Взвешенные графы

В предыдущей главе было показано, что ребра графа могут иметь направление. В этой главе рассматривается другая характеристика ребер: вес. Например, если вершины взвешенного графа соответствуют городам, то вес ребер может представлять расстояние между городами, стоимость перелета между ними или количество ежегодных автомобильных поездок (важно для проектировщиков дорожной сети).

При включении весