutput()` — служебная функция, которая передается методу сети `validate()` для определения правильных классификаций.

Наконец сеть готова к обучению (листинг 7.16). Определим метод `classify()`, который настроит сеть, обучит ее и запустит.

Листинг 7.16. IrisTest.java (продолжение)

```
public Network<String>.Results classify() {
    // 4-, 6-, 3-слойная структура; 0,3 — скорость обучения;
    // sigmoid — функция активации
    Network<String> irisNetwork = new Network<>(new int[] { 4, 6, 3 },
        0.3, Util::sigmoid, Util::derivativeSigmoid);
```

Аргумент `layerStructure` определяет сеть с тремя слоями (одним входным, одним скрытым и одним выходным) с параметрами `{4, 6, 3}`. Входной слой имеет четыре нейрона, скрытый — шесть, а выходной — три. Четыре нейрона входного слоя соответствуют четырем параметрам, используемым для классификации каждого образца. Три нейрона выходного слоя соответствуют непосредственно трем разным видам, к которым мы стремимся отнести каждый входной элемент. Шесть нейронов скрытого слоя являются скорее результатом проб и ошибок, чем некоей формулой. То же самое относится и к `LearningRate`. Эти два значения (количество нейронов в скрытом слое и скорость обучения) могут быть получены экспериментально, если точность сети окажется ниже оптимальной (листинг 7.17).

Листинг 7.17. IrisTest.java (продолжение)

```
    // обучение для первых 140 ирисов из набора данных, и так 50 раз
    List<double[]> irisTrainers = irisParameters.subList(0, 140);
    List<double[]> irisTrainersCorrects = irisClassifications.subList(0, 140);
    int trainingIterations = 50;
    for (int i = 0; i < trainingIterations; i++) {
        irisNetwork.train(irisTrainers, irisTrainersCorrects);
    }
```

Обучаем сеть на первых 140 ирисах из набора данных, который состоит из 150 ирисов. Напомню, что строки, прочитанные из CSV-файла, были перетасованы. Это гарантирует, что при каждом запуске программы сеть обучается на разных подмножествах набора данных. Обратите внимание на то, что мы обучаем сеть 50 раз на 140 ирисах. Изменение этого значения сильно влияет на количество времени, которое потребуется на обучение нейронной сети. Как правило, чем дольше обучение, тем точнее будет работать нейронная сеть, хотя есть риск так называемого *переобучения*. Финальным тестом станет проверка правильности классификации последних десяти ирисов из набора данных. Выполним это в конце функции `classify()` и запустим сеть из функции `main()` (листинг 7.18).

Листинг 7.18. IrisTest.java (продолжение)

```
        // тест на последних десяти ирисах из набора данных
        List<double[]> irisTesters = irisParameters.subList(140, 150);
        List<String> irisTestersCorrects = irisSpecies.subList(140, 150);
        return irisNetwork.validate(irisTesters, irisTestersCorrects,
          this::irisInterpretOutput);
    }

    public static void main(String[] args) {
        IrisTest irisTest = new IrisTest();
        Network<String>.Results results = irisTest.classify();
        System.out.println(results.correct + " correct of " +
          results.trials + " = " + results.percentage * 100 + "%");
    }

}
```

Вся работа сводится к этому последнему вопросу: сколько из десяти случайно выбранных из набора данных ирисов будут правильно классифицированы нейронной сетью? Поскольку начальные веса каждого нейрона определялись случайным образом, разные прогоны могут давать разные результаты. Можете попробовать настроить скорость обучения, количество скрытых нейронов и количество итераций обучения, чтобы сделать сеть более точной.

В итоге вы должны увидеть следующий результат:

```
9 correct of 10 = 90.0%
```

7.5.3. Классификация вина

Мы собираемся протестировать нейронную сеть на другом наборе данных, основанном на химическом анализе сортов итальянских вин[1]. Этот набор данных содержит 178 образцов. Механизм работы с ним будет примерно таким же, что и с набором данных об ирисах, но структура CSV-файла немного иная. Вот его фрагмент:

```
1,14.23,1.71,2.43,15.6,127,2.8,3.06,.28,2.29,5.64,1.04,3.92,1065
1,13.2,1.78,2.14,11.2,100,2.65,2.76,.26,1.28,4.38,1.05,3.4,1050
1,13.16,2.36,2.67,18.6,101,2.8,3.24,.3,2.81,5.68,1.03,3.17,1185
1,14.37,1.95,2.5,16.8,113,3.85,3.49,.24,2.18,7.8,.86,3.45,1480
1,13.24,2.59,2.87,21,118,2.8,2.69,.39,1.82,4.32,1.04,2.93,735
```

Первым значением в каждой строке всегда будет целое число от 1 до 3, представляющее один из трех сортов, к которым может принадлежать образец. Но обратите внимание на то, сколько здесь еще параметров для классификации. В наборе данных об ирисах было только четыре параметра. В наборе данных о винах их тринадцать.

Наша модель нейронной сети отлично масштабируется, нужно только увеличить количество входных нейронов. Файл `WineTest.java` аналогичен `IrisTest.java`, но в него внесено несколько незначительных изменений, чтобы учесть различия в их структуре (листинг 7.19).

Листинг 7.19. WineTest.java

```
package chapter7;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class WineTest {
    private List<double[]> wineParameters = new ArrayList<>();
    private List<double[]> wineClassifications = new ArrayList<>();
    private List<Integer> wineSpecies = new ArrayList<>();

    public WineTest() {
        // убедитесь, что файл wine.csv расположен правильно
        List<String[]> wineDataset = Util.loadCSV("/chapter7/data/wine.csv");
        // получаем наши строки данных в случайном порядке
        Collections.shuffle(wineDataset);
        for (String[] wine : wineDataset) {
            // последние тринадцать пунктов — параметры (doubles)
```

[1] *Lichman M.* UCI Machine Learning Repository. — Irvine, CA: University of California, School of Information and Computer Science, 2013. http://archive.ics.uci.edu/ml.

```
            double[] parameters = Arrays.stream(wine)
                    .skip(1)
                    .mapToDouble(Double::parseDouble)
                    .toArray();
            wineParameters.add(parameters);
            // первый пункт – это виды
            int species = Integer.parseInt(wine[0]);
            switch (species) {
                case 1 :
                    wineClassifications.add(new double[] { 1.0, 0.0, 0.0 });
                    break;
                case 2 :
                    wineClassifications.add(new double[] { 0.0, 1.0, 0.0 });
                    break;
                default :
                    wineClassifications.add(new double[] { 0.0, 0.0, 1.0 });
                    break;
            }
            wineSpecies.add(species);
        }
        Util.normalizeByFeatureScaling(wineParameters);
    }
```

Функция `wineInterpretOutput()` аналогична функции `irisInterpretOutput()`. Поскольку у нас нет названий сортов вина, мы просто работаем с целочисленными значениями в исходном наборе данных (листинг 7.20).

Листинг 7.20. WineTest.java (продолжение)

```
    public Integer wineInterpretOutput(double[] output) {
        double max = Util.max(output);
        if (max == output[0]) {
            return 1;
        } else if (max == output[1]) {
            return 2;
        } else {
            return 3;
        }
    }
```

Как уже упоминалось, конфигурация слоя в сети для классификации сортов вин требует 13 входных нейронов (по одному для каждого параметра). Кроме того, нужны три выходных нейрона. Существует три сорта вина, точно так же как раньше было три вида ирисов. Интересно, что сеть хорошо работает, когда число нейронов в скрытом слое меньше, чем во входном (листинг 7.21). Одним из возможных интуитивных объяснений этого является то, что некоторые входные параметры на самом деле не помогают выполнить классификацию и их лучше опустить во время обработки. На самом деле меньшее количество нейронов в скрытом слое работает не совсем так, но это интересное предположение.

Листинг 7.21. WineTest.java (продолжение)

```
public Network<Integer>.Results classify() {
    // 13-, 7-, 3-слойная структура; 0,9 — скорость обучения;
    // sigmoid — функция активации
    Network<Integer> wineNetwork = new Network<>(new int[] { 13, 7, 3 },
      0.9, Util::sigmoid, Util::derivativeSigmoid);
```

Подчеркну еще раз: иногда интересно поэкспериментировать с другим количеством скрытых нейронов или с другой скоростью обучения (листинг 7.22).

Листинг 7.22. WineTest.java (продолжение)

```
    // обучение на первых 150 образцах вина, повторяется 50 раз
    List<double[]> wineTrainers = wineParameters.subList(0, 150);
    List<double[]> wineTrainersCorrects = wineClassifications.subList(0, 150);
    int trainingIterations = 10;
    for (int i = 0; i < trainingIterations; i++) {
        wineNetwork.train(wineTrainers, wineTrainersCorrects);
    }
```

Функция `wine_interpret_output()` аналогична `iris_interpret_output()`. Поскольку у нас нет названий сортов вин, просто присваиваем сортам целочисленные номера и используем их в исходном наборе данных.

Будем обучать сеть на первых 150 образцах из набора данных, оставляя последние 28 для проверки, и повторим обучение десять раз для каждой выборки, что значительно меньше, чем 50 для набора данных об ирисах. По какой-то причине (возможно, из-за специфических свойств этого набора данных или настроек параметров, таких как скорость обучения и количество скрытых нейронов) этот набор данных требует более короткого обучения для достижения значительной точности, чем набор данных об ирисах (листинг 7.23).

Листинг 7.23. WineTest.java (продолжение)

```
        // тестирование на последних 28 образцах вина в наборе данных
        List<double[]> wineTesters = wineParameters.subList(150, 178);
        List<Integer> wineTestersCorrects = wineSpecies.subList(150, 178);
        return wineNetwork.validate(wineTesters, wineTestersCorrects,
          this::wineInterpretOutput);
    }

    public static void main(String[] args) {
        WineTest wineTest = new WineTest();
        Network<Integer>.Results results = wineTest.classify();
        System.out.println(results.correct + " correct of " + results.trials
          + " = " + results.percentage * 100 + "%");
    }
}
```

Если повезет, то ваша нейронная сеть сможет довольно точно классифицировать 28 образцов:

```
27 correct of 28 = 96.42857142857143%
```

7.6. ПОВЫШЕНИЕ СКОРОСТИ РАБОТЫ НЕЙРОННОЙ СЕТИ

Нейронные сети требуют множества операций с векторами и матрицами. По сути, это означает, что нужно взять список чисел и выполнить операцию для всех них одновременно. Поскольку машинное обучение продолжает проникать в наше общество, библиотеки для оптимизированной, эффективной векторной и матричной математики приобретают все большее значение. Во многих библиотеках используются преимущества графических процессоров, поскольку они оптимизированы для этой роли. (На векторах и матрицах базируется компьютерная графика.) Более старая спецификация библиотеки, о которой вы, возможно, слышали, называется BLAS (Basic Linear Algebra Subprograms — подпрограммы базовой линейной алгебры). Реализация BLAS лежит в основе популярной числовой библиотеки Java ND4J.

Помимо графических процессоров, можно задействовать расширения центрального процессора, позволяющие ускорить обработку векторов и матриц. В состав BLAS входят функции, которые задействуют *SIMD*-инструкции (*single instruction, multiple data* — одна инструкция, несколько данных). SIMD-инструкции — это специальные инструкции для микропроцессора, которые позволяют обрабатывать несколько фрагментов данных одновременно. Их иногда называют также *векторными инструкциями*.

Разные микропроцессоры содержат различные SIMD-инструкции. Например, SIMD-расширение для G4 (процессор архитектуры PowerPC, который встречается в ранних версиях Mac) называлось AltiVec. Микропроцессоры ARM-архитектуры, а также те, что устанавливаются в iPhone, имеют расширение NEON. Современные микропроцессоры Intel включают в себя SIMD-расширения, известные как MMX, SSE, SSE2 и SSE3. К счастью, вам не нужно знать их все. И нет необходимости знать различия между ними. Хорошо отлаженная числовая библиотека автоматически выберет правильные инструкции для эффективных вычислений на базовой архитектуре, на которой работает ваша программа.

Поэтому неудивительно, что реальные библиотеки нейронных сетей, в отличие от игрушечной библиотеки, созданной в этой главе, используют в качестве базовой структуры данных специализированные типы вместо списков или массивов стандартных библиотек Java. Но этим они не ограничиваются. Популярные

библиотеки нейронных сетей Java, такие как TensorFlow и PyTorch, не только задействуют SIMD-инструкции, но и широко применяют вычисления на GPU. Поскольку графические процессоры специально оптимизированы для быстрых векторных вычислений, это на порядок ускоряет работу нейронных сетей по сравнению с работой только на центральном процессоре.

Подчеркну еще раз: *вы вряд ли станете строить нейронную сеть для практической работы, используя только стандартную библиотеку Java, как мы делали в этой главе.* Вместо этого следует задействовать хорошо оптимизированную библиотеку с поддержкой SIMD и GPU, такую как TensorFlow. Единственным исключением будет библиотека нейронных сетей, предназначенная для обучения, или библиотека, которая должна работать на встроенном устройстве, не поддерживающем SIMD-инструкции и не имеющем GPU.

7.7. ПРОБЛЕМЫ И РАСШИРЕНИЯ НЕЙРОННЫХ СЕТЕЙ

Благодаря достижениям в области глубокого обучения нейронные сети сейчас в моде, но у них есть существенные недостатки. Самая большая проблема заключается в том, что решение задачи в нейронной сети — это что-то вроде черного ящика. Даже когда нейронные сети работают хорошо, они не позволяют пользователю детально разобраться в том, как именно они решают проблему. Например, классификатор набора данных ирисов, над которым мы работали в этой главе, не особенно ясно показывает, насколько каждый из четырех входных параметров влияет на выходные данные. Что важнее для классификации образца — длина или ширина чашелистика?

Возможно, тщательный анализ окончательных весов в обученной сети может дать некоторое представление об этом, но он нетривиален и не дает такого понимания, которое дает, скажем, линейная регрессия с точки зрения значения каждой переменной в моделируемой функции. Другими словами, нейронная сеть может решить задачу, но не объясняет, как именно это делает.

Другая проблема нейронных сетей заключается в том, что для обеспечения точности им часто нужны очень большие наборы данных. Представьте себе классификатор изображений для пейзажей. Возможно, потребуется классифицировать тысячи различных типов изображений (леса, долины, горы, ручьи, степи и т. п.). Потенциально для этого потребуются миллионы обучающих образов. Нужны будут миллионы обучающих изображений. Такие большие наборы данных не только трудно найти — для некоторых приложений их может вообще не существовать. Как правило, хранилищами данных и техническими средствами для сбора

и хранения таких объемных наборов данных располагают крупные корпорации и правительства.

Наконец, нейронные сети дóроги с вычислительной точки зрения. Даже простое обучение на наборе данных среднего размера может вызвать перегрузку компьютера. И это не просто наивные реализации нейронных сетей. На любой вычислительной платформе, где применяются нейронные сети, выполняется огромное количество вычислений для обучения сети — ничто другое не занимает так много времени. Существует множество приемов, позволяющих повысить производительность нейронной сети, например применение SIMD-инструкций или графических процессоров, но в любом случае обучение нейронной сети требует большого количества операций с плавающей точкой.

Один из приятных нюансов — то, что в вычислительном отношении обучение обходится намного дороже, чем использование сети. Некоторые приложения не требуют постоянного обучения. В этих случаях обученную сеть можно просто вставить в приложение для решения задачи. Например, первая версия платформы Apple Core ML даже не поддерживает обучение. Она только помогает разработчикам приложений запускать предварительно обученные модели нейронных сетей в своих приложениях. Разработчик, создающий приложение для обработки фотографий, может загрузить модель классификации изображений с открытой лицензией, вставить ее в Core ML и сразу же начать задействовать эффективное машинное обучение в своем приложении.

В этой главе мы работали с нейронной сетью только одного типа — с прямой связью и обратным распространением. Как уже упоминалось, существует много других видов нейронных сетей. Сверточные нейронные сети также имеют прямую связь, но у них есть несколько типов скрытых слоев, различные механизмы распределения весов и другие интересные свойства, из-за чего они особенно хорошо подходят для классификации изображений. В рекуррентных нейронных сетях сигналы не просто движутся в одном направлении — такие сети допускают петли обратной связи. Эти сети оказались полезными для приложений непрерывного ввода, распознающих написанное от руки и прочитанное вслух.

Простым расширением нейронной сети, которое могло бы сделать ее более производительной, было бы добавление нейронов смещения. *Нейроны смещения* подобны фиктивным нейронам в слое, который позволяет предоставлять выходным данным следующего слоя больше функций, обеспечивая подачу на него постоянного входного сигнала (все еще измененного посредством весов). Даже простые нейронные сети, используемые для реальных задач, обычно содержат нейроны смещения. Если вы добавите такие нейроны в созданную здесь сеть, то, вероятно, обнаружите, что для достижения аналогичного уровня точности потребуется меньше времени на обучение.

7.8. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

Несмотря на то что искусственные нейронные сети появились еще в середине XX века, до последнего десятилетия они не были широко распространены. Массовое применение нейронных сетей сдерживалось отсутствием достаточно производительного оборудования. Сегодня в машинном обучении искусственные нейронные сети стали областью с поистине взрывным ростом, потому что они работают!

За последние десятилетия искусственные нейронные сети позволили создать одни из самых удивительных компьютерных приложений, ориентированных на пользователя. К ним относятся распознавание голоса (практичное с точки зрения достаточной точности), распознавание изображений и почерка. Технология распознавания голоса присутствует в средствах набора текста, таких как Dragon NaturallySpeaking, и в цифровых помощниках, таких как Siri, Alexa и Cortana. Пример распознавания изображений — автоматическое распознавание людей по фотографиям в Facebook с помощью функции распознавания лиц. В последних версиях iOS с помощью функции распознавания рукописного ввода вы можете искать слова в своих заметках, даже если они написаны от руки.

Более старая технология распознавания, эффективность которой можно повысить за счет использования нейронных сетей, — это оптическое распознавание символов (optical character recognition, OCR). Технология OCR применяется при сканировании документов и возвращает редактируемый текст вместо изображения. OCR позволяет почтовой службе считывать индексы на конвертах для быстрой сортировки корреспонденции.

В этой главе было показано, что нейронные сети успешно действуют в задачах классификации. Подобные приложения, в которых хорошо работают нейронные сети, — это системы выдачи рекомендаций. Именно так Netflix предлагает фильм, который вас может заинтересовать, а Amazon — книгу, которую вы, вероятно, захотите прочитать. Существуют и другие методы машинного обучения, хорошо подходящие для систем рекомендаций (Amazon и Netflix не обязательно применяют нейронные сети для этих целей, детали реализации таких систем обычно не разглашаются), поэтому нейронные сети следует выбирать только после того, как все параметры изучены.

Нейронные сети могут применяться в любой ситуации, когда необходима аппроксимация неизвестной функции. Это делает их полезными для прогнозирования. Нейронные сети могут использоваться и используются для прогнозирования результатов спортивных событий, выборов или торгов на фондовом рынке. Конечно, их точность зависит от того, насколько хорошо они обучены, то есть от того, насколько велик был набор данных, относящихся к событию с неизвестным результатом, насколько хорошо настроены параметры

нейронной сети и сколько итераций обучения выполнено. Как и в большинстве приложений с применением нейронных сетей, одна из самых сложных частей прогнозирования — выбор структуры сети, которая часто определяется методом проб и ошибок.

7.9. УПРАЖНЕНИЯ

1. Воспользуйтесь инфраструктурой нейронной сети, разработанной в этой главе, для классификации элементов из другого набора данных.
2. Попробуйте запустить примеры с другой функцией активации (не забудьте также найти ее производную). Как изменение функции активации влияет на точность сети? Требуется ли при этом большее или меньшее обучение?
3. Проработайте заново решения задач из этой главы, используя популярную инфраструктуру нейронных сетей, такую как TensorFlow или PyTorch.
4. Перепишите классы `Network`, `Layer` и `Neuron` с помощью сторонней числовой библиотеки Java, чтобы ускорить работу нейронной сети, разработанной в этой главе.

8 Состязательный поиск

Идеальная информационная игра для двух игроков с нулевой суммой и точной информацией — это игра, в которой оба соперника имеют всю доступную им информацию о состоянии игры и все, что является преимуществом для одного, становится потерей преимущества для другого. К таким играм относятся крестики-нолики, Connect Four, шашки и шахматы. В этой главе вы узнаете, как создать сильного искусственного соперника, способного играть в такие игры. В сущности, обсуждаемые методы в сочетании с современными вычислительными возможностями позволяют создавать искусственных соперников, которые прекрасно играют в простые игры этого класса и способны играть в сложные игры, выходящие за пределы возможностей любого соперника-человека.

8.1. ОСНОВНЫЕ КОМПОНЕНТЫ НАСТОЛЬНОЙ ИГРЫ

Как и при рассмотрении большинства более сложных задач в предыдущих главах, постараемся сделать решение как можно более обобщенным. В случае состязательного поиска это означает, что наши алгоритмы поиска не должны зависеть от игры. Начнем с определения нескольких простых базовых классов, которые выявляют состояние, необходимое алгоритмам поиска. Затем создадим подклассы базовых классов для конкретных игр (крестики-нолики и Connect Four) и введем эти подклассы в алгоритмы поиска, чтобы они могли «играть» в эти игры. Вот базовые классы (листинг 8.1).

`Piece` — это базовый класс для фигуры на доске в игре. Его другая роль — индикатор хода. Именно для этого необходимо свойство `opposite`. Нам нужно знать, чей ход следует за текущим ходом (листинг 8.2).

Листинг 8.1. Piece.java

```
package chapter8;

public interface Piece {
    Piece opposite();
}
```

СОВЕТ

Поскольку в крестиках-ноликах и игре Connect Four существует только один вид фигур, класс Piece в этой главе может использоваться как индикатор хода. В более сложных играх, таких как шахматы, где есть разные виды фигур, ходы могут обозначаться целым числом или логическим значением. В качестве альтернативы можно применять для обозначения хода атрибут color более сложного типа Piece.

Листинг 8.2. Board.java

```
package chapter8;

import java.util.List;

public interface Board<Move> {
    Piece getTurn();

    Board<Move> move(Move location);

    List<Move> getLegalMoves();

    boolean isWin();

    default boolean isDraw() {
        return !isWin() && getLegalMoves().isEmpty();
    }

    double evaluate(Piece player);
}
```

Класс `Board` является фактическим хранилищем состояния. Для любой конкретной игры, которую будут выполнять алгоритмы поиска, они должны уметь ответить на следующие вопросы.

- Чей сейчас ход?
- Какие ходы можно сделать из текущей позиции согласно правилам?
- Выиграна ли сейчас игра?
- Сыграна ли игра вничью?

Последний вопрос, касающийся ничьих, на самом деле комбинация двух предыдущих вопросов для многих игр. Если игра не выиграна, но возможных ходов нет, то это ничья. Вот почему в классе `Board` сразу можно создать конкретную

реализацию метода `isDraw()`. Кроме того, есть еще несколько действий, которые необходимо реализовать.

- Сделать ход, чтобы перейти из текущей в новую позицию.
- Оценить позицию, чтобы увидеть, какой игрок имеет преимущество.

Каждый метод и свойство класса `Board` являются реализацией одного из предыдущих вопросов или действий. На языке игры класс `Board` можно было бы назвать `Position`, но мы используем это имя для чего-то более конкретного в каждом из подклассов.

Класс `Board` имеет общий тип `Move`. Тип `Move` представляет собой ход в игре. Это может быть просто целое число. В таких играх, как крестики-нолики и Connect Four, целое число представляет ход, указывая клетку или столбец, где должна быть размещена фигура. В более сложных играх для описания хода может потребоваться большее число, чем целое число. `Move` позволяет классу `Board` представлять более широкий спектр игр.

8.2. КРЕСТИКИ-НОЛИКИ

Крестики-нолики — простая игра, но ее можно взять для иллюстрации того же минимаксного алгоритма, который применяется в сложных стратегических играх, таких как Connect Four, шашки и шахматы. Мы построим искусственный интеллект, который прекрасно играет в крестики-нолики с помощью минимаксного алгоритма.

ПРИМЕЧАНИЕ

В этом разделе предполагается, что вы знакомы с игрой в крестики-нолики и ее стандартными правилами. Если нет, то, чуть-чуть поискав в интернете, вы найдете их.

8.2.1. Управление состоянием игры в крестики-нолики

Давайте разработаем несколько структур, позволяющих отслеживать состояние игры в крестики-нолики по мере ее развития.

Прежде всего нужен способ представления каждой клетки на игровом поле. Будем использовать перечисление `TTTPiece` — подкласс класса `Piece`. Клетка в игре в крестики-нолики может иметь значение `X`, `0` или быть пустой (в перечислении такие обозначаются `E`) (листинг 8.3).

В классе `TTTPiece` имеется свойство `opposite`, которое возвращает другой экземпляр `TTTPiece`. Это будет полезно для передачи хода от одного игрока к другому

после того, как очередной ход игры завершен. Для представления ходов используем обычное целое число, соответствующее клетке на поле, где был поставлен крестик или нолик. Как вы помните, `Move` был определен в классе `Board`. Укажем, что `Move` при определении `TTTPiece` — целое число.

Листинг 8.3. TTTPiece.java

```
package chapter8;

public enum TTTPiece implements Piece {
    X, O, E; // E — пустая клетка

    @Override
    public TTTPiece opposite() {
        switch (this) {
        case X:
            return TTTPiece.O;
        case O:
            return TTTPiece.X;
        default: // E, пустая клетка
            return TTTPiece.E;
        }
    }

    @Override
    public String toString() {
        switch (this) {
        case X:
            return "X";
        case O:
            return "O";
        default: // E, пустая клетка
            return " ";
        }
    }
}
```

В игре в крестики-нолики существует девять позиций, образующих три ряда по три столбца. Для простоты эти девять позиций можно представить в виде одномерного списка. То, какие клетки получат какие номера (другими словами, индексы в массиве), не имеет значения, но мы будем придерживаться схемы, показанной на рис. 8.1.

0	1	2
3	4	5
6	7	8

Рис. 8.1. Каждой клетке на игровом поле соответствует индекс одномерного списка

Главным хранилищем состояния игры будет класс `TTTBoard`. Он отслеживает два элемента состояния: позицию, представленную вышеупомянутым одномерным списком, и игрока, который сейчас делает ход (листинг 8.4).

Листинг 8.4. TTTBoard.java

```
package chapter8;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class TTTBoard implements Board<Integer> {
    private static final int NUM_SQUARES = 9;
    private TTTPiece[] position;
    private TTTPiece turn;

    public TTTBoard(TTTPiece[] position, TTTPiece turn) {
        this.position = position;
        this.turn = turn;
    }

    public TTTBoard() {
        // по умолчанию в начале игры поле пустое
        position = new TTTPiece[NUM_SQUARES];
        Arrays.fill(position, TTTPiece.E);
        // Первый игрок ставит X
        turn = TTTPiece.X;
    }

    @Override
    public Piece getTurn() {
            return turn;
    }
```

Исходное состояние поля — такое, при котором еще не сделано ни одного хода (пустое поле). Конструктор без параметров для `TTTBoard` инициализирует такую позицию, при которой первый игрок готовится поставить `X` (обычный первый ход в игре). `getTurn()` указывает, чья очередь находится в текущей позиции, `X` или `O`.

`TTTBoard` — это по соглашению неизменяемая структура данных: структуры `TTTBoard` не должны изменяться. Вместо этого каждый раз, когда необходимо сделать ход, будет генерироваться новая структура `TTTBoard`, позиция которой изменена с учетом хода. Впоследствии это пригодится в алгоритме поиска. При ветвлении поиска мы не сможем случайно изменить положение на поле, начиная с которого все еще анализируются потенциально возможные ходы (листинг 8.5).

Допустимым ходом в игре является любая пустая клетка. `getLegalMoves()` ищет любые пустые клетки на поле и возвращает их список (листинг 8.6).

Листинг 8.5. TTTBoard.java (продолжение)

```
@Override
public TTTBoard move(Integer location) {
    TTTPiece[] tempPosition = Arrays.copyOf(position, position.length);
    tempPosition[location] = turn;
    return new TTTBoard(tempPosition, turn.opposite());
}
```

Листинг 8.6. TTTBoard.java (продолжение)

```
@Override
public List<Integer> getLegalMoves() {
    ArrayList<Integer> legalMoves = new ArrayList<>();
    for (int i = 0; i < NUM_SQUARES; i++) {
        // пустые клетки — допустимые ходы
        if (position[i] == TTTPiece.E) {
            legalMoves.add(i);
        }
    }
    return legalMoves;
}
```

Существует множество способов просмотреть строки, столбцы и диагонали на поле игры, чтобы проверить, завершилась ли она победой. В показанной далее реализации метода `isWin()` вместе с его вспомогательным методом `checkPos()` это сделано посредством жестко закодированной конструкции из `&&`, `||` и `==`, которая кажется бесконечной (листинг 8.7). Это не самый красивый код, но он простой и выполняет свою работу.

Листинг 8.7. TTTBoard.java (продолжение)

```
@Override
public boolean isWin() {
    // проверяем три строки, три столбца и две диагонали
    return
    checkPos(0, 1, 2) || checkPos(3, 4, 5) || checkPos(6, 7, 8)
    || checkPos(0, 3, 6) || checkPos(1, 4, 7) || checkPos(2, 5, 8)
    || checkPos(0, 4, 8) || checkPos(2, 4, 6);
}

private boolean checkPos(int p0, int p1, int p2) {
    return position[p0] == position[p1] && position[p0] ==
      position[p2] && position[p0] != TTTPiece.E;
}
```

Если все клетки строки, столбца или диагонали не пустые и содержат одинаковые элементы, то игра выиграна.

Игра закончена вничью, если она не выиграна и не осталось допустимых ходов (это свойство уже описано в методе `isDraw()` класса `Board`). Наконец, нам нужен способ оценки конкретной позиции и структурного вывода состояния поля (листинг 8.8).

Листинг 8.8. TTTBoard.java (продолжение)

```
    @Override
    public double evaluate(Piece player) {
        if (isWin() && turn == player) {
            return -1;
        } else if (isWin() && turn != player) {
            return 1;
        } else {
            return 0.0;
        }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int row = 0; row < 3; row++) {
            for (int col = 0; col < 3; col++) {
                sb.append(position[row * 3 + col].toString());
                if (col != 2) {
                    sb.append("|");
                }
            }
            sb.append(System.lineSeparator());
            if (row != 2) {
                sb.append("-----");
                sb.append(System.lineSeparator());
            }
        }
        return sb.toString();
    }
}
```

В большинстве игр приходится вычислять позицию приблизительно, поскольку мы не можем пройти игру до самого конца, чтобы точно определить, кто выиграет или проиграет в зависимости от того, какие ходы будут сделаны. Но в крестиках-ноликах довольно малое пространство поиска, так что мы можем пройти его от любой позиции до конца игры. Следовательно, метод `evaluate()` может просто возвращать некое число, если игрок выиграл, меньшее число — в случае ничьей и совсем малое — в случае проигрыша.

8.2.2. Минимакс

Минимакс — это классический алгоритм для поиска наилучшего хода в играх с двумя игроками, нулевой суммой и отличной информацией, таких как крестики-нолики, шашки или шахматы. Этот алгоритм был расширен и модифицирован для других типов игр. Минимакс обычно реализуется с использованием рекурсивной функции, в которой каждый игрок обозначается как максимизирующий или минимизирующий.

Максимизирующий игрок стремится найти ход, который приведет к максимальному выигрышу. Однако он должен учитывать ходы минимизирующего игрока. После каждой попытки максимизирующего игрока максимизировать выигрыш рекурсивно вызывается минимакс, чтобы найти ответ противника, который минимизирует максимизирующий выигрыш игрока. Это продолжается в обоих направлениях (максимизация, минимизация, максимизация и т. д.), пока не будет достигнут базовый случай рекурсивной функции. Базовый случай — это конечная позиция (выигрыш или ничья) либо достижение максимальной глубины поиска.

Минимакс вычисляет стартовую позицию для максимизирующего игрока. Для метода `evaluate()` класса `TTTBoard`, если наилучшая возможная игра обеих сторон приведет к выигрышу максимизирующего игрока, возвращается 1 балл. Если наилучшая возможная игра приведет к проигрышу, то возвращается –1. Наконец, если наилучшая игра — это ничья, то возвращается 0.

Эти числа возвращаются при достижении базового случая. Затем они передаются через все рекурсивные вызовы, которые привели к базовому случаю. Чтобы максимизировать каждый рекурсивный вызов, вверх по рекурсии передаются наилучшие вычисленные ходы. Чтобы минимизировать каждый рекурсивный вызов, передаются худшие вычисленные ходы. Таким образом строится дерево решений. На рис. 8.2 показано такое дерево, которое упрощает передачу данных вверх по рекурсивным вызовам для игры, в которой осталось сделать два хода.

В играх, имеющих слишком глубокое пространство поиска, таких как шашки и шахматы, чтобы достичь конечной позиции, минимакс останавливается после достижения определенной глубины (выполнения заданного количества ходов поиска, иногда называемого *ply*). Затем включается функция оценки, использующая эвристику для оценки состояния игры. Чем лучше выглядит игра для первого игрока, тем выше присуждаемая ей оценка. Мы вернемся к этой концепции, когда будем обсуждать игру Connect Four, которая имеет гораздо большее пространство поиска, чем крестики-нолики.

В листинге 8.9 представлена полная реализация функции `minimax()`.

Листинг 8.9. Minimax.java

```
package chapter8;

public class Minimax {
    // Находим наилучший из возможных результатов для первого игрока
    public static <Move> double minimax(Board<Move> board,
      Boolean maximizing, Piece originalPlayer, int maxDepth) {
        // Базовый случай – достигнута финальная
        // позиция или максимальная глубина поиска
        if (board.isWin() || board.isDraw() || maxDepth == 0) {
            return board.evaluate(originalPlayer);
        }
```

```
        // Рекурсивный случай – максимизируйте свою выгоду или
        // минимизируйте выгоду противника
        if (maximizing) {
            double bestEval = Double.NEGATIVE_INFINITY; // результат выше
            for (Move move : board.getLegalMoves()) {
                double result = minimax(board.move(move), false,
                  originalPlayer, maxDepth – 1);
                bestEval = Math.max(result, bestEval);
            }
            return bestEval;
        } else { // минимизация
            double worstEval = Double.POSITIVE_INFINITY; // результат ниже
            for (Move move : board.getLegalMoves()) {
                double result = minimax(board.move(move), true,
                  originalPlayer, maxDepth – 1);
                worstEval = Math.min(result, worstEval);
            }
            return worstEval;
        }
    }
```

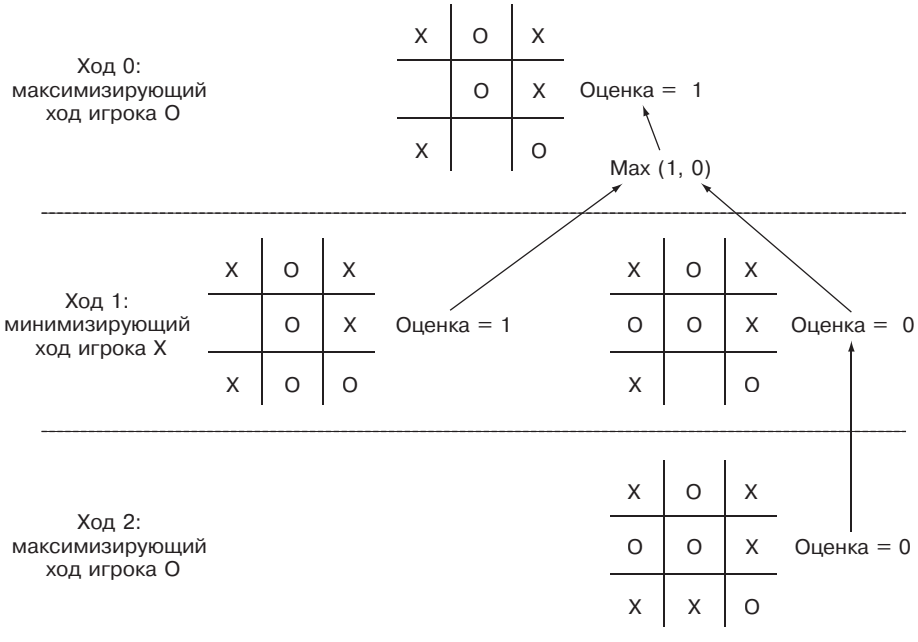


Рис. 8.2. Минимаксное дерево решений для игры в крестики-нолики, в которой осталось сделать два хода. Чтобы максимизировать вероятность выигрыша, первоначальный игрок, 0, должен поставить 0 по центру в нижнем ряду. Стрелки указывают позиции, с которых принимается решение

При каждом рекурсивном вызове нужно отслеживать позицию на поле независимо от того, максимизируем мы ее или минимизируем и для кого пытаемся оценить позицию (`originalPlayer`).

Первые несколько строк функции `minimax()` касаются базового случая — заключительной позиции (выигрыш, проигрыш или ничья) или максимально достижимой глубины. Остальная часть функции обрабатывает рекурсивные случаи.

Один из рекурсивных случаев — это максимизация. В этой ситуации мы ищем ход, который даст максимально возможную оценку. Второй рекурсивный случай — минимизация: ищем ход, который приведет к наименьшей возможной оценке. Как бы то ни было, эти два случая чередуются, пока не будет достигнута финальная позиция или максимальная глубина поиска (базовый случай).

К сожалению, мы не можем использовать нашу реализацию `minimax()`, чтобы найти наилучший ход для данной позиции, так как функция возвращает оценку — значение с плавающей запятой. Оно не сообщает, какой лучший первый ход привел к данной оценке.

Вместо этого создадим вспомогательную функцию `findBestMove()`, которая перебирает вызовы `minimax()` для каждого допустимого хода из данной позиции и позволяет найти ход, который имел бы максимальную оценку. Функцию `findBestMove()` можно представить как первый максимизирующий вызов `minimax()`, но с отслеживанием начальных ходов (листинг 8.10).

Листинг 8.10. Minimax.java (продолжение)

```
    // Найти наилучший возможный ход из текущей
    // позиции, просматривая maxDepth ходов вперед
    public static <Move> Move findBestMove(Board<Move> board, int maxDepth) {
        double bestEval = Double.NEGATIVE_INFINITY;
        Move bestMove = null; // Наверняка не примет значение null
        for (Move move : board.getLegalMoves()) {
            double result = minimax(board.move(move), false,
              board.getTurn(), maxDepth);
            if (result > bestEval) {
                bestEval = result;
                bestMove = move;
            }
        }
        return bestMove;
    }
}
```

Теперь у нас есть все необходимое, чтобы найти наилучший возможный ход для любой позиции при игре в крестики-нолики.

8.2.3. Тестирование минимакса для игры в крестики-нолики

Крестики-нолики — такая простая игра, что даже нам, людям, не составляет труда найти в ней правильный ход для текущей позиции. Это позволяет легко разрабатывать модульные тесты. В следующем фрагменте кода мы испытаем минимаксный алгоритм и попытаемся найти правильный ход среди трех различных позиций. Первый тест простой: требуется найти лишь один ход, чтобы одержать победу. Второй тест требует блокировать соперника — искусственный интеллект должен помешать противнику одержать победу. Последний тест немного сложнее и требует от искусственного интеллекта продумать игру на два шага вперед (листинг 8.11).

Листинг 8.11. TTTMinimaxTests.java

```
package chapter8;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Method;

// Аннотация для модульных тестов
@Retention(RetentionPolicy.RUNTIME)
@interface UnitTest {
    String name() default "";
}

public class TTTMinimaxTests {

    // Проверьте, равны ли два значения, и сообщите об этом
    public static <T> void assertEquality(T actual, T expected) {
        if (actual.equals(expected)) {
            System.out.println("Passed!");
        } else {
            System.out.println("Failed!");
            System.out.println("Actual: " + actual.toString());
            System.out.println("Expected: " + expected.toString());
        }
    }

    @UnitTest(name = "Easy Position")
    public void easyPosition() {
        TTTPiece[] toWinEasyPosition = new TTTPiece[] {
                TTTPiece.X, TTTPiece.O, TTTPiece.X,
                TTTPiece.X, TTTPiece.E, TTTPiece.O,
                TTTPiece.E, TTTPiece.E, TTTPiece.O };
        TTTBoard testBoard1 = new TTTBoard(toWinEasyPosition, TTTPiece.X);
        Integer answer1 = Minimax.findBestMove(testBoard1, 8);
        assertEquality(answer1, 6);
    }
```

```
    @UnitTest(name = "Block Position")
    public void blockPosition() {
        TTTPiece[] toBlockPosition = new TTTPiece[] {
                TTTPiece.X, TTTPiece.E, TTTPiece.E,
                TTTPiece.E, TTTPiece.E, TTTPiece.O,
                TTTPiece.E, TTTPiece.X, TTTPiece.O };
        TTTBoard testBoard2 = new TTTBoard(toBlockPosition, TTTPiece.X);
        Integer answer2 = Minimax.findBestMove(testBoard2, 8);
        assertEquality(answer2, 2);
    }

    @UnitTest(name = "Hard Position")
    public void hardPosition() {
        TTTPiece[] toWinHardPosition = new TTTPiece[] {
                TTTPiece.X, TTTPiece.E, TTTPiece.E,
                TTTPiece.E, TTTPiece.E, TTTPiece.O,
                TTTPiece.O, TTTPiece.X, TTTPiece.E };
        TTTBoard testBoard3 = new TTTBoard(toWinHardPosition, TTTPiece.X);
        Integer answer3 = Minimax.findBestMove(testBoard3, 8);
        assertEquality(answer3, 1);
    }

    // Запустите все методы, отмеченные аннотацией UnitTest
    public void runAllTests() {
        for (Method method : this.getClass().getMethods()) {
            for (UnitTest annotation :
              method.getAnnotationsByType(UnitTest.class)) {
                System.out.println("Running Test " + annotation.name());
                try {
                    method.invoke(this);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                System.out.println("_____________________");
            }
        }
    }

    public static void main(String[] args) {
        new TTTMinimaxTests().runAllTests();
    }
}
```

Как я уже говорил, не рекомендуется вместо `JUnit` использовать собственную платформу модульного тестирования. Тем не менее это не так уж сложно благодаря возможностям отражения в Java. Каждый метод, представляющий тест, содержит специальную аннотацию под названием `UnitTest`, определенную в верхней части файла. Метод `runAllTests()` ищет все методы с этой аннотацией и запускает их вместе с некоторыми полезными распечатками. Метод `assertEquality()` проверяет, равны ли два элемента, и если нет, выводит их на печать. Хотя определение собственной инфраструктуры модульного тестирования может быть не очень

хорошей идеей, интересно посмотреть, как она будет работать. Чтобы вывести нашу структуру на следующий уровень, определим базовый класс, включающий методы `runAllTests()` и `assertEquality()`, которые могут быть расширены другими классами тестирования.

Для выполнения всех трех тестов запустите файл `TTTMinimaxTests.java`.

СОВЕТ

Для реализации минимакса нет необходимости писать много кода — данный алгоритм пригоден для гораздо большего количества игр, чем просто крестики-нолики. Если вы планируете внедрить минимакс для другой игры, важно настроиться на успех, создавая структуры данных, которые хорошо работают в сочетании с минимаксом, такие как класс Board. Распространенная ошибка тех, кто изучает минимакс, — использование изменяемой структуры данных, которая изменяется при рекурсивном вызове минимакса и затем не может быть возвращена в исходное состояние для последующих вызовов алгоритма.

8.2.4. Разработка ИИ для игры в крестики-нолики

Теперь, когда у нас есть все необходимые ингредиенты, можно легко сделать следующий шаг — разработать полностью искусственного противника, способного пройти всю игру в крестики-нолики. Вместо того чтобы оценивать тестовую позицию, ИИ будет оценивать только позицию, генерируемую на каждом ходе противника. В следующем фрагменте короткого кода ИИ играет в крестики-нолики против оппонента-человека, который делает первый ход (листинг 8.12).

Листинг 8.12. TicTacToe.java

```
package chapter8;

import java.util.Scanner;

public class TicTacToe {

    private TTTBoard board = new TTTBoard();
    private Scanner scanner = new Scanner(System.in);

    private Integer getPlayerMove() {
        Integer playerMove = -1;
        while (!board.getLegalMoves().contains(playerMove)) {
            System.out.println("Enter a legal square (0-8):");
            Integer play = scanner.nextInt();
            playerMove = play;
        }
        return playerMove;
    }

    private void runGame() {
```

```
        // главный цикл игры
        while (true) {
            Integer humanMove = getPlayerMove();
            board = board.move(humanMove);
            if (board.isWin()) {
                System.out.println("Human wins!");
                break;
            } else if (board.isDraw()) {
                System.out.println("Draw!");
                break;
            }
            Integer computerMove = Minimax.findBestMove(board, 9);
            System.out.println("Computer move is " + computerMove);
            board = board.move(computerMove);
            System.out.println(board);
            if (board.isWin()) {
                System.out.println("Computer wins!");
                break;
            } else if (board.isDraw()) {
                System.out.println("Draw!");
                break;
            }
        }
    }

    public static void main(String[] args) {
        new TicTacToe().runGame();
    }
}
```

Поскольку по умолчанию значение `maxDepth` для `findBestMove()` равно 9 (на самом деле 8), этот ИИ для игры в крестики-нолики всегда будет просматривать ходы до конца игры. (Максимальное количество ходов в крестиках-ноликах равно 9, а ИИ ходит вторым.) Поэтому ИИ всегда должен играть идеально. Идеальная игра — это игра, в которой оба противника каждый раз выбирают наилучший ход. Результат идеальной игры в крестики-нолики — ничья. Учитывая это, вы никогда не сможете выиграть у искусственного интеллекта. Если будете играть идеально, то сыграете вничью, если сделаете ошибку, то ИИ победит. Попробуйте сами. Вы не должны победить. Далее приведен пример запуска программы:

```
Enter a legal square (0-8):
4
Computer move is 0
O| |
-----
 |X|
-----
 | |

Enter a legal square (0-8):
2
```

```
Computer move is 6
O| |X
-----
 |X|
-----
O| |

Enter a legal square (0-8):
3
Computer move is 5
O| |X
-----
X|X|O
-----
O| |

Enter a legal square (0-8):
1
Computer move is 7
O|X|X
-----
X|X|O
-----
O|O|

Enter a legal square (0-8):
8
Draw!
```

8.3. CONNECT FOUR

В игре Connect Four[1] два игрока поочередно бросают разноцветные фишки в вертикальную сетку, состоящую из семи столбцов и шести рядов. Фишки падают вдоль сетки сверху вниз, пока не достигнут дна или не лягут на другую фишку. По сути, единственным выбором игрока на каждом ходе является то, в какой из семи столбцов бросить фишку. Игрок не может поместить фишку в заполненный столбец. Первый игрок, которому удастся выставить четыре фишки своего цвета в строку, столбец или по диагонали без разрывов, выигрывает. Если ни одному из игроков не удалось этого достичь, а сетка заполнена, считается, что игра закончилась вничью.

8.3.1. Подключите четыре игровых автомата

Игра Connect Four во многом похожа на крестики-нолики. Обе они ведутся на поле в виде сетки, и для выигрыша требуется, чтобы игрок расположил фишки

[1] Игра Connect Four («Четыре в ряд») — товарный знак компании Hasbro, Inc. Здесь она используется только в описательной форме.

в ряд. Но в Connect Four сетка больше, что предполагает гораздо больше способов выиграть, поэтому оценить каждую позицию значительно сложнее.

Представленный далее код (листинг 8.13) будет выглядеть в чем-то очень знакомым, но структуры данных и метод оценки сильно отличаются от использованных для игры в крестики-нолики. Обе игры реализованы как подклассы тех же базовых классов `Piece` и `Board`, с которыми мы познакомились в начале главы, что делает функцию `minimax()` пригодной для обеих игр.

Листинг 8.13. C4Piece.java

```
package chapter8;

public enum C4Piece implements Piece {
    B, R, E; // E — пустое поле

    @Override
    public C4Piece opposite() {
        switch (this) {
        case B:
            return C4Piece.R;
        case R:
            return C4Piece.B;
        default: // E, пустое поле
            return C4Piece.E;
        }
    }

    @Override
    public String toString() {
        switch (this) {
        case B:
            return "B";
        case R:
            return "R";
        default: // E, пустое поле
            return " ";
        }
    }
}
```

Класс `C4Piece` практически идентичен классу `TTTPiece`. У нас также будет удобный класс `C4Location` для отслеживания ячейки в сетке поля (пара «столбец/строка»). Connect Four — это игра, ориентированная на столбцы, поэтому мы реализуем весь код ее сетки в необычном формате: сначала столбец (листинг 8.14).

Затем переходим к классу `C4Board`. Он определяет несколько статических констант и один статический метод. Статический метод `generateSegments()` возвращает список массивов ячеек сетки (`C4Locations`). Каждый массив в списке содержит четыре ячейки сетки. Мы называем каждый из этих массивов из четырех точек

сетки *сегментом*. Если какой-либо сегмент на доске окажется окрашен в один цвет, это будет означать, что данный цвет выиграл.

Листинг 8.14. C4Location.java

```
package chapter8;

public final class C4Location {
    public final int column, row;

    public C4Location(int column, int row) {
        this.column = column;
        this.row = row;
    }
}
```

Возможность быстрого поиска всех сегментов на доске позволяет не только проверить, закончилась ли игра (кто-то выиграл), но и оценить позицию. Поэтому, как вы увидите в следующем фрагменте кода (листинг 8.15), мы кэшируем сегменты для доски заданного размера в виде переменной `SEGMENTS` в классе `C4Board`.

Листинг 8.15. C4Location.java

```
package chapter8;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class C4Board implements Board<Integer> {
    public static final int NUM_COLUMNS = 7;
    public static final int NUM_ROWS = 6;
    public static final int SEGMENT_LENGTH = 4;
    public static final ArrayList<C4Location[]> SEGMENTS = generateSegments();

    // генерируем все сегменты для данного поля,
    // этот статический метод запускается только один раз
    private static ArrayList<C4Location[]> generateSegments() {
        ArrayList<C4Location[]> segments = new ArrayList<>();
        // генерируем вертикальные сегменты
        for (int c = 0; c < NUM_COLUMNS; c++) {
            for (int r = 0; r <= NUM_ROWS - SEGMENT_LENGTH; r++) {
                C4Location[] bl = new C4Location[SEGMENT_LENGTH];
                for (int i = 0; i < SEGMENT_LENGTH; i++) {
                    bl[i] = new C4Location(c, r + i);
                }
                segments.add(bl);
            }
        }
        // генерируем горизонтальные сегменты
        for (int c = 0; c <= NUM_COLUMNS - SEGMENT_LENGTH; c++) {
            for (int r = 0; r < NUM_ROWS; r++) {
```

```
                C4Location[] bl = new C4Location[SEGMENT_LENGTH];
                for (int i = 0; i < SEGMENT_LENGTH; i++) {
                    bl[i] = new C4Location(c + i, r);
                }
                segments.add(bl);
            }
        }
        // генерируем сегменты диагонали из нижнего левого в верхний правый угол
        for (int c = 0; c <= NUM_COLUMNS - SEGMENT_LENGTH; c++) {
            for (int r = 0; r <= NUM_ROWS - SEGMENT_LENGTH; r++) {
                C4Location[] bl = new C4Location[SEGMENT_LENGTH];
                for (int i = 0; i < SEGMENT_LENGTH; i++) {
                    bl[i] = new C4Location(c + i, r + i);
                }
                segments.add(bl);
            }
        }
        // генерируем сегменты диагонали из нижнего правого в верхний левый угол
        for (int c = NUM_COLUMNS - SEGMENT_LENGTH; c >= 0; c--) {
            for (int r = SEGMENT_LENGTH - 1; r < NUM_ROWS; r++) {
                C4Location[] bl = new C4Location[SEGMENT_LENGTH];
                for (int i = 0; i < SEGMENT_LENGTH; i++) {
                    bl[i] = new C4Location(c + i, r - i);
                }
                segments.add(bl);
            }
        }

        return segments;
    }
```

Мы сохраняем текущую позицию `position` в двумерном массиве `C4Piece`. В большинстве случаев двумерные массивы индексируются, начиная с первой строки. Но представление о поле Connect Four как о группе из семи столбцов упрощает написание остальной части класса `C4Board`. Например, сопутствующий массив `columnCount` отслеживает, сколько частей одновременно находится в любом заданном столбце. Это упрощает создание допустимых ходов, поскольку каждый ход представляет собой выделение незаполненного столбца.

Следующие четыре метода очень похожи на их эквиваленты в игре в крестики-нолики (листинг 8.16).

Листинг 8.16. C4Board.java (продолжение)

```
    private C4Piece[][] position; // сначала столбец, затем строка
    private int[] columnCount; // количество фишек в каждом столбце
    private C4Piece turn;

    public C4Board() {
        // обратите внимание на то, что мы сначала создаем столбцы
        position = new C4Piece[NUM_COLUMNS][NUM_ROWS];
```

```
        for (C4Piece[] col : position) {
            Arrays.fill(col, C4Piece.E);
        }
        // ints по умолчанию инициализируются значением 0
        columnCount = new int[NUM_COLUMNS];
        turn = C4Piece.B; // черные ходят первыми
    }

    public C4Board(C4Piece[][] position, C4Piece turn) {
        this.position = position;
        columnCount = new int[NUM_COLUMNS];
        for (int c = 0; c < NUM_COLUMNS; c++) {
            int piecesInColumn = 0;
            for (int r = 0; r < NUM_ROWS; r++) {
                if (position[c][r] != C4Piece.E) {
                    piecesInColumn++;
                }
            }
            columnCount[c] = piecesInColumn;
        }
        this.turn = turn;
    }

    @Override
    public Piece getTurn() {
        return turn;
    }

    @Override
    public C4Board move(Integer location) {
        C4Piece[][] tempPosition = Arrays.copyOf(position, position.length);
        for (int col = 0; col < NUM_COLUMNS; col++) {
            tempPosition[col] = Arrays.copyOf(position[col],
     position[col].length);
        }
        tempPosition[location][columnCount[location]] = turn;
        return new C4Board(tempPosition, turn.opposite());
    }

    @Override
    public List<Integer> getLegalMoves() {
        List<Integer> legalMoves = new ArrayList<>();
        for (int i = 0; i < NUM_COLUMNS; i++) {
            if (columnCount[i] < NUM_ROWS) {
                legalMoves.add(i);
            }
        }
        return legalMoves;
    }
```

Вспомогательный метод `countSegment()` возвращает количество черных и красных фишек в определенном сегменте. За ним следует метод проверки выигрыша `isWin()`, который просматривает все сегменты на поле и определяет, была ли игра

выиграна, используя метод `countSegment()`, чтобы подсчитать, есть ли в каком-либо сегменте четыре фишки одного цвета (листинг 8.17).

Листинг 8.17. C4Board.java (продолжение)

```
    private int countSegment(C4Location[] segment, C4Piece color) {
        int count = 0;
        for (C4Location location : segment) {
            if (position[location.column][location.row] == color) {
                count++;
            }
        }
        return count;
    }

    @Override
    public boolean isWin() {
        for (C4Location[] segment : SEGMENTS) {
            int blackCount = countSegment(segment, C4Piece.B);
            int redCount = countSegment(segment, C4Piece.R);
            if (blackCount == SEGMENT_LENGTH || redCount == SEGMENT_LENGTH) {
                return true;
            }
        }
        return false;
    }
```

Как и `TTTBoard`, `C4Board` может использовать свойство `isDraw()` абстрактного базового класса `Board` без изменений.

Чтобы оценить позицию, мы по очереди оценим все представляющие ее сегменты, суммируем оценки и вернем результат. Сегмент с красными и черными фишками будет считаться бесполезным. Сегмент, имеющий два поля с фишками одного цвета и два пустых поля, получит 1 балл. Сегмент с тремя фишками одного цвета будет оценен в 100 баллов. Наконец, сегмент с четырьмя фишками одного цвета (победа) набирает 1 000 000 баллов. Если сегмент принадлежит противнику, то его очки вычитаются. Функция `evaluateSegment()` — вспомогательный метод, который оценивает сегмент с применением предыдущей формулы. Суммарная оценка всех сегментов методом `evaluateSegment()` выполняется с помощью `evaluate()` (листинг 8.18).

Листинг 8.18. C4Board.java (продолжение)

```
    private double evaluateSegment(C4Location[] segment, Piece player) {
        int blackCount = countSegment(segment, C4Piece.B);
        int redCount = countSegment(segment, C4Piece.R);
        if (redCount > 0 && blackCount > 0) {
            return 0.0; // смешанные сегменты нейтральны
        }
```

```
        int count = Math.max(blackCount, redCount);
        double score = 0.0;
        if (count == 2) {
            score = 1.0;
        } else if (count == 3) {
            score = 100.0;
        } else if (count == 4) {
            score = 1000000.0;
        }
        C4Piece color = (redCount > blackCount) ? C4Piece.R : C4Piece.B;
        if (color != player) {
            return -score;
        }
        return score;
    }

    @Override
    public double evaluate(Piece player) {
        double total = 0.0;
        for (C4Location[] segment : SEGMENTS) {
            total += evaluateSegment(segment, player);
        }
        return total;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int r = NUM_ROWS - 1; r >= 0; r--) {
            sb.append("|");
            for (int c = 0; c < NUM_COLUMNS; c++) {
                sb.append(position[c][r].toString());
                sb.append("|");
            }
            sb.append(System.lineSeparator());
        }
        return sb.toString();
    }
}
```

8.3.2. ИИ для Connect Four

Удивительно, но функции `minimax()` и `findBestMove()`, которые мы разработали для игры в крестики-нолики, можно без изменений использовать в реализации Connect Four. В следующем фрагменте кода (листинг 8.19) добавилась лишь пара изменений по сравнению с кодом ИИ для игры в крестики-нолики. Главное различие заключается в том, что значение `maxDepth` теперь равно 5. Это обеспечивает разумное время, отводимое компьютеру на обдумывание хода. Другими словами, наш ИИ для Connect Four рассматривает (оценивает) позиции не более чем на пять ходов вперед.

Листинг 8.19. ConnectFour.java

```
package chapter8;

import java.util.Scanner;

public class ConnectFour {

    private C4Board board = new C4Board();
    private Scanner scanner = new Scanner(System.in);

    private Integer getPlayerMove() {
        Integer playerMove = -1;
        while (!board.getLegalMoves().contains(playerMove)) {
            System.out.println("Enter a legal column (0-6):");
            Integer play = scanner.nextInt();
            playerMove = play;
        }
        return playerMove;
    }

    private void runGame() {
        // главный цикл игры
        while (true) {
            Integer humanMove = getPlayerMove();
            board = board.move(humanMove);
            if (board.isWin()) {
                System.out.println("Human wins!");
                break;
            } else if (board.isDraw()) {
                System.out.println("Draw!");
                break;
            }
            Integer computerMove = Minimax.findBestMove(board, 5);
            System.out.println("Computer move is " + computerMove);
            board = board.move(computerMove);
            System.out.println(board);
            if (board.isWin()) {
                System.out.println("Computer wins!");
                break;
            } else if (board.isDraw()) {
                System.out.println("Draw!");
                break;
            }
        }
    }

    public static void main(String[] args) {
        new ConnectFour().runGame();
    }
}
```

Попробуйте сыграть в Connect Four с этим ИИ. Вы заметите, что генерация компьютером хода занимает несколько секунд, в отличие от ИИ для игры

в крестики-нолики. Компьютер, вероятно, все еще будет выигрывать, и единственный способ победить его — тщательно продумывать свои ходы. Во всяком случае, ИИ не будет делать совершенно очевидных ошибок. Мы можем улучшить его игру, увеличив глубину поиска, но тогда время, затрачиваемое компьютером на каждый ход, будет экспоненциально увеличиваться. Далее приведены несколько ходов против ИИ:

```
Enter a legal column (0-6):
3
Computer move is 3
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |R| | | |
| | | |B| | | |

Enter a legal column (0-6):
4
Computer move is 5
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |R| | | |
| | | |B|B|R| |

Enter a legal column (0-6):
4
Computer move is 4
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |R| | |
| | | |R|B| | |
| | | |B|B|R| |
```

ПРИМЕЧАНИЕ

Знаете ли вы, что специалисты по информатике решили игру Connect Four? Решить игру — значит найти наилучший ход в любой позиции. Наилучший первый ход в Connect Four — поместить свою фишку в центральный столбец.

8.3.3. Улучшение минимакса с помощью альфа-бета-отсечения

Минимакс работает хорошо, но сейчас нам не нужен глубокий поиск. Существует небольшое расширение минимакса, известное как *альфа-бета-отсечение*, которое

позволяет улучшить глубину поиска, исключая те позиции, которые не приведут к улучшению по сравнению с уже найденными позициями. Магия достигается отслеживанием между рекурсивными вызовами минимакса двух значений, которые получили названия «альфа» и «бета». *Альфа* представляет собой оценку наилучшего максимизирующего хода, найденного до этого момента в дереве поиска, а *бета* — оценку наилучшего минимизирующего хода, найденного для противника. Если в какой-то момент `beta` окажется меньше `alpha` или равной ей, то исследовать эту ветвь поиска далее нет смысла, поскольку уже найденный ход — лучший или как минимум не хуже тех, что могут быть найдены далее по этой ветви. Такая эвристика значительно сокращает пространство поиска.

Далее показана функция `alphabeta()`, работающая по только что описанному алгоритму (листинг 8.20). Ее следует поместить в уже существующий файл `Minimax.java`.

Листинг 8.20. Minimax.java (продолжение)

```
// Помощник, который устанавливает alpha и beta для первого вызова
public static <Move> double alphabeta(Board<Move> board,
  Boolean maximizing, Piece originalPlayer, int maxDepth) {
    return alphabeta(board, maximizing, originalPlayer, maxDepth,
      Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY);
}

// Оценивает поле b
private static <Move> double alphabeta(Board<Move> board,
  Boolean maximizing, Piece originalPlayer, int maxDepth,
        double alpha,
        double beta) {
    // Базовый случай — достигнута финальная
    // позиция или максимальная глубина поиска
    if (board.isWin() || board.isDraw() || maxDepth == 0) {
        return board.evaluate(originalPlayer);
    }

    // Рекурсивный случай — максимизируйте свою
    // выгоду или минимизируйте выгоду противника
    if (maximizing) {
        for (Move m : board.getLegalMoves()) {
            alpha = Math.max(alpha, alphabeta(board.move(m), false,
              originalPlayer, maxDepth - 1, alpha, beta));
            if (beta <= alpha) { // отсечение
                break;
            }
        }
        return alpha;
    } else { // минимизация
        for (Move m : board.getLegalMoves()) {
            beta = Math.min(beta, alphabeta(board.move(m), true,
              originalPlayer, maxDepth - 1, alpha, beta));
            if (beta <= alpha) { // отсечение
```

```
                    break;
                }
            }
            return beta;
        }
    }
```

Теперь мы можем внести два небольших изменения, чтобы воспользоваться новой функцией. Измените `findBestMove()` в `Minimax.java`, задействуя `alphabeta()` вместо `minimax()`, и замените глубину поиска в `ConnectFour.java` с 5 на 7. Теперь средний игрок в Connect Four больше не сможет выиграть у ИИ. На моем компьютере, применяя `minimax()` с глубиной 7, ИИ для Connect Four тратил около трех минут на ход, а с `alphabeta()` при той же глубине ход занимал около 20 секунд. Одна десятая затраченного времени — невероятное улучшение!

8.4. ДРУГИЕ УЛУЧШЕНИЯ МИНИМАКСА

В этой главе представлены глубоко изученные алгоритмы, и за последние годы для них было найдено много улучшений. Некоторые из них относятся к конкретной игре, например «битовая доска» в шахматах, которая позволяет сократить время на генерирование правильных ходов, но большинство — это общие методы, которые можно использовать для любой игры.

Один из распространенных методов — это *итеративное углубление*. В этом случае функция поиска выполняется сначала с максимальной глубиной, равной 1. Затем она запускается с максимальной глубиной 2, 3 и т. д. По истечении указанного времени поиск прекращается и возвращается результат поиска для последней выбранной глубины.

Примеры, приведенные в этой главе, были жестко закодированы до определенной глубины. Это нормально, если игра идет без применения часов и ограничений по времени или если нам все равно, сколько времени потребуется компьютеру на «подумать». Итеративное углубление позволяет задать для ИИ фиксированное время поиска следующего хода вместо фиксированной глубины поиска с переменным количеством времени на выполнение хода.

Другое потенциальное улучшение — *поиск покоя*. При использовании этой технологии минимаксное дерево поиска дополнительно расширяется в тех направлениях, которые вызывают наибольшие изменения в положении (например, в шахматах), а не в тех, которые имеют относительно тихие позиции. Таким образом, в идеале алгоритм поиска не будет тратить время на вычисление скучных позиций, которые вряд ли принесут игроку значительное преимущество.

Два наилучших способа улучшить минимаксный поиск — это пройти бóльшую глубину за отведенное время или улучшить функцию, применяемую для оценки

позиции. Перебор большего количества позиций за неизменное время позволяет тратить меньше времени на каждую позицию. Это может быть связано с повышением эффективности кода или использованием более быстрого аппаратного обеспечения, но может происходить и за счет второго способа — улучшения оценки каждой позиции. Применение большего количества параметров или эвристики для оценки позиции может занять больше времени, но в итоге привести к созданию лучшего механизма, которому требуется меньшая глубина поиска, чтобы найти хороший ход.

Некоторые функции оценки, используемые для поиска минимакса с альфа-бета-отсечением в шахматах, имеют десятки эвристик. Для настройки эвристик даже применялись генетические алгоритмы. Сколько стоит ход со взятием коня в шахматах? Равно ли данное значение стоимости взятия слона? Эти эвристики могут быть тем секретным соусом, который отличает отличный шахматный движок от хорошего.

8.5. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

Минимакс в сочетании с дополнительными расширениями, такими как альфа-бета-отсечение, — это основа большинства современных шахматных движков. Этот алгоритм успешно применяется в разнообразных стратегических играх. В сущности, большинство искусственных противников в компьютерных играх основаны на той или иной форме минимакса.

Минимакс (с расширениями, такими как альфа-бета-отсечение) оказался настолько эффективным в шахматах, что привел к громкому поражению чемпиона мира по шахматам среди людей Гарри Каспарова в 1997 году от шахматного компьютера Deep Blue, созданного компанией IBM. Этот долгожданный матч стал событием, которое в корне изменило ситуацию. Прежде шахматы считались самой высокоинтеллектуальной игрой. Тот факт, что компьютер смог превзойти в ней человека, показал, что к искусственному интеллекту следует относиться серьезно.

Спустя два десятилетия подавляющее большинство шахматных движков попрежнему основаны на той или иной версии минимакса. Сегодняшние минимаксные шахматные движки намного превосходят возможности лучших шахматистов мира. Новые методы машинного обучения бросают вызов чисто шахматным движкам, построенным на основе минимакса (с расширениями), но им еще предстоит доказать свое превосходство в шахматах.

Чем выше коэффициент ветвления в игре, тем меньше эффективность минимакса. Коэффициент ветвления — это среднее количество потенциальных ходов в данной позиции игры. Вот почему последние достижения в компьютерной реализации игры го потребовали изучения других областей, таких как машинное обучение.

Недавно искусственный интеллект для го, построенный на основе машинного обучения, победил лучшего игрока. Коэффициент ветвления (и, следовательно, пространство поиска) для го оказывается просто непосильным для алгоритмов, основанных на минимаксном алгоритме, которые пытаются генерировать деревья, содержащие будущие позиции игры. Но го — это скорее исключение, чем правило. Большинство традиционных настольных игр — шашки, шахматы, Connect Four, скрэбл и т. п. — имеют довольно малое пространство поиска, так что методы, основанные на минимаксных алгоритмах, в них хорошо работают.

Если вам нужно реализовать нового искусственного противника в настольной игре или даже построить ИИ для пошаговой компьютерной игры, то минимакс, вероятно, первый алгоритм, к которому следует обратиться. Минимакс можно использовать для экономических и политических симуляций, а также для экспериментов в области теории игр. Альфа-бета-отсечение должно работать для любой формы минимакса.

8.6. УПРАЖНЕНИЯ

1. Напишите модульные тесты для игры в крестики-нолики, чтобы убедиться в правильной работе свойств `getLegalMoves()`, `isWin()` и `isDraw()`.
2. Напишите модульные тесты, проверяющие минимакс в Connect Four.
3. Код в файлах `TicTacToe.java` и `ConnectFour.java` практически идентичен. Разделите его на два метода, которые можно использовать для любой игры.
4. Измените файл `ConnectFour.java` так, чтобы компьютер играл против самого себя. Какой игрок выиграет — первый или второй? Это каждый раз один и тот же игрок?
5. Можете ли вы путем профилирования существующего кода или иным образом оптимизировать метод оценки в `ConnectFour.java`, чтобы увеличить глубину поиска за то же время?
6. Используя функцию `alphabeta()`, разработанную в этой главе, в сочетании с библиотекой Java, создайте функции построения корректных шахматных ходов и поддержки состояния шахматной игры для разработки шахматного ИИ.

9 Другие задачи

В этой книге мы рассмотрели множество методов решения проблем, связанных с современными задачами разработки программного обеспечения. Все технологии изучили на примере известных задач информатики. Но не все задачи соответствуют темам предыдущих глав. В данной главе собраны известные задачи, которые не вполне вписываются в другие главы. Можете считать их бонусом: более интересные задачи с меньшим количеством строительных лесов вокруг них.

9.1. ЗАДАЧА О РЮКЗАКЕ

Это задача оптимизации, которая касается типичной потребности, часто возникающей при вычислениях, — найти наилучший вариант использования ограниченных ресурсов при ограниченном наборе вариантов — и превращает ее в забавную историю. Вор входит в дом с намерением обчистить его. У него есть рюкзак, и он может вынести из дома только то, что туда поместится. Как узнать, что стоит положить в рюкзак? Задача проиллюстрирована на рис. 9.1.

Если бы вор имел возможность взять любую долю какой-либо вещи, то он мог бы просто разделить ценность каждого предмета на его вес, чтобы определить наиболее ценные из них, способные поместиться в доступную емкость. Но мы сделаем сценарий более реалистичным: допустим, вор не может взять часть предмета, например 2,5 телевизора. Вместо этого придумаем способ решения задачи в так называемом формате 0/1, в котором применяется другое правило: вор может или взять предмет целиком, или же не брать его вовсе.

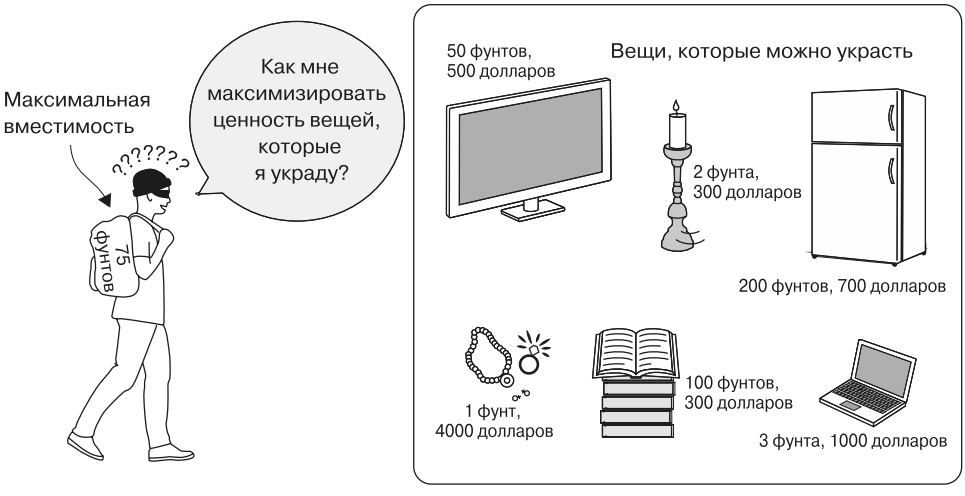


Рис. 9.1. Взломщику необходимо решить, какие предметы украсть, потому что вместимость рюкзака ограниченна

Прежде всего определим класс `Item` для хранения вещей (листинг 9.1).

Листинг 9.1. Knapsack.java

```
package chapter9;

import java.util.ArrayList;
import java.util.List;

public final class Knapsack {

    public static final class Item {
        public final String name;
        public final int weight;
        public final double value;

        public Item(String name, int weight, double value) {
            this.name = name;
            this.weight = weight;
            this.value = value;
        }
    }
```

Если бы мы попытались решить эту задачу методом грубой силы, нам пришлось бы рассмотреть каждую комбинацию предметов, которые можно положить в рюкзак. Те, у кого есть склонности к математике, называют это *множеством всех подмножеств*, и множество всех подмножеств для данного множества (в нашем случае множества предметов) состоит из 2^N возможных подмножеств, где

N — количество предметов. Поэтому нам необходимо проанализировать 2^N комбинаций (O(2^N)). Для небольшого количества предметов это нормально, но для большого — не годится. Следует избегать любого подхода, при котором задача решается за экспоненциальное число шагов.

Здесь мы будем использовать технологию, известную как *динамическое программирование*, которое по своей концепции похоже на мемоизацию (см. главу 1). Вместо того чтобы решать задачу методом грубой силы, при динамическом программировании решаются подзадачи, которые составляют большую задачу, их результаты сохраняются, а затем применяются для решения более крупной задачи.

Поскольку вместимость рюкзака рассматривается в отдельных шагах, задачу можно решить методом динамического программирования.

Например, чтобы решить задачу для рюкзака вместимостью 3 фунта и трех предметов, мы можем сначала решить задачу для вместимости 1 фунт и одного возможного предмета, потом для вместимости 2 фунта и одного возможного предмета и вместимости 3 фунта и одного возможного предмета.

Затем можем использовать полученные результаты, чтобы решить задачу для вместимости 1 фунт и двух возможных предметов, вместимости 2 фунта и двух возможных предметов, вместимости 3 фунта и двух возможных предметов. И наконец, решить задачу для всех трех возможных предметов.

На протяжении пути мы станем заполнять таблицу, которая будет предлагать нам наилучшее возможное решение для каждой комбинации предметов и вместимости рюкзака. Функция сначала заполнит таблицу, а затем на ее основе выберет решение[1].

Листинг 9.2. Knapsack.java (продолжение)

```
public static List<Item> knapsack(List<Item> items, int maxCapacity) {
    // построение таблицы динамического программирования
    double[][] table = new double[items.size() + 1][maxCapacity + 1];
    for (int i = 0; i < items.size(); i++) {
        Item item = items.get(i);
        for (int capacity = 1; capacity <= maxCapacity; capacity++) {
            double prevItemValue = table[i][capacity];
            if (capacity >= item.weight) { // предмет помещается в рюкзак
```

[1] Чтобы создать это решение, я изучил несколько источников, наиболее авторитетным из которых была книга: *Sedgewick R., Wayne K.* Algorithms — 2nd ed. — Addison-Wesley, 1988. Я рассмотрел несколько примеров задачи о рюкзаке типа 0/1 в «Розеттском коде», в первую очередь решение для динамического программирования на Python (http://mng.bz/kx8C), к которому в значительной степени относится данная функция, по сравнению с версией из книги о Swift (код был переведен с Python на Swift и затем снова на Python).

```
                double valueFreeingWeightForItem = table[i][capacity –
                    item.weight];
                // только если этот предмет ценнее предыдущего
                table[i + 1][capacity] = Math.max(valueFreeingWeightForItem +
                    item.value, prevItemValue);
            } else { // для этого предмета нет места
                table[i + 1][capacity] = prevItemValue;
            }
        }
    }
    // найти решение в таблице
    List<Item> solution = new ArrayList<>();
    int capacity = maxCapacity;
    for (int i = items.size(); i > 0; i--) { // идем в обратном направлении
        // этот предмет уже выбран?
        if (table[i – 1][capacity] != table[i][capacity]) {
            solution.add(items.get(i – 1));
            // если предмет выбран, то вычитаем его вес
            capacity -= items.get(i – 1).weight;
        }
    }
    return solution;
}
```

Внутренний цикл в первой части этой функции будет выполняться NC раз, где N — количество предметов, а C — максимальная вместимость рюкзака. Следовательно, алгоритм выполняется за время $O(NC)$, что для большого количества предметов значительно лучше, чем метод грубой силы. Например, для 11 описанных далее предметов с помощью алгоритма грубой силы мы должны были рассмотреть 2^{11}, или 2048 комбинаций. Представленная ранее функция динамического программирования будет выполняться 825 раз ($11 \cdot 75$), поскольку максимальная вместимость рюкзака составляет в данном случае 75 произвольных единиц. Эта разница будет расти экспоненциально по мере увеличения количества предметов.

Давайте посмотрим на решение в действии (листинг 9.3).

Листинг 9.3. Knapsack.java (продолжение)

```
public static void main(String[] args) {
    List<Item> items = new ArrayList<>();
    items.add(new Item("television", 50, 500));
    items.add(new Item("candlesticks", 2, 300));
    items.add(new Item("stereo", 35, 400));
    items.add(new Item("laptop", 3, 1000));
    items.add(new Item("food", 15, 50));
    items.add(new Item("clothing", 20, 800));
    items.add(new Item("jewelry", 1, 4000));
    items.add(new Item("books", 100, 300));
    items.add(new Item("printer", 18, 30));
    items.add(new Item("refrigerator", 200, 700));
```

```
        items.add(new Item("painting", 10, 1000));
        List<Item> toSteal = knapsack(items, 75);
        System.out.println("The best items for the thief to steal are:");
        System.out.printf("%-15.15s %-15.15s %-15.15s%n", "Name", "Weight",
            "Value");
        for (Item item : toSteal) {
            System.out.printf("%-15.15s %-15.15s %-15.15s%n",
              item.name, item.weight, item.value);
        }
    }
}
```

Если вы проверите результаты, выведенные в консоль, то увидите, что оптимальными предметами, которые можно взять, являются картина, украшения, одежда, ноутбук, стереосистема и подсвечники. Вот пример выходных данных, показывающий наиболее ценные для вора вещи с учетом того, что вместимость рюкзака ограниченна:

```
The best items for the thief to steal are:
Name            Weight          Value
painting        10              1000.0
jewelry         1               4000.0
clothing        20              800.0
laptop          3               1000.0
stereo          35              400.0
candlesticks    2               300.0
```

Чтобы лучше понять, как это все работает, рассмотрим некоторые особенности функции `knapsack()`:

```
for (int i = 0; i < items.size(); i++) {
    Item item = items.get(i);
    for (int capacity = 1; capacity <= maxCapacity; capacity++) {
```

Для каждого возможного количества предметов мы перебираем в цикле все варианты вместимости рюкзака вплоть до максимальной. Обратите внимание на то, что я говорю «каждое возможное количество предметов», а не «каждый предмет». Когда *i* равно 2, это означает не просто предмет номер 2, а все возможные комбинации из первых двух предметов для каждой исследуемой вместимости. `item` — это следующий из рассматриваемых предметов, который можно украсть:

```
double prevItemValue = table[i][capacity];
if (capacity >= item.weight) { // предмет помещается в рюкзак
```

`prevItemValue` — это значение последней комбинации предметов для текущей исследуемой вместимости рюкзака. Для каждой возможной комбинации предметов мы проверяем, можно ли добавить в рюкзак еще один новый элемент.

Если предмет весит больше, чем позволяет рассматриваемая вместимость рюкзака, то мы просто копируем значение для последней комбинации предметов, которую рассматривали для данной вместимости:

```
else { // для этого предмета нет места
    table[i + 1][capacity] = prevItemValue;
}
```

В противном случае проверяем, приведет ли добавление нового предмета к более высокой стоимости всего украденного, чем последнее сочетание предметов для той вместимости рюкзака, которую мы рассматриваем. Для этого мы прибавляем стоимость предмета к значению, уже вычисленному в таблице для предыдущей комбинации предметов. Если это значение больше, чем значение для последней комбинации предметов для текущей вместимости, то вставляем его в таблицу, если же нет — вставляем последнее значение:

```
double valueFreeingWeightForItem = table[i][capacity – item.weight];
// только если этот предмет более ценный, чем предыдущий
table[i + 1][capacity] = Math.max(valueFreeingWeightForItem + item.value,
  prevItemValue);
```

На этом составление таблицы завершается. Однако для того, чтобы действительно узнать, какие предметы будут в решении, нужно проделать обратную работу для максимальной вместимости и результирующей исследуемой комбинации предметов:

```
for (int i = items.size(); i > 0; i--) { // идем в обратном направлении
    // был ли предмет использован?
    if (table[i – 1][capacity] != table[i][capacity]) {
```

Мы начинаем с конца и перебираем таблицу справа налево, на каждом этапе проверяя, было ли изменено значение, вставленное в нее. Если изменено, значит, мы добавили новый предмет, который рассматривался в данной комбинации, потому что эта комбинация оказалась более ценной, чем предыдущая. Поэтому мы добавляем этот предмет в решение. Кроме того, по мере перемещения по таблице вместимость рюкзака уменьшается на вес предмета:

```
solution.add(items.get(i – 1));
// если предмет выбран, то вычитаем его вес
capacity -= items.get(i – 1).weight;
```

ПРИМЕЧАНИЕ

И при построении таблицы, и при поиске решения вы могли заметить увеличение и уменьшение итераторов и размера таблицы на единицу. Это сделано для удобства с точки зрения программирования. Подумайте, как эта задача строится снизу вверх. В начале решения мы имеем дело с рюкзаком нулевой вместимости. Если вы поднимете снизу вверх по таблице, то поймете, зачем нужны дополнительные строка и столбец.

Вы все еще в замешательстве? Таблица 9.1 — это то, что строит функция `knapsack()`. Для предыдущей задачи это была бы довольно большая таблица, так что вместо этого рассмотрим таблицу для рюкзака вместимостью 3 фунта и трех предметов: спичек (1 фунт), фонарика (2 фунта) и книги (1 фунт). Предположим, что эти предметы оцениваются в 5, 10 и 15 долларов соответственно.

Таблица 9.1. Пример задачи о рюкзаке для трех предметов

Предмет	**Вместимость рюкзака**			
	0 фунтов	**1 фунт**	**2 фунта**	**3 фунта**
Спички (1 фунт, 5 долларов)	0	5	5	5
Фонарик (2 фунта, 10 долларов)	0	5	10	15
Книга (1 фунт, 15 долларов)	0	15	20	25

Если просматривать таблицу слева направо, то вес предметов, которые вы пытаетесь уместить в рюкзаке, увеличивается. Если просматривать таблицу сверху вниз, то увеличивается количество предметов, которые вы пытаетесь поместить в рюкзак. В первом ряду вы пробуете поместить в рюкзак только спички. Во втором ряду выбираете наиболее ценную комбинацию из спичек и фонарика, которая может поместиться в рюкзак. В третьем ряду находите самую ценную комбинацию из всех трех предметов.

В качестве упражнения, которое поможет вам лучше понять задачу, попробуйте заполнить пустую версию этой таблицы самостоятельно, используя алгоритм, описанный в функции `knapsack()`, и эти же три предмета. Затем примените алгоритм, представленный в конце функции, чтобы выбрать из таблицы правильные предметы. Эта таблица соответствует переменной `table` в функции.

9.2. ЗАДАЧА КОММИВОЯЖЕРА

Проблема коммивояжера — классическая и одна из наиболее широко обсуждаемых задач во всей теории вычислений. Коммивояжер должен посетить все города на карте ровно один раз и вернуться в конце путешествия в город, с которого начался маршрут. Между любыми двумя городами существует прямая дорога, и коммивояжер может посещать города в любом порядке. Каков самый короткий маршрут?

Эту задачу можно представить как графовую (см. главу 4), в которой города — это вершины, а дороги между ними — ребра. Вашим первым побуждением может быть желание найти минимальное связующее дерево, как описано в главе 4. К сожалению, решить задачу коммивояжера не так просто. Минимальное связующее дерево — это кратчайший путь для соединения всех городов, но оно не обеспечивает кратчайшего пути для посещения всех городов ровно один раз.

Несмотря на то что поставленная задача кажется довольно простой, не существует алгоритма, который мог бы быстро решить ее для произвольного числа городов. Что я подразумеваю, говоря «быстро»? Я имею в виду, что есть проблема: перед нами *НП-трудная задача*. НП-трудная (недетерминированная полиномиальная трудная) задача — это задача, для которой не существует алгоритма полиномиального времени. (Требуемое время — это полиномиальная функция от размера входных данных.) По мере увеличения числа городов, которые должен посетить коммивояжер, сложность задачи растет исключительно быстро. Решить задачу для двадцати городов намного труднее, чем для десяти. Решить задачу идеально (оптимально) для нескольких миллионов городов за разумное время невозможно (насколько нам известно).

ПРИМЕЧАНИЕ

Наивный подход к решению задачи коммивояжера имеет сложность O(n!). Почему это так, говорится в подразделе 9.2.2. Перед тем как читать его, рекомендую прочесть подраздел 9.2.1, потому что после попытки реализации наивного решения задачи ее сложность станет очевидной.

9.2.1. Наивный подход

Наивный подход к решению задачи — просто перепробовать все возможные перестановки городов. Попробуем использовать наивный подход, чтобы проиллюстрировать сложность задачи и непригодность такого решения для применения в более крупных масштабах.

Тестовые данные

В нашей версии задачи коммивояжер заинтересован в посещении пяти крупных городов штата Вермонт. Мы не будем указывать начальный (следовательно, конечный) город. На рис. 9.2 показаны пять городов и расстояния между ними. Обратите внимание на то, что для построения маршрута указано расстояние между каждой парой городов.

Возможно, вам уже встречались таблицы расстояний. В них легко найти расстояние между любыми двумя городами. В табл. 9.2 перечислены расстояния для пяти городов, используемых в задаче.

Таблица 9.2. Расстояния между городами штата Вермонт

Город	Ратленд	Берлингтон	Уайт-Ривер	Беннингтон	Браттлборо
Ратленд	0	67	46	55	75
Берлингтон	67	0	91	122	153
Уайт-Ривер	46	91	0	98	65
Беннингтон	55	122	98	0	40
Браттлборо	75	153	65	40	0

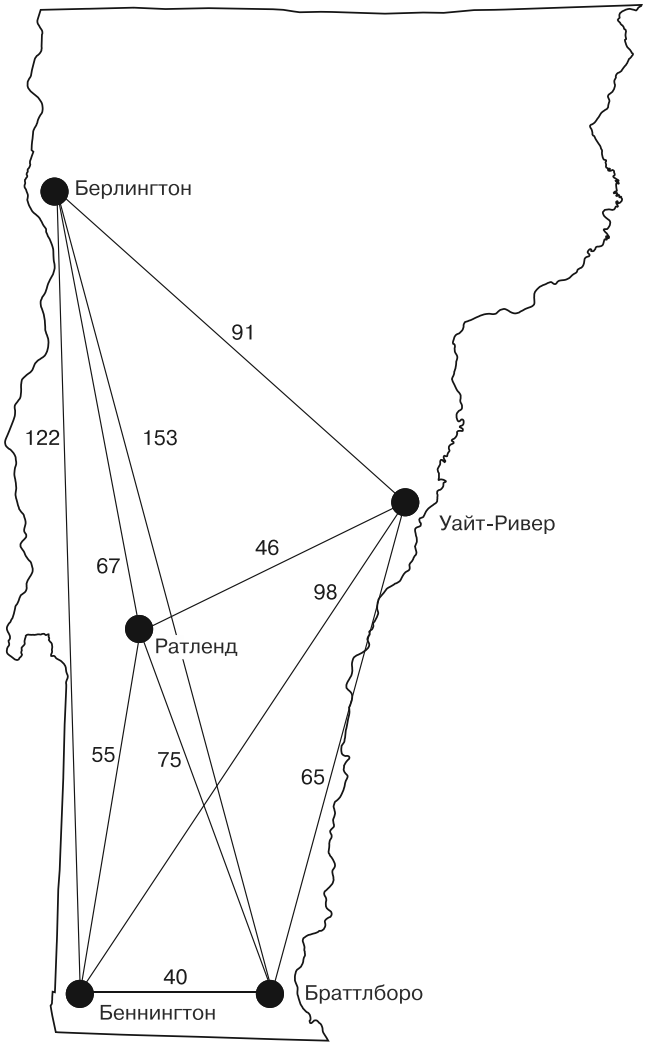


Рис. 9.2. Пять городов штата Вермонт и расстояния между ними

Для решения задачи нужно кодифицировать города и расстояния между ними. Чтобы облегчить поиск расстояний между городами, мы будем использовать словарь словарей с множеством внешних ключей, представляющих первый город пары, и множеством внутренних ключей, представляющих ее второй город. Это будет тип `Map<String, Map<String, Integer>>`, и он будет допускать поиск типа `vtDistances.get("Rutland").get("Burlington")`, который должен возвращать `67` (листинг 9.4). Мы применим карту `vtDistances`, когда на самом деле решим

задачу для Вермонта, но сначала давайте выполним некоторые настройки. Класс содержит карту и служебный метод, который мы будем использовать позже для обмена элементов в массиве.

Листинг 9.4. TSP.java

```
package chapter9;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Map;

public class TSP {
    private final Map<String, Map<String, Integer>> distances;

    public TSP(Map<String, Map<String, Integer>> distances) {
        this.distances = distances;
    }

    public static <T> void swap(T[] array, int first, int second) {
        T temp = array[first];
        array[first] = array[second];
        array[second] = temp;
    }
```

Перебор всех вариантов

Наивный подход к решению задачи коммивояжера требует сгенерировать все возможные варианты перестановок городов. Существует множество алгоритмов генерации перестановок, они довольно просты, так что вы почти наверняка сможете придумать такой алгоритм самостоятельно.

Один из типичных подходов — поиск с возвратом. Впервые он встретился нам в главе 3 в контексте решения задачи с ограничениями. При решении такой задачи поиск с возвратом используется после того, как найдено частичное решение, которое не удовлетворяет ограничениям задачи. В таком случае вы возвращаетесь к более раннему состоянию и продолжаете поиск по пути, отличному от того, который привел к частично неверному решению.

Найти все перестановки элементов списка (в частности, наших городов) можно также с помощью поиска с возвратом. Чтобы после перестановки элементов перейти к последующим перестановкам, можно вернуться к состоянию, предшествовавшему тому, при котором была выполнена перестановка, и выбрать другую комбинацию для перехода по другому пути (листинг 9.5).

Данная рекурсивная функция отмечена как «помощник», так как на самом деле она будет вызываться другой функцией, которая принимает меньше аргументов. Параметры `allPermutationsHelper()` — это начальная перестановка, с которой

мы работаем, перестановки, сгенерированные на данный момент, и количество оставшихся элементов для обмена.

Листинг 9.5. TSP.java (продолжение)

```
private static <T> void allPermutationsHelper(T[] permutation, List<T[]>
 permutations, int n) {
    // Базовый случай – мы нашли новую перестановку, добавляем
    // ее и возвращаем
    if (n <= 0) {
        permutations.add(permutation);
        return;
    }
    // Рекурсивный случай – найдем больше перестановок, поменяв местами
    T[] tempPermutation = Arrays.copyOf(permutation, permutation.length);
    for (int i = 0; i < n; i++) {
        swap(tempPermutation, i, n – 1); // переместим элемент из i в конец
        // переместим все остальное, удерживая конечную константу
        allPermutationsHelper(tempPermutation, permutations, n – 1);
        swap(tempPermutation, i, n – 1); // возврат
    }
}
```

Распространенный шаблон для рекурсивных функций, которым необходимо сохранять несколько элементов состояния между вызовами, — наличие отдельной функции, обращенной наружу, с меньшим количеством параметров и которую проще использовать. `allPermutations()` — это более простая функция (листинг 9.6).

Листинг 9.6. TSP.java (продолжение)

```
private static <T> List<T[]> permutations(T[] original) {
    List<T[]> permutations = new ArrayList<>();
    allPermutationsHelper(original, permutations, original.length);
    return permutations;
}
```

Функция `allPermutations()` принимает только один аргумент — массив, для которого должны быть сгенерированы перестановки. Чтобы найти эти перестановки, функция `allPermutations()` вызывает функцию `allPermutationsHelper()`. Это избавляет пользователя функции `allPermutations()` от необходимости предоставлять перестановки параметров и значение `n` в функции `allPermutationsHelper()`.

Подход с возвратом к поиску всех представленных здесь перестановок довольно эффективен. Для поиска каждой перестановки требуется всего две перестановки в массиве. Однако можно найти все перестановки массива с помощью всего одной перестановки. Один из эффективных алгоритмов, выполняющих эту задачу, — алгоритм Хипа (не путать со структурой данных heap (куча), в данном случае Хип — это имя создателя алгоритма). Эта разница в эффективности может быть важна для очень больших наборов данных (это не то, с чем мы здесь имеем дело).

Поиск методом грубой силы

Теперь мы можем сгенерировать все перестановки для списка городов, но это не совсем то же самое, что путь для задачи коммивояжера. Напомню, что в задаче о коммивояжере продавец должен в конце вернуться в тот город, с которого начал. Мы можем легко добавить расстояние от последнего города, в котором побывал продавец, до города, который он посетил первым, когда вычислим, какой путь кратчайший. Теперь мы готовы проверить все сгенерированные пути. Поиск методом грубой силы просматривает каждый путь в списке и, используя расстояние между двумя городами в таблице поиска (`distances`), вычисляет суммарное расстояние для каждого пути. Функция выводит кратчайший путь и суммарное расстояние для него (листинг 9.7).

Листинг 9.7. TSP.java (продолжение)

```
    public int pathDistance(String[] path) {
        String last = path[0];
        int distance = 0;
        for (String next : Arrays.copyOfRange(path, 1, path.length)) {
            distance += distances.get(last).get(next);
            // расстояние, чтобы вернуться из последнего города в первый
            last = next;
        }
        return distance;
    }

    public String[] findShortestPath() {
        String[] cities = distances.keySet().toArray(String[]::new);
        List<String[]> paths = permutations(cities);
        String[] shortestPath = null;
        int minDistance = Integer.MAX_VALUE; // произвольное большое число
        for (String[] path : paths) {
            int distance = pathDistance(path);
            // необходимо добавить расстояние от последнего города до первого
            distance += distances.get(path[path.length - 1]).get(path[0]);
            if (distance < minDistance) {
                minDistance = distance;
                shortestPath = path;
            }
        }
        // добавим первый город в конец и вернемся
        shortestPath = Arrays.copyOf(shortestPath, shortestPath.length + 1);
        shortestPath[shortestPath.length - 1] = shortestPath[0];
        return shortestPath;
    }

    public static void main(String[] args) {
        Map<String, Map<String, Integer>> vtDistances = Map.of(
                "Rutland", Map.of(
                        "Burlington", 67,
                        "White River Junction", 46,
```

```
                "Bennington", 55,
                "Brattleboro", 75),
            "Burlington", Map.of(
                "Rutland", 67,
                "White River Junction", 91,
                "Bennington", 122,
                "Brattleboro", 153),
            "White River Junction", Map.of(
                "Rutland", 46,
                "Burlington", 91,
                "Bennington", 98,
                "Brattleboro", 65),
            "Bennington", Map.of(
                "Rutland", 55,
                "Burlington", 122,
                "White River Junction", 98,
                "Brattleboro", 40),
            "Brattleboro", Map.of(
                "Rutland", 75,
                "Burlington", 153,
                "White River Junction", 65,
                "Bennington", 40));
        TSP tsp = new TSP(vtDistances);
        String[] shortestPath = tsp.findShortestPath();
        int distance = tsp.pathDistance(shortestPath);
        System.out.println("The shortest path is " +
          Arrays.toString(shortestPath) + " in " + distance + " miles.");
    }
}
```

Наконец, мы можем перебрать методом грубой силы все города Вермонта и найти кратчайший путь через заданные пять городов. Результат работы программы должен выглядеть примерно так, а лучший путь показан на рис. 9.3.

9.2.2. Переходим на следующий уровень

У задачи коммивояжера нет простого решения. По мере увеличения количества городов наивный подход быстро становится нереальным. Количество генерируемых перестановок равно факториалу от n ($n!$), где n — количество городов в задаче. Если бы мы добавили еще один город и их стало не пять, а шесть, то число оцененных маршрутов увеличилось бы в шесть раз. А при добавлении еще одного города решить задачу стало бы в семь раз сложнее. Этот подход не масштабируется!

В реальном мире наивный подход к задаче коммивояжера используется крайне редко. Большинство алгоритмов для различных вариаций этой задачи с большим количеством городов — приближенные. Они пытаются получить решение, близкое к оптимальному. Оно может находиться в пределах небольшой заранее известной полосы, к которой принадлежит идеальное решение (возможно, такие решения будут меньше чем на 5 % менее эффективными, чем идеальное).

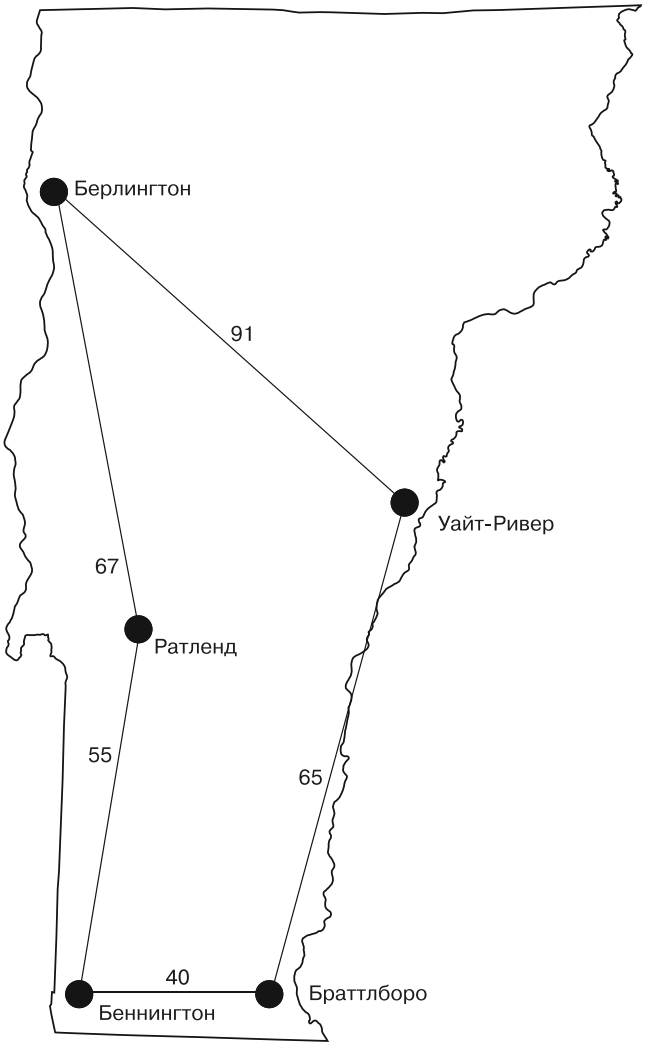


Рис. 9.3. Кратчайший путь для коммивояжера, позволяющий посетить все пять городов в штате Вермонт

Для решения задачи коммивояжера на больших наборах данных были задействованы два метода, уже описанные в этой книге. Первый — динамическое программирование, которое мы применили в этой главе, рассматривая задачу о рюкзаке. Второй — генетические алгоритмы, описанные в главе 5. Было опубликовано множество журнальных статей, в которых генетические алгоритмы называют почти оптимальным решением задачи коммивояжера с большим количеством городов.

9.3. МНЕМОНИКА ДЛЯ ТЕЛЕФОННЫХ НОМЕРОВ

До того как появились смартфоны со встроенной адресной книгой, на каждой кнопке телефона кроме цифры были еще несколько букв. Причиной этого было обеспечение возможности выработать простые мнемонические правила для запоминания телефонных номеров. В Соединенных Штатах, как правило, на клавише 1 букв не было, на клавише 2 стояло ABC, на клавише 3 — DEF, на клавише 4 — GHI, на клавише 5 — JKL, на клавише 6 — MNO, на клавише 7 — PQRS, на клавише 8 — TUV, на клавише 9 — WXYZ, на клавише 0 букв также не было. Таким образом, номер 1-800-MY-APPLE соответствует номеру телефона 1-800-69-27753. Такие сочетания все еще встречаются в рекламных объявлениях, поэтому цифры с клавиатуры попали в современные приложения для смартфонов (рис. 9.4).

Рис. 9.4. В приложении Phone для iOS сохранены буквы на клавишах, которые были на старых телефонах

Как придумать новое мнемоническое правило для номера телефона? В 1990-х годах было популярно условно-бесплатное приложение, позволяющее это сделать. Разные части данной программы генерировали все перестановки букв телефонного номера и затем просматривали словарь, чтобы найти слова, которые содержали эти перестановки. Затем программа показывала пользователю перестановки с наиболее полными словами. Мы решим первую половину этой задачи. Поиск в словаре останется в качестве самостоятельного упражнения.

В предыдущей задаче, рассматривая генерацию перестановок, мы сгенерировали ответы, взяв существующую перестановку и поменяв местами позиции в ней, чтобы получить другую перестановку. Однако, как уже упоминалось, есть много способов генерации перестановок. В частности, в этой задаче для создания новой перестановки мы не будем менять местами две позиции существующей, а сгенерируем каждую перестановку с нуля. Сделаем это, просмотрев потенциальные буквы, которые соответствуют всем цифрам телефонного номера, и, переходя к каждой последующей цифре, станем добавлять новые варианты в конец списка. Это своего рода декартово произведение.

Что такое декартово произведение? В теории множеств декартово произведение — это совокупность всех комбинаций членов одного набора с каждым из членов другого набора. Например, если один набор содержит буквы «A» и «B», а другой — «C» и «D», то декартово произведение будет набором «AC», «AD», «BC» и «BD». «A» было объединено с каждой буквой из второго набора, и «B» было объединено с каждой буквой из второго набора. Если бы наш номер телефона был 234, нам необходимо было бы найти декартово произведение «A», «B», «C» на «D», «E» и «F». Как только мы получим этот результат, необходимо будет найти его декартово произведение с «G», «H», «I», и этот результат станет ответом.

Мы не будем работать с наборами. Мы будем работать с массивами, так как это удобнее для нашего представления данных.

Сначала определим соответствие чисел и потенциальных букв (листинг 9.8).

Листинг 9.8. PhoneNumberMnemonics.java

```
package chapter9;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Map;
import java.util.Scanner;

public class PhoneNumberMnemonics {
    Map<Character, String[]> phoneMapping = Map.of(
            '1', new String[] { "1" },
            '2', new String[] { "a", "b", "c" },
            '3', new String[] { "d", "e", "f" },
            '4', new String[] { "g", "h", "i" },
            '5', new String[] { "j", "k", "l" },
            '6', new String[] { "m", "n", "o" },
            '7', new String[] { "p", "q", "r", "s" },
            '8', new String[] { "t", "u", "v" },
            '9', new String[] { "w", "x", "y", "z" },
            '0', new String[] { "0", });
    private final String phoneNumber;
```

```
    public PhoneNumberMnemonics(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
```

Следующий метод выполняет декартово произведение двух массивов `String`, просто добавляя каждый элемент первого массива к каждому элементу второго массива и затем агрегируя эти результаты (листинг 9.9).

Листинг 9.9. PhoneNumberMnemonics.java (продолжение)

```
    public static String[] cartesianProduct(String[] first, String[] second)
    {
        ArrayList<String> product = new ArrayList<>(first.length *
          second.length);
        for (String item1 : first) {
            for (String item2 : second) {
                product.add(item1 + item2);
            }
        }
        return product.toArray(String[]::new);
    }
```

Теперь мы можем найти все возможные мнемонические правила для данного номера телефона. Функция `getMnemonics()` выполняет это, последовательно считывая декартово произведение каждого предыдущего произведения (начиная с массива, содержащего одну пустую строку) и массива букв, представляющих следующую цифру. `main()` запускает функцию `getMnemonics()` для любого телефонного номера, предоставленного пользователем (листинг 9.10).

Листинг 9.10. PhoneNumberMnemonics.java (продолжение)

```
    public String[] getMnemonics() {
        String[] mnemonics = { "" };
        for (Character digit : phoneNumber.toCharArray()) {
            String[] combo = phoneMapping.get(digit);
            if (combo != null) {
                mnemonics = cartesianProduct(mnemonics, combo);
            }
        }
        return mnemonics;
    }

    public static void main(String[] args) {
        System.out.println("Enter a phone number:");
        Scanner scanner = new Scanner(System.in);
        String phoneNumber = scanner.nextLine();
        scanner.close();
        System.out.println("The possible mnemonics are:");
        PhoneNumberMnemonics pnm = new PhoneNumberMnemonics(phoneNumber);
        System.out.println(Arrays.toString(pnm.getMnemonics()));
    }
}
```

Оказывается, номер телефона 144-07-87 можно записать как 1GH0STS, что легче запомнить.

9.4. РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

Динамическое программирование, с помощью которого решалась задача о рюкзаке, — это широко используемый метод, который позволяет решить задачи, на первый взгляд кажущиеся нерешаемыми, путем разбиения их на более мелкие задачи и составления решения из этих частей. Сама задача о рюкзаке связана с другими оптимизационными задачами, где конечное количество ресурсов (вместимость рюкзака) должно быть распределено среди конечного, но исчерпывающего набора вариантов (предметов, которые можно украсть). Представьте колледж, который должен освоить свой спортивный бюджет. У него недостаточно денег для финансирования всех команд и есть ожидания относительно того, сколько пожертвований для выпускников внесет каждая команда. Это может привести к возникновению похожей на задачу о рюкзаке проблемы оптимизации распределения бюджета. Подобные задачи часто встречаются в реальном мире.

Задача коммивояжера — типичное явление для таких компаний, как UPS и FedEx. Компании по доставке посылок хотят, чтобы их машины двигались по наикратчайшим маршрутам. Это не только делает работу водителей более приятной, но и экономит топливо и уменьшает расходы на техническое обслуживание. Мы все путешествуем во время работы или для удовольствия, и поиск оптимальных маршрутов при посещении нескольких мест помогает экономить ресурсы. Но задача коммивояжера не сводится к одной лишь маршрутизации путешествий, она встречается практически в любом сценарии маршрутизации, который требует единичных посещений узлов. Несмотря на то что минимальное связующее дерево (см. главу 4) может свести к минимуму длину кабелей, необходимых для соединения соседних узлов, оно не сообщает нам оптимальную длину проводов, если каждый дом должен быть напрямую подключен только к одному другому дому как части гигантской сети, которая замыкается на свою начальную точку. Эту проблему решает задача коммивояжера.

Методы генерации перестановок, подобные используемым при наивном подходе к решению задачи коммивояжера и задачи мнемоники телефонных номеров, полезны для тестирования всевозможных алгоритмов перебора. Например, если вы пытаетесь взломать короткий пароль, то можете сгенерировать все возможные перестановки символов, которые потенциально в нем присутствуют. Специалистам, выполняющим масштабные задачи генерации перестановок, стоит применять очень эффективный алгоритм генерации перестановок, такой как алгоритм Хипа[1].

---

[1] *Sedgewick R.* Permutation Generation Methods. — Princeton University. http://mng.bz/87Te.

9.5. УПРАЖНЕНИЯ

1. Перепишите реализацию наивного подхода к задаче коммивояжера, используя графовую структуру из главы 4.
2. Реализуйте генетический алгоритм (см. главу 5) для решения задачи коммивояжера. Начните с простого набора данных о городах штата Вермонт, описанного в этой главе. Сможете ли вы получить генетический алгоритм для достижения оптимального решения в короткие сроки? Затем попытайтесь решить задачу, постепенно увеличивая количество городов. Хорошо ли работает генетический алгоритм? В интернете вы найдете большое количество наборов данных, специально созданных для решения задачи коммивояжера. Разработайте структуру тестирования для проверки эффективности своего метода.
3. Подключите словарь к программе мнемоники телефонных номеров и возвращайте только те перестановки, которые содержат допустимые слова из словаря.

10 Интервью с Брайаном Гетцем

Брайан Гетц — один из лучших Java-экспертов в мире. В качестве архитектора Java в Oracle он помогает направлять развитие языка и поддерживающих его библиотек. Он провел несколько важных модернизаций языка, включая Project Lambda. Брайан имеет большой опыт в разработке программного обеспечения, он автор бестселлера *Java Concurrency in Practice*[1] (Addison-Wesley Professional, 2006).

Интервью проведено 25 августа 2020 года в доме Брайана в Уиллистоне, штат Вермонт. Для ясности расшифровка стенограммы была отредактирована и сокращена.

Как вы заинтересовались информационными технологиями?

— Я начал изучать информационные технологии в 1978 году, когда мне было 13 или 14 лет. У меня был доступ к системе таймшер через местную школу, и мой старший брат, который прошел ту же программу до меня, приносил мне книги и другие материалы для изучения. И меня в какой-то момент зацепило. Я был полностью увлечен этой системой, в которой использовался сложный, но понятный набор правил. После школы я проводил по возможности все свое свободное время в компьютерном классе. В то время в программировании не существовало доминирующего языка, как, вероятно, в последние 25 лет. Ожидалось, что все будут знать несколько языков программирования. Я самостоятельно изучал BASIC, Fortran, COBOL, APL

[1] *Гетц Б., Пайерлс Т., Блох Д., Боубер Д., Холмс Д. , Ли Д.* Java Concurrency на практике. — СПб.: Питер, 2020.

и ассемблер. Я понимал, что каждый из них был отдельным инструментом для решения разных задач. Я был абсолютным самоучкой, так как в то время не было никакого формального образования. Моя степень не по информатике, а по математике, так как в то время во многих школах не было даже факультетов информатики. И я думаю, что математическая направленность мне очень пригодилась.

Был ли язык программирования, который оказал на вас самое большое влияние?

— Когда я только учился, такого языка для меня не было. Я просто изучал разные языки. В то время для различных категорий задач преобладающими языками были Fortran, COBOL и BASIC. Но позже, когда я уже стал аспирантом, у меня была возможность пройти курс «Структура и интерпретация компьютерных программ» в Массачусетском технологическом институте, где я изучил схему. И именно здесь мне пришло озарение. На тот момент я программировал уже почти десять лет и столкнулся с множеством интересных задач. Тогда я узнал, что существует всеобъемлющая теория, которая может связать многие сделанные мной наблюдения. Что касается меня, мне очень повезло, что я учился в этом классе в качестве аспиранта, а не в качестве первокурсника, так как первокурсники были просто ошеломлены объемом материала для изучения. Наличие большего опыта позволило мне увидеть скрытую красоту и структуру, не отвлекаясь на детали. Если мне нужно выбрать момент, когда красота вычислений действительно стала для меня очевидна, то это был именно мой урок.

Как вы развили свои навыки программирования и информатики после колледжа?

— Я думаю, так же, как и почти все остальные, — я просто много работал. В типичных рабочих ситуациях вы чаще всего не знаете, что делать с поставленной задачей, и вам приходится решать ее самостоятельно. В вашем распоряжении имеется целый ряд инструментов, и не всегда очевидно, какой из них правильный, но никто не отменял метод проб и ошибок. Когда задача подходит к определенному порогу сложности, вы начинаете понимать, что работает, а что нет. Надеюсь, параллельно с этим приходит какое-то индуктивное умозаключение, когда вы можете выяснить, почему что-то сработало и когда это может сработать снова, а может, и нет. В начале моей карьеры в области разработки программного обеспечения я занимал несколько должностей. Я работал в исследовательской лаборатории небольшой компании, занимающейся разработкой программного обеспечения для сетей. Я учился на практике и экспериментировал, как и большинство разработчиков в наше время.

Как ваша карьера привела к тому, что вы решили стать архитектором языка Java?

— Довольно странным и интересным путем! Первую половину своей карьеры я был в основном обычным программистом. Но в какой-то момент стал промежуточным звеном между программированием и обучением: проводил презентации, писал статьи и в конечном итоге написал книгу. Я всегда старался выбирать темы, которые сбивали меня с толку, исходя из того, что они могут сбивать с толку и других, и я старался объяснить их понятным языком. Я обнаружил, что у меня есть талант разрешать проблемы между техническими деталями и интуитивно понятными ментальными моделями, что привело к написанию книги «Java Concurrency на практике» — это было почти 15 лет назад! Затем я перешел на работу в компанию Sun на должность, которая была больше связана с информационно-разъяснительной работой, чем с разработкой. Я объяснял людям, как работает JVM, что делает динамический компилятор и почему динамическая компиляция потенциально лучше статической. Я пытался демистифицировать технологию и развенчать окружающие ее мифы. Оказавшись там, я понял, как работать в команде Java, и в какой-то момент появилась возможность и мне внести более существенный вклад в эту деятельность. Если бы мне пришлось на карте изобразить, какой путь я проделал, не знаю, как бы я его нарисовал. Это определенно не было бы прямой линией.

Что значит быть архитектором языка Java?

— Попросту говоря, я должен решить, в каком направлении развивается модель программирования Java. И конечно же, здесь существует множество вариантов и людей, которые с радостью дадут мне совет! Я уверен, что у каждого из девяти миллионов разработчиков Java есть идея. Безусловно, мы не сможем реализовать многие из них. Поэтому нам необходимо выбирать очень и очень внимательно. Моя работа состоит в том, чтобы сбалансировать необходимость двигаться вперед, сохраняя актуальность Java как языка программирования. Однако актуальность имеет много измерений: отношение к задачам, которые мы хотим решить; отношение к оборудованию, на котором работаем; отношение к интересам, ориентации и даже моде в сфере программирования. Мы должны развиваться, но в то же время не можем двигаться так быстро, чтобы терять людей. Если мы в одночасье радикально изменим ситуацию, люди скажут: «Это не тот Java, который я знаю» и займутся чем-нибудь другим. Итак, мы должны выбрать как направление, так и скорость развития, чтобы оставаться в курсе проблем, которые люди хотят решать, не вызывая дискомфорта.

Для меня это означает оказаться на месте Java-разработчиков и понять, где их проблемные места. Затем мы пытаемся изучить язык так, чтобы смочь

с его помощью устранять проблемы. Генри Форду приписывают старую поговорку: «Если бы я поспрашивал людей о том, чего они хотят, они бы ответили: “Более быстрого коня!”». Программисты очень склонны говорить: «Вам необходимо добавить эту функцию». Тем самым они надеются, что вы прислушаетесь к их совету и поймете все проблемные вопросы. Сравнивая это с тем, что мы слышали от других разработчиков, возможно, мы сможем понять, чего действительно не хватает, что на самом деле может решить проблемы людей, сделает их более продуктивными, а программы — более безопасными и эффективными.

Как развивается язык Java? Как было решено добавить в язык новые функции?

— На самом деле довольно редко что-либо изобретается полностью. Почти каждая «новая» мысль витает в мире программирования десятилетиями. Когда кто-то приходит ко мне с идеей добавления функции, я вижу, что это уже было сделано на каком-то другом языке давным-давно. Бóльшая часть процесса — это ожидание подходящего момента, чтобы раскрыть новую концепцию и адаптировать ее таким образом, чтобы она соответствовала остальному языку. Всегда есть идеи усовершенствования функций, и в каждом языке вы найдете множество замечательных функций. Настоящая игра состоит в том, чтобы понять их и спросить: «Что дает вам эта функция, что вы не смогли бы сделать иначе? Как это делает ваши программы безопаснее? Как это позволит улучшить проверку типов? Как это позволит сделать ваши программы менее подверженными ошибкам, более понятными и т. д.?»

Это довольно субъективный процесс. Если мы стремимся облегчить труд людей, то должны делать субъективные запросы о том, какие виды задач более важно упростить именно сейчас. Вы можете посмотреть на большие функции, которые ранее были добавлены в Java. В середине 2000-х годов были дженерики, и это был очевидный пробел. В то время язык требовал параметрического полиморфизма. Его хотели получить в 1995 году, но не знали, как это сделать так, чтобы это имело смысл для языка. И не хотели прививать шаблоны C++ к Java. Еще почти десять лет ушло на то, чтобы понять, как добавить параметрический полиморфизм и абстракцию данных в Java так, чтобы это выглядело естественным. И я считаю, что они проделали фантастическую работу. Мы сделали то же самое с поведенческой абстракцией, когда недавно создавали лямбды. Опять же вся тяжелая работа заключалась не в теории. Теория лямбда-выражений хорошо изучена с 1930-х годов. Самым сложным было добиться того, чтобы лямбда-выражения вписались в Java. И в итоге, когда вы делаете что-то через 3, 5 или 7 лет, люди спрашивают, что у нас заняло так много времени, если это слишком очевидно. Мы не хотим навязывать людям свои теории, поэтому не должны торопиться.

Как вы понимаете, рассматривая функцию, что это не просто мода, а что-то важное, что действительно нужно разработчикам?

— Это действительно хороший вопрос, так как уже было допущено несколько серьезных промахов. В начале 2000-х был большой порыв добавить в язык Java XML-литералы, и я считаю, что правильно отказались от этой идеи. Не все языки этого избежали.

Я не могу предоставить вам алгоритм. Часто вам просто нужно долго думать и анализировать, как все взаимосвязано. Мы все видели языки, в которых функция закреплена на стороне для решения конкретной задачи, но эта задача может быть проблемой не на все времена. Если вы готовы набраться терпения и думать об этом снова и снова, прежде чем принять решение, очень часто вы можете почувствовать, что это лучшее, что вы придумали.

Какими дополнительными возможностями в Java, внесенными за время вашей работы в качестве архитектора языка, вы больше всего гордитесь?

— Я был руководителем команды по добавлению лямбда-выражений в Java. Это не только было огромным изменением, но и меняло то, как язык будет развиваться. В каком-то смысле это было сделано и затем разрушено, так как в то время, когда это происходило, компания Sun была занята тем, что медленно уходила из бизнеса, и у нас не было возможности развивать платформу так быстро, как хотелось. В то время было ясно, что Java отстает, и это был наш большой шанс доказать миру, что Java может оставаться актуальной и увлекательной для программирования.

Основная проблема добавления лямбда-выражений в Java заключалась в том, что нужно было сделать так, чтобы это выглядело естественно. Всегда были идеи и советы, как это сделать, и почти всегда предлагали делать так, как в другом языке. Все это было бы неудачной попыткой. Мы могли бы добиться этого на год или два быстрее, но не получили бы результат, который работает так, как сейчас. Чем я действительно горжусь — это тем, что нам удалось выяснить, как интегрировать лямбда-выражения в язык на нескольких уровнях. Это очень чисто работает с системой универсальных типов. Нам пришлось пересмотреть ряд других аспектов языка, чтобы заставить лямбда-выражения работать. Нам пришлось пересмотреть вывод типов и взаимодействие с выбором перегрузки. Если вы спросите людей, какие функции были добавлены в Java 8, лямбда-выражений никогда не будет ни в одном списке. Но это была огромная работа — это основа, необходимая для того, чтобы вы могли написать рабочий код.

Мы также должны были приступать к решению проблемы эволюции совместимых API. Один из огромных рисков, с которыми мы столкнулись в Java 8, заключался в том, что способ написания библиотеки с лямбда-выражениями

сильно отличается от написания библиотеки без лямбда-выражений. У нас не было цели совершенствовать язык, а затем оказаться в ситуации, когда внезапно все наши библиотеки окажутся на 20 лет старше. Мы должны были решить вопрос, как собираемся развивать библиотеки одновременно с преимуществами новых идей проектирования. Это привело к возможности совместимого добавления методов к интерфейсам, что было важной частью сохранения актуальности имеющихся у нас библиотек. С первого дня язык и библиотеки были готовы к новому стилю программирования.

Многие студенты задают мне вопрос: сколько им следует использовать лямбда-выражений? Можно ли постоянно применять их в своем коде?

— Я подхожу к этому вопросу, возможно, с другой точки зрения, чем многие студенты, так как бóльшая часть написанного мной кода представляет собой библиотеки, предназначенные для использования. Требования к написанию такой библиотеки очень высоки, поскольку вам необходимо сделать это правильно с первого раза. Требования к совместимости для изменения кода также очень высоки. Когда вы переходите границу между пользовательским кодом и кодом библиотеки, вы видите, что главное, что могут лямбда-выражения, — это проектировать API, которые можно параметризовать не только данными, но и поведением, так как лямбда-выражения позволяют нам рассматривать поведение в качестве данных. Поэтому я сосредоточен на взаимодействии между клиентом и этой библиотекой в естественном потоке управления. Когда для клиента имеет смысл анализировать все строки, а когда — передать какое-то поведение библиотеке, которую он вызовет в определенное время? Я не уверен, что мой опыт напрямую влияет на опыт ваших учеников. Но ключ к успеху здесь — это способность распознать границы в вашем коде и разделение ответственности. Независимо от того, строго ли они разграничены в отдельно скомпилированных модулях или тщательно задокументированных API или это просто соглашения о том, как мы организуем наш код, мы хотим всегда помнить об этом.

Мы используем границы в нашем коде для управления сложностью по принципу «разделяй и властвуй». Каждый раз, проектируя одну из этих границ, вы проектируете небольшое протокольное взаимодействие и должны думать о ролях участников с каждой стороны и о том, какой информацией они обмениваются и как выглядит этот обмен.

Вы ранее говорили о периоде застоя для Java. Когда это было и почему?

— Я бы сказал, что версии Java 6 и 7 уже давно устарели. Неслучайно это было время, когда Scala начала стремительно развиваться, отчасти потому, я думаю, что экосистема предполагала следующее: «Что ж, нам может потребоваться

найти другую лошадь, если Java не встанет и не запустится». К счастью, наша лошадь встала и побежала и с тех пор работает.

В настоящее время мы наблюдаем довольно быстрое развитие языка. Как изменилась философия?

— В целом она не сильно изменилась, но в деталях произошли довольно значительные перемены. Начиная с Java 9 мы перешли к шестимесячной схеме выпуска, отказавшись от выпуска раз в несколько лет. Для этого было множество веских причин, одна из них — наличие хороших идей, которые всегда игнорировались, когда мы планировали выпуски раз в несколько лет с крупными драйверами. Более частый выпуск позволил нам лучше сочетать большие и маленькие функции. Многие из этих шестимесячных выпусков имеют более мелкие языковые функции, такие как определение типа локальной переменной. На это не обязательно уходило всего шесть месяцев. На это может потребоваться год или два, но теперь у нас есть больше возможностей представить что-либо готовое. В дополнение к более мелким функциям вы увидите более крупные, такие как сопоставление с образцом, которые могут постепенно воспроизводиться в течение нескольких лет. Более ранние элементы языка могут дать нам представление о том, в каком направлении он развивается.

Существуют также группы связанных функций, которые могут быть доставлены индивидуально. Например, сопоставление с образцом, записи и запечатанные типы работают вместе, чтобы поддерживать более ориентированную на данные модель программирования. И это не случайно. Это основано на наблюдении за вопросами, использующими систему статических типов Java для моделирования данных. А как изменились программы за последние десять лет? Они стали меньше. Люди пишут небольшие функциональные единицы и развертывают их как микросервисы. Таким образом, большая часть кода находится ближе к границе, на которой он будет получать данные от какого-либо партнера, будь то JSON, XML или YAML, через соединение сокета, который затем превратится в некоторую модель данных Java. Мы хотели упростить моделирование данных. Таким образом, этот набор функций предназначен для совместной работы. И вы можете увидеть похожие группы функций во многих других языках, только с другими названиями. В ML вы бы назвали их алгебраическими типами данных, так как записи — это типы продуктов, а закрытые классы — типы сумм, и вы выполняете полиморфизм над алгебраическими типами данных с сопоставлением с образцом. Это отдельные функции, которых, возможно, разработчики Java не видели раньше, так как они не программировали на Scala, ML или Haskell. Они могут быть новичками в Java, но это не новые концепции, и было доказано, что их совместная работа позволяет создать

стиль программирования, соответствующий задачам, которые люди решают в настоящее время.

Интересно, есть ли в Java хотя бы одна новая функция, которая вас интересует больше всего?

— Я очарован функцией сопоставления с образцом, так как, работая над ней, понял, что в Java все это время отсутствовала часть объектной модели, а мы просто этого не заметили. Java предлагает хорошие инструменты для инкапсуляции, но это только один путь. Вы вызываете конструктор с некоторыми данными, и он дает вам объект, который затем очень уклончиво отказывается от своего состояния. И это обычно происходит через какой-то специальный API, о котором сложно думать программно. Но есть большая категория классов, которые просто моделируют старые простые данные. Понятие шаблона деконструкции на самом деле двойственное понятие, которым мы пользовались с самого начала, а именно — конструктор. Конструктор принимает состояние и превращает его в объект. Как деконструировать объект до состояния, с которого вы начали (или с которого можно было бы перезапустить)? Это именно то, что позволяет вам выполнять сопоставление с образцом. Оказывается, существует множество проблем, для которых решение задачи сопоставления с образцом намного более прямолинейно, элегантно и, что наиболее важно, компонуемо, чем решение специальных задач.

Я говорю об этом, так как, несмотря на то что мы узнали об успехах в теории языков программирования за последние 50 лет, мое краткое изложение истории языков программирования, состоящее из одного предложения, звучит так: «У нас есть один хороший рабочий трюк». И этот трюк — композиция. Это единственное, что помогает справиться со сложностью. И вы, как архитекторы языка программирования, должны найти методы, позволяющие разработчикам работать с композицией, а не против нее.

Почему важно знать методы решения проблем из области программирования?

— Стоять на плечах гигантов! Существует множество задач, которые уже были решены кем-то другим, часто с бóльшими усилиями и затратами и с бóльшим количеством фальстартов. Если вы не знаете, как распознать такую задачу, у вас возникнет соблазн заново изобрести ее решение — и, вероятно, вы так и не соберетесь это делать.

На днях я читал забавный комикс о том, как работает математика. Когда изобретается что-то новое, сначала никто не верит в достоверность изобретения и на выяснение деталей уходят годы. Может пройти немало времени, прежде чем остальная часть математического сообщества согласится с тем, что

изобретение действительно имеет смысл. А позже вы тратите на это 45 минут на лекции, и когда студент не понимает, профессор спрашивает: «Мы вчера потратили на эту тему весь урок, как вы могли этого не понять?» Многие концепции, которые мы рассматриваем на уроке, являются результатом многолетних усилий. Задачи, которые мы решаем, довольно сложны, поэтому нам необходима любая помощь. Если мы сможем рассмотреть проблему так, чтобы какую-то часть можно было решить с помощью существующей техники, это очень поможет. Это означает, что вам не нужно изобретать заново решение, и тем более неплохое решение. Вам не нужно заново открывать для себя все способы, в которых очевидное решение оказывается неправильным. Вы можете просто положиться на существующее решение и сосредоточиться на уникальной части своей задачи.

Иногда учащимся трудно представить себе, как изучаемые ими проблемы со структурой данных и алгоритмами на самом деле появятся при разработке программного обеспечения в реальном мире. Часто ли возникают проблемные вопросы при разработке программного обеспечения?

— Это напоминает мне разговор с моим научным руководителем, состоявшийся, когда я примерно через 10–15 лет после выпуска навещал его. Он задал мне два вопроса. Первый: «Используете ли вы математику, которую выучили здесь, в своей работе?» И я сказал: «Ну, если честно, не очень часто». Второй: «Используете ли вы навыки мышления и анализа, полученные при изучении математики?» И я ответил: «Безусловно, каждый день». И он улыбался с гордостью за хорошо выполненную работу.

Например, рассмотрим пример применения красно-черных деревьев. Как они работают? В большинстве случаев мне не о чем беспокоиться. При необходимости в каждом языке программирования имеется заранее написанная, хорошо протестированная высокопроизводительная библиотека. Важным навыком считается не возможность воссоздать эту библиотеку, а знание того, как определить, когда вы можете использовать ее с выгодой для решения более серьезной проблемы, является ли она правильным инструментом, как впишется во временную или пространственную сложность вашего общего решения и т. д. Это навыки, которые вы задействуете постоянно. Когда вы находитесь в центре класса структур данных, может быть трудно за деревьями увидеть лес. Вы можете проводить много времени в классе, работая над механикой красно-черного дерева, и это может быть важно, но это то, что вам, скорее всего, больше никогда не придется делать. И надеюсь, вас не попросят сделать это во время интервью, так как я думаю, что это плохой вопрос для интервью! Но вам необходимо знать, какой может быть временная сложность поиска в дереве, какие условия должны быть при распределении ключей, чтобы достичь этой сложности, и т. д. Это тот вид мышления, который разработчики должны применять каждый день.

Можете ли вы привести пример, когда вы или другой специалист смогли использовать знания из области информатики для более эффективного решения проблемы?

— В моей работе это довольно забавно, так как теория очень важна для многого из того, что мы делаем. Но теория не может решить проблему за вас в реальном языковом дизайне. Например, Java — это не чистый язык, поэтому теоретически из монад нельзя чему-то научиться. В то же время у монад есть чему поучиться. Поэтому когда я рассматриваю возможную функцию, я могу опираться на множество теорий. Это дает мне основы, но настоящее обучение мне придется пройти самостоятельно. То же самое с системами типов. Большая часть теории типов не занимается такими эффектами, как выброс исключений. Что ж, у Java есть исключения. Это не значит, что теория типов бесполезна. При разработке языка Java я могу опираться на множество теорий типов. Но должен признать, что теория поможет мне только иметь представление, а основное обучение мне придется пройти самостоятельно.

Найти этот баланс сложно. Но это важно, так как слишком легко сказать: «О, теория мне не поможет», и тогда вы начинаете все заново.

Какие области информатики важны для развития языка?

— Теория типов — это очевидно. У большинства языков существует система типов, а у некоторых их даже больше одной. Java, например, во время статической компиляции имеет одну систему типов, во время выполнения — другую. Есть даже третья система типов для проверки времени. Эти системы типов, конечно, должны быть согласованными, но они должны иметь разную степень точности и детализации. Так что теория типов, безусловно, важна. Формальной работы много над семантикой программ, о которой полезно знать, но не обязательно что-то, что применяется в повседневном языковом дизайне. И я не думаю, что какой-нибудь разумный проект пройдет без открытия книг по теории шрифтов и чтения десятков статей.

Если кто-то заинтересован в изучении языкового дизайна, что вы можете порекомендовать, опираясь на свой опыт, чтобы когда-нибудь он занял такую же должность, как ваша?

— Очевидно, для того чтобы заниматься языковым дизайном, вы должны разбираться в инструментах, которые используют разработчики языка. Вы должны понимать компиляторы, системы типов и все детали теории вычислений: конечные автоматы, контекстно-свободные грамматики и т. д. Это необходимое условие для понимания всего этого. Также очень важно уметь программировать на нескольких языках, чтобы видеть разные способы решения проблем, различные предположения, инструменты, уметь их сравнивать

и анализировать. Чтобы преуспеть в языковом дизайне, вам необходимо иметь широкий взгляд на программирование. Вам также нужно иметь перспективу системного мышления. Когда вы добавляете функцию к языку, это меняет то, как разработчики будут программировать на нем, и меняет набор направлений в будущем. Вам необходимо не только понять, как функция будет применяться, но и проанализировать и сравнить новое и старое направления.

Фактически я бы дал следующий совет: всем независимо от того, интересуется кто-то языками программирования или нет, начать изучать разные типы языков программирования. Изучение более чем одной парадигмы программирования сделает их более компетентными программистами. Когда так подходишь к проблеме, легче найти несколько способов ее решения. Я особенно рекомендую выучить функциональный язык, потому что он поможет вам найти способ создавать программы и расширит ваши навыки (в хорошем смысле).

Какие ошибки вы часто наблюдаете у Java-программистов, которых можно избежать, если лучше использовать возможности языка?

— Я думаю, что самое важное — это понять, как работают дженерики. В дженериках есть несколько неочевидных концепций, но их не так много и они не так уж и сложны, как кажется на первый взгляд. Дженерики — это основа других функций, таких как лямбда-выражения, а также ключ к пониманию многих библиотек. Но многие разработчики рассматривают это как упражнение (что мне сделать, чтобы красные волнистые линии исчезли?), а не как рычаг.

Что, по вашему мнению, изменится в следующие 5–10 лет для работающих программистов?

— Я подозреваю, что это будет интеграция традиционного решения вычислительных задач с машинным обучением. В настоящее время программирование и машинное обучение — это совершенно разные области. Современные методы машинного обучения бездействовали в течение 40 лет. Вся работа над нейронными сетями велась в 1960–1970-е годы. Но до сих пор у нас не было ни вычислительных мощностей, ни данных для их обучения. Теперь у нас все это есть, и они внезапно стали актуальными. Вы видите, как машинное обучение применяется к таким вещам, как распознавание рукописного ввода, распознавание речи, обнаружение мошенничества и все то, что мы пытались решить (не очень хорошо) с помощью систем на основе правил или эвристики. Но проблема в том, что инструменты, которые мы используем для машинного обучения, и стили мышления, которые применяем для машинного обучения, полностью отличаются от написания традиционных программ. Я думаю, что в следующие 20 лет это будет большой проблемой для программистов. Как они собираются соединить эти два разных типа мышления в этих двух разных наборах инструментов для решения проблем, требующих и тех и других навыков?

Как вы думаете, какими будут самые большие эволюционные изменения в языках программирования в течение следующего десятилетия?

— Сейчас мы наблюдаем широкую тенденцию, которая заключается в конвергенции объектно-ориентированных и функциональных языков. Двадцать лет назад языки программирования были строго разделены на функциональные, процедурные и объектно-ориентированные, и у каждого из них была собственная философия моделирования мира. Но каждая из этих моделей была в некотором роде несовершенной, так как представляла только часть мира. То, что мы видели за примерно последнее десятилетие, начиная с таких языков, как Scala и F#, а теперь и таких языков, как C# и Java, — это то, что многие концепции, которые изначально укоренились в функциональном программировании, находят свое отражение в языках с более широким спектром. И я думаю, что эта тенденция будет только развиваться. Некоторые люди любят шутить, что все языки сходятся в $MY_FAVORITE_LANGUAGE. В этой шутке есть доля правды: функциональные языки получают все больше инструментов для инкапсуляции данных, а объектно-ориентированные языки — больше инструментов для функциональной композиции. И тому есть очевидная причина: оба этих набора считаются полезными. Каждый из них отлично справляется с той или иной проблемой, и мы решаем проблемы, имеющие оба аспекта. Поэтому я думаю, что в следующие 10 лет мы увидим усиление конвергенции концепций, которые традиционно считались объектно-ориентированными, и концепций, которые традиционно считались функциональными.

Я думаю, что существует много примеров влияния мира функционального программирования на Java. Можете ли вы привести несколько?

— Самый очевидный — это лямбда-выражения. И знаете, не совсем справедливо называть их концепцией функционального программирования, поскольку лямбда-исчисление предшествует возникновению вычислительной техники на несколько десятилетий. Это естественная модель для описания и формирования поведения. В таком языке, как Java или C#, это имеет такой же смысл, как и в Haskell или ML. Так что это однозначно. Другой аналогичный метод — сопоставление с образцом, которое большинство пользователей связывают с функциональными языками, так как это первое место, где они его увидели, но на самом деле сопоставление с образцом в первую очередь относится к таким языкам, как SNOBOL из 1970-х годов, который был языком обработки текста. Сопоставление с образцом очень четко вписывается в объектную модель. Это не чисто функциональная концепция. Просто так случилось, что функциональные языки заметили, что это полезно, немного раньше нас. Многие из концепций, которые мы ассоциируем с функциональными языками, имеют смысл и в объектно-ориентированных языках.

По многим параметрам Java — один из самых популярных языков программирования в мире. Как вы думаете, что привело к такому успеху и почему он будет иметь успех в будущем?

— Вы должны осознавать, какую роль успех сыграл в вашей жизни. Я думаю, что во многих отношениях Java появился как раз в подходящее время. В то время принималось решение о переходе с C на C++. Тогда C был доминирующим языком, а C++, с одной стороны, предлагал лучшую основу, чем C, а с другой — был очень сложным. Неужели мы действительно хотим совершить этот прыжок? И появился язык Java, в котором предоставлялась большая часть того, что было в C++, но без особых сложностей. Да, пожалуй, это то, что мы хотим! Это было правильно и в подходящее время. На вооружение был взят ряд старых идей, которые годами изучались в компьютерном мире, включая сборку мусора и встраивание параллелизма в модель программирования, которые ранее не использовались в серьезных коммерческих языках. Все это было связано с проблемами, которые люди решали в 1990-е годы.

Джеймс Гослинг сравнивал Java с волком в овечьей шкуре. Людям нужна была очистка памяти от ненужных данных, им нужна была интегрированная модель параллелизма, которая была бы совершеннее библиотеки Pthreads. Но им не нужны были языки, с которыми эти вещи традиционно поставлялись, так как они поставлялись со множеством других вещей, которые чертовски пугали. В то же время язык Java аналогичен C. На самом деле разработчики изо всех сил старались сделать синтаксис похожим на C. По поводу создания Java возникало множество идей. Одна из них состояла в том, что создатели спроектировали всю среду исполнения языка в ожидании того, что своевременная компиляция должна была появиться, но не совсем там. Первая версия Java 1995 года была строго интерпретирована. Java развивался медленно, но каждое дизайнерское решение относительно языка, формата файла класса и структуры времени выполнения принималось с учетом ноу-хау, позволяющего сделать это быстро. В конце концов Java стал достаточно быстрым, а в некоторых случаях даже быстрее, чем C (хотя некоторые разработчики до сих пор не верят, что это возможно). Было много замечательных идей о том, где будут развиваться технологии и что действительно нужно людям. Все это привело к развитию Java. Однако чтобы Java оставался на первом месте, нам необходимо было нечто большее. И я думаю, что то, что поддерживало нас даже в те мрачные времена, — это неустанная приверженность совместимости.

Внесение несовместимых изменений нарушает ваши обещания. Это делает недействительными вложения ваших клиентов в свой код. Всякий раз, когда вы изменяете код, практически вы даете возможность переписать его на каком-нибудь другом языке, а в Java никогда такой возможности не было. Код Java, который вы написали 5, 10, 15, 20, 25 лет назад, все еще работает. Это означает, что мы развиваемся немного медленнее, но наше понимание кода

и того, как работает язык, сохраняется. Мы не нарушаем своих обещаний и не причиняем вреда нашим пользователям. Проблема в том, как найти баланс между продвижением вперед и стремлением к совместимости. Я думаю, что это наше секретное оружие. За последние 25 лет мы придумали, как это сделать, и у нас это неплохо получается. Это то, что позволяет нам добавлять универсальные шаблоны, лямбда-выражения, модули, сопоставление с образцом и другие вещи, которые могут показаться чуждыми Java, потому что мы выяснили, как это сделать.

Игра го получает большое признание за свою интегрированную модель параллелизма, но в Java уже были примитивы синхронизации, ключевые слова и потоковая модель, встроенные в язык еще в 1995 году. Как вы думаете, почему из-за этого к Java не относятся с бóльшим вниманием?

— Я думаю, отчасти это связано с тем, что большая часть реализаций невидима. То, что просто работает, часто не приковывает должного внимания. По нескольким причинам я не большой поклонник го. Все думают, что модель параллелизма — это секретное оружие го, но я думаю, что их модель параллелизма склонна к ошибкам. То есть у вас есть сопрограммы с очень простым механизмом передачи сообщений (каналы). Но почти во всех случаях объекты на одной или другой стороне канала будут иметь какое-то общее изменяемое состояние, защищенное блокировками. А это означает, что у вас объединяются ошибки, которые вы можете совершить при передаче сообщений, и ошибки, которые вы можете допустить при параллелизме с общим состоянием, в сочетании с тем фактом, что примитивы параллелизма в го для параллелизма с общим состоянием значительно слабее, чем в Java. Например, их блокировки нереентерабельны, что означает: вы не можете составить какое-либо поведение, использующее блокировки. Это говорит о том, что вам часто приходится писать две версии одного и того же: одну — для вызова с удерживаемой блокировкой, а другую — без нее. Я думаю, что люди обнаружат, что модель параллелизма го, в отличие от Reactive, будет переходной технологией, которая какое-то время выглядела привлекательно, но вскоре обязательно появится что-то лучше и совершеннее. Конечно, это может просто показывать мою предвзятость.

Как выглядит ваша повседневная работа в роли архитектора языка Java?

— Фактически я работаю везде и всюду. В любой день я мог просто исследовать эволюцию языка и анализировать будущие возможности. Я мог создать прототип чего-нибудь, чтобы увидеть, как движущиеся элементы сочетаются друг с другом. Я мог написать заявление для своей команды: «По моему мнению, вот где мы находимся в процессе решения этой проблемы, вот что мы выяснили и такие проблемы еще остались». Я мог выступать на конференциях, общаться с пользователями, пытаясь понять, с какими проблемами они сталкиваются, и в какой-то степени продавать информацию о том, что ожидается

в будущем. В любой день может быть что угодно. Некоторые из этих вещей очень актуальны, некоторые станут актуальны в будущем, а некоторые уже устарели, некоторые направлены к сообществу, а некоторые касаются только меня. Каждый день разный!

В настоящий момент один из проектов, в котором я участвую и над которым мы работаем в течение нескольких лет, — это обновление системы общих типов для поддержки примитивов и агрегатов, подобных примитивам. Это то, что касается языка, компилятора, стратегии перевода, формата файла класса и JVM. Чтобы можно было достоверно сказать, что у нас есть история, все эти направления должны быть выстроены в одну линию. Так что в любой день я мог работать со всеми направлениями, чтобы увидеть, правильно выстраивается история или нет. Это процесс, который может занять годы!

Что вы посоветуете программистам-самоучкам, студентам или опытным разработчикам, чтобы улучшить свои навыки в области компьютерных наук?

— Один из наиболее ценных советов, помогающих понять технологию: поместить ее в исторический контекст. Как эта технология соотносится со всем, что существовало до нее для решения той же проблемы? Большинство разработчиков не всегда могут выбирать, какую технологию они задействуют для решения проблемы. Если бы вы были разработчиком в 2000 году и устроились на работу, вам бы сказали: «Мы используем эту базу данных, этот контейнер приложения, эту IDE и этот язык. Необходимо придерживаться наших инструкций». Выбор был бы сделан за вас, и вы будете поражены этой сложностью. Но каждая из тех частей, с которыми вы работаете, существует в историческом контексте и уже является чьей-то идеей. Если вы проанализируете ошибки предшествующих технологий, то сможете лучше понять, как работает данная технология, и скажете себе: «Давай не будем так поступать. Давай сделаем по-другому». Поскольку история вычислений настолько сжата и большая часть этого материала все еще доступна, вы можете вернуться и прочитать то, о чем было написано в выпуске версии 1.0. Дизайнеры расскажут вам, почему они изобрели это и какие проблемы не смогли решить с помощью прежних технологий. Этот метод чрезвычайно полезен для понимания как того, для чего он нужен, так и ограничений, с которыми вы столкнетесь.

Twitter Брайана Гетца — @BrianGoetz

А Глоссарий

В этом приложении даются определения ключевых терминов, использованных в книге.

CSV. Формат обмена текстовыми данными, где строки представляют собой наборы данных, значения которых разделены запятыми, а сами строки обычно разделяются символами новой строки. Аббревиатура *CSV* расшифровывается как *comma-separated values* — значения, разделенные запятыми. CSV — распространенный формат для экспорта данных из электронных таблиц и баз данных (см. главу 7).

NP-трудный. Задача, относящаяся к классу задач, для которых не существует известного алгоритма с полиномиальным временем решения (см. главу 9).

SIMD-инструкции. Инструкции микропроцессора, оптимизированные для выполнения вычислений с помощью векторов, иногда называемые также векторными инструкциями. Аббревиатура *SIMD* расшифровывается как *single instruction, multiple data* — один поток инструкций, много потоков данных (см. главу 7).

XOR. Побитовая логическая операция, которая возвращает `true`, если любой ее операнд имеет значение `True`, но не тогда, когда оба операнда равны `True` или ни один из них не равен `True`. Аббревиатура *XOR* расшифровывается как *exclusive or* — исключающее ИЛИ. В Java для обозначения операции XOR используется оператор `^` (см. главу 1).

Z-оценка. Количество стандартных отклонений единицы данных от среднего значения набора данных (см. главу 6).

Автомемоизация. Вариант *мемоизации*, реализованный на уровне языка, в котором результаты вызовов функций без побочных эффектов сохраняются для использования при последующих идентичных вызовах (см. главу 1).

Ациклический. *Граф* без циклов (см. главу 4).

Бесконечная рекурсия. Множество рекурсивных вызовов, которые не завершаются, а продолжают выполнять дополнительные рекурсивные вызовы. Аналог *бесконечного цикла*. Обычно вызвана отсутствием базового случая (см. главу 1).

Бесконечный цикл. Цикл, который не заканчивается (см. главу 1).

Битовая строка. Структура данных, которая хранит последовательность единиц и нулей, на каждый из которых отводится 1 бит памяти. Иногда битовые строки называют *битовыми векторами* или *битовыми массивами* (см. главу 1).

Вершина. Отдельный узел *графа* (см. главу 4).

Входной слой. Первый слой *искусственной нейронной сети с прямой связью*, который получает входные данные от какого-либо внешнего объекта (см. главу 7).

Выходной слой. Последний слой в искусственной нейронной сети с прямой связью, который используется для определения результата сети для конкретного набора входных данных и конкретной задачи (см. главу 7).

Генетическое программирование. Программы, которые модифицируют сами себя с помощью операторов *отбора*, *кроссинговера* и *мутации*, чтобы найти неочевидные решения задач программирования (см. главу 5).

Глубокое обучение. Еще одно модное слово. Глубоким обучением могут называть любой из нескольких методов, в которых используются передовые алгоритмы машинного обучения для анализа больших данных. Чаще всего глубокое обучение означает применение многослойных *искусственных нейронных сетей* для решения задач с большими наборами данных (см. главу 7).

Градиентный спуск. Метод изменения весов в *искусственной нейронной сети* с использованием *дельт*, рассчитанных во время *обратного распространения*, и *скорости обучения* (см. главу 7).

Граф. Абстрактная математическая конструкция, которая применяется для моделирования реальной задачи путем разделения этой задачи на множество *связных* узлов. Эти узлы называются *вершинами*, а соединения между ними — *ребрами* (см. главу 4).

Декомпрессия. Отмена процесса *компрессии*, возврат данных в исходную форму (см. главу 1).

Дельта. Значение, описывающее разрыв между ожидаемым значением веса в *нейронной сети* и его фактическим значением. Ожидаемое значение определяется с помощью данных *обучения* и *обратного распространения* (см. главу 7).

Дерево. *Граф*, который имеет только один *путь* между любыми двумя вершинами. Дерево является *ациклическим* (см. главу 4).

Дираф. См. *Направленный граф* (см. главу 4).

Динамическое программирование. Вместо решения большой задачи напрямую, методом грубой силы, в динамическом программировании задача разбивается на более мелкие подзадачи, каждой из которых легче управлять (см. главу 9).

Допустимая эвристика. *Эвристика* алгоритма поиска A*, которая никогда не переоценивает затраты на достижение цели (см. главу 2).

Естественный отбор. Процесс эволюции, благодаря которому хорошо адаптированные организмы процветают, а плохо адаптированные терпят неудачу. Учитывая ограниченный набор ресурсов в окружающей среде, те организмы, которые лучше всего подходят для использования этих ресурсов, будут выживать и размножаться. На протяжении нескольких *поколений* это приводит к тому, что полезные черты распространяются в *популяции*, а следовательно, естественным образом отбираются за счет ограничений окружающей среды (см. главу 5).

Жадный алгоритм. Алгоритм, который в любой момент принятия решения делает наилучший немедленный выбор, рассчитывая на то, что это приведет к глобально оптимальному решению (см. главу 4).

Исключающее ИЛИ. См. *XOR* (см. главу 1).

Искусственная нейронная сеть. Симуляция биологической *нейронной сети* с использованием вычислительных инструментов для решения задач, которые не удается легко представить в форме, поддающейся традиционным алгоритмическим подходам. Обратите внимание на то, что работа *искусственной нейронной сети* обычно значительно отличается от работы ее биологического аналога (см. главу 7).

Кластер. См. *Кластеризация* (см. главу 6).

Кластеризация. Методика *неконтролируемого обучения*, при которой набор данных делится на группы связанных точек, известных как *кластеры* (см. главу 6).

Кодон. Комбинация из трех *нуклеотидов*, которые образуют аминокислоту (см. главу 2).

Компрессия. Кодирование (изменение формы) данных, после чего они занимают меньше места (см. главу 1).

Контролируемое обучение. Любая технология машинного обучения, в которой алгоритм каким-либо образом направляется на получение правильных результатов с использованием внешних ресурсов (см. главу 7).

Кроссинговер. В генетическом алгоритме — объединение особей *популяции* для создания потомков, которые получат часть генов родителей и будут частью следующего *поколения* (см. главу 5).

Мемоизация. Методика, в которой результаты вычислительных задач сохраняются и впоследствии при необходимости извлекаются из памяти, экономя время вычислений, которое иначе было бы потрачено на повторное вычисление тех же результатов (см. главу 1).

Минимальное связующее дерево. *Связующее дерево*, которое соединяет все вершины при минимальном общем весе *ребер* (см. главу 4).

Мутация. В генетическом алгоритме — случайное изменение некоторого свойства особи перед тем, как включить ее в состав следующего *поколения* (см. главу 5).

Направленный граф. Также известный как *дираф* — это *граф*, *ребра* которого можно проходить только в одном направлении (см. главу 4).

Нейрон. Отдельная нервная клетка, такая как клетка человеческого мозга (см. главу 7).

Нейронная сеть. Сеть, состоящая из нескольких *нейронов*, которые действуют согласованно для обработки информации. *Нейроны* часто объединяются в слои (см. главу 7).

Неконтролируемое обучение. Любая технология машинного обучения, при которой не используется прогноз, чтобы сделать выводы, — другими словами, технология, которая не является управляемой, а работает сама по себе (см. главу 6).

Нормализация. Процесс, в результате которого различные типы данных становятся сравнимыми (см. главу 6).

Нуклеотид. Экземпляр одного из четырех оснований ДНК: аденина (A), цитозина (C), гуанина (G) и тимина (T) (см. главу 2).

Область определения. Возможные значения *переменной* в задаче с ограничениями (см. главу 3).

Обратное распространение. Методика, используемая для *обучения нейронной сети*, при котором веса назначаются в соответствии с набором входных данных с известными правильными выходными данными. Для вычисления степени ответственности каждого веса за отклонение реальных результатов от ожидаемых применяются частные производные. Эти *дельты* задействуются для обновления весов при последующих прогонах (см. главу 7).

Обучение. Фаза, в которой веса искусственной нейронной сети корректируются посредством обратного распространения с известными правильными выходными данными для некоторых заданных входных данных (см. главу 7).

Ограничение. Требование, которое должно быть выполнено для решения задачи с ограничениями (см. главу 3).

Отбор. Процесс отбора особей в *поколении* при выполнении генетического алгоритма для размножения с целью создания особей следующего *поколения* (глава 5).

Очередь. Абстрактная структура данных, обеспечивающая упорядочение типа FIFO («первым пришел — первым вышел»). Реализация очереди обеспечивает по крайней мере операции `push` и `pop` для добавления и удаления элементов очереди соответственно (см. главу 2).

Очередь с приоритетом. Структура данных, которая выводит элементы на основе приоритетной последовательности. Например, очередь с приоритетом может использоваться для множества экстренных вызовов, чтобы сначала отвечать на вызовы с самым высоким приоритетом (см. главу 2).

Переменная. В контексте задач с ограничениями переменная — это параметр, который должен быть выдержан в заданных рамках как часть решения задачи. Возможные значения переменной — это ее *область определения*. Требованиями к решению являются одно или несколько *ограничений* (см. главу 3).

Позиция. Ход в игре для двух игроков (см. главу 8).

Поиск с возвратом. Возвращение к более раннему моменту принятия решения в задаче поиска, чтобы пойти в другом направлении, если предыдущий поиск зашел в тупик (см. главу 3).

Поколение. Этап при выполнении генетического алгоритма; также используется для обозначения особей *популяции*, активных на данном этапе (см. главу 5).

Популяция. В генетическом алгоритме это совокупность особей, каждая из которых представляет собой потенциальное решение задачи, конкурирующих за решение задачи (см. главу 5).

Путь. Набор *ребер*, соединяющих две вершины *графа* (см. главу 4).

Ребро. Связь между двумя *вершинами* (узлами) *графа* (см. главу 4).

Рекурсивная функция. Функция, которая вызывает саму себя (см. главу 1).

Связность. Свойство графа, которое указывает на то, что существует *путь* от любой *вершины* графа к любой другой его *вершине* (см. главу 4).

Связующее дерево. *Дерево*, которое соединяет все вершины *графа* (см. главу 4).

Сигмоидная функция. Одна из множества популярных *функций активации*, используемых в *искусственных нейронных сетях*. Эпонимическая сигмоидная функция всегда возвращает значение в диапазоне 0...1. Сигмоидная функция полезна также для обеспечения способности сети возвращать не только результаты простых линейных преобразований (см. главу 7).

Синапсы. Разрывы между *нейронами*, в которых высвобождаются нейротрансмиттеры, что позволяет проводить электрический ток. С точки зрения непрофессионала это связи между нейронами (см. главу 7).

Скорость обучения. Значение, обычно постоянное, которое используется для корректировки скорости изменения весов в *искусственной нейронной сети* на основе вычисленных *дельт* (см. главу 7).

Скрытый слой. Любой слой между входным и выходным слоями в искусственной нейронной сети с прямой связью (см. главу 7).

Стек. Абстрактная структура данных, обеспечивающая упорядочение по принципу «последним пришел — первым вышел» (last-in-first-out, LIFO). Реализация стека обеспечивает как минимум операции `push` и `pop` для добавления и удаления элементов соответственно (см. главу 2).

Упреждение. Тип *нейронной сети*, в которой сигналы распространяются в одном направлении (см. главу 7).

Функция активации. Функция, которая преобразует выходной сигнал *нейрона* в *искусственной нейронной сети*, как правило, для того, чтобы эта сеть могла обрабатывать нелинейные преобразования или ограничить выходное значение некоторым диапазоном (см. главу 7).

Функция жизнеспособности. Функция, которая оценивает эффективность потенциального решения задачи (см. главу 5).

Хромосома. В генетическом алгоритме каждая особь *популяции* называется хромосомой (см. главу 5).

Центроид. Центральная точка кластера. Как правило, каждое измерение этой точки является средним значением данного измерения для остальных точек (см. главу 6).

Цикл. *Путь* в *графе*, который дважды проходит через одну и ту же вершину без *поиска с возвратом* (см. главу 4).

Эвристика. Интуитивная догадка о том, в каком направлении следует двигаться, чтобы решить задачу (см. главу 2).

Б Дополнительные ресурсы

Что дальше? Книга охватывает широкий спектр тем, а изучить их глубже вам помогут отличные ресурсы, ссылки на которые содержит это приложение.

JAVA

Как говорилось во введении, предполагается, что вы знаете язык Java хотя бы на среднем уровне. Здесь я привожу книгу по Java, которой пользовался сам, и рекомендую ее, чтобы вы поднялись на следующий уровень знания языка. За последние несколько лет Java претерпел значительные изменения. Рекомендую ознакомиться с последними разработками языка Java — это поможет совершенствовать ваши навыки.

- *Urma R.-G., Fusco M., Mycroft A.* Modern Java in Action. — Manning, 2018, www.manning.com/books/modern-java-in-action:
  - охватывает темы лямбда-выражений, потоки и современные функциональные механизмы в Java;
  - в примерах используется последняя LTS (долгосрочная поддержка) версия Java — 11;
  - охватывает широкий спектр расширенных возможностей Java, что поможет многим разработчикам, изучавшим этот язык в эпоху до Java 8.

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Процитирую введение к этой книге: «Это не учебник по структурам данных и алгоритмам». В этой книге редко используется нотация О большого (big-O notation) и нет математических доказательств. Это скорее практическое руководство по важным методикам программирования. Поэтому есть смысл обзавестись и настоящим учебником, который не только даст вам более формальное объяснение того, почему работают те или иные методы, но и послужит полезным справочным пособием. Онлайн-ресурсы хороши, но иногда полезно иметь информацию, тщательно проверенную учеными и издателями.

- *Cormen T., Leiserson C., Rivest R., Stein C.* Introduction to Algorithms, 3rd ed. — MIT Press, 2009, https://mitpress.mit.edu/books/ introduction-algorithms-third-edition (*Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К.* Алгоритмы: построение и анализ. 3-е изд. — М.: Вильямс, 2013):
  - один из самых часто цитируемых текстов в области информатики, настолько исчерпывающий, что на него часто ссылаются по инициалам его авторов — CLRS;
  - комплексное и строгое изложение;
  - стиль преподнесения материала делает его не столь доступным, как другие тексты, тем не менее он является отличным справочным материалом;
  - большинство алгоритмов описано на псевдокоде;
  - сейчас готовится к выходу четвертое издание, и поскольку эта книга стоит дорого, возможно, стоит подождать его выхода.
- *Sedgewick R., Wayne K.* Algorithms. 4th ed. — Addison-Wesley Professional, 2011, http:// algs4.cs.princeton.edu/home/ (*Седжвик Р., Уэйн К.* Алгоритмы на Java. 4-е изд. — М.: Вильямс, 2013):
  - доступное и вместе с тем всеобъемлющее введение в алгоритмы и структуры данных;
  - хорошо организованный материал с полными примерами всех алгоритмов на Java;
  - алгоритмы классов, широко используемые в учебных курсах колледжей.
- *Skiena S.* The Algorithm Design Manual. 2nd ed. — Springer, 2011, http://www.algorist.com:
  - своим подходом эта книга отличается от других учебников по данной дисциплине;
  - в книге содержится не очень много кода, акцент сделан на наглядном обсуждении правильного использования каждого алгоритма;
  - читателю советуют, как найти собственный путь в области применения широкого круга алгоритмов.

- *Bhargava A.* Grokking Algorithms. — Manning, 2016, https://www.manning.com/books/grokking-algorithms (*Бхаргава А.* Грокаем алгоритмы: иллюстрированное пособие для программистов и любопытствующих. — СПб.: Питер, 2020):
  - графический подход к освоению основных алгоритмов, имеются забавные анимационные ролики, которые можно загрузить;
  - это не справочник, а руководство для тех, кто впервые приступает к изучению отдельных важных тем;
  - наглядные материалы и легкое для понимания пособие;
  - пример кода на Python.

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Искусственный интеллект меняет наш мир. В этой книге вы познакомились не только с некоторыми традиционными технологиями поиска, применяемыми искусственным интеллектом, такими как A* и минимакс, но и с методиками из столь захватывающей субдисциплины, как машинное обучение, такими как *k*-средние и нейронные сети. Изучить искусственный интеллект глубже очень интересно, а еще это позволит подготовиться к следующей волне развития компьютерных технологий.

- *Russell S., Norvig P.* Artificial Intelligence: A Modern Approach. — 3rd ed. — Pearson, 2010, http://aima.cs.berkeley.edu (*Рассел С., Норвиг П.* Искусственный интеллект: современный подход. — 3-е изд. — М.: Вильямс, 2019):
  - всеобъемлющий учебник по ИИ, часто используемый в учебных курсах колледжей;
  - широкий охват тем;
  - превосходный репозиторий исходного кода (реализованные версии алгоритмов, описанных в книге на псевдокоде), доступный онлайн.
- *Lucci S., Kopec D.* Artificial Intelligence in the 21st Century. — 2nd ed. — Mercury Learning and Information, 2015, http://mng.bz/1N46:
  - изложение доступно для тех, кто ищет более практичное и красочное руководство, чем у Рассела и Норвига;
  - интересные зарисовки для практиков и множество ссылок на реальные приложения.
- *Ng A.* Machine Learning, учебный курс (Стэнфордский университет), https://www.coursera.org/learn/machine-learning/:
  - бесплатный онлайн-курс, который охватывает многие фундаментальные алгоритмы машинного обучения, в изложении всемирно известного эксперта;

- обучение от всемирно известного эксперта;
- многие практики считают этот курс отличной отправной точкой в данной области.

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

На Java можно программировать в функциональном стиле, но в общем-то язык для этого не предназначен. Можно углубиться в функциональное программирование на самом Java, но может быть полезно работать и на чисто функциональном языке, а затем перенести некоторые идеи, которые вы извлекли из этого опыта, обратно в Java.

- *Abelson H., Sussman G. J., Sussman J.* Structure and Interpretation of Computer Programs — MIT Press, 1996, https://mitpress.mit.edu/sicp/:
  - классическое введение в функциональное программирование, часто используемое во вводных курсах по информатике;
  - четко структурированный, простой для освоения, чисто функциональный язык программирования;
  - книга бесплатная, доступна онлайн.
- *Płachta M.* Grokking Functional Programming. — Manning, 2021, www.manning.com/books/grokking-functional-programming:
  - графически оформленное понятное введение в функциональное программирование.
- *Saumont P.-Y.* Functional Programming in Java. — Manning, 2017, www.manning.com/books/functional-programming-in-java:
  - книга представляет собой базовое введение в некоторые утилиты функционального программирования из стандартной библиотеки Java;
  - демонстрирует функциональное использование Java.

Дэвид Копец

Классические задачи Computer Science на языке Java

Перевел с английского *А. Павлов*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*Н. Терех, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 20.08.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 23,220. Тираж 1000. Заказ 0000.

Классические задачи Computer Science на языке Java

Дэвид Копец

MANNING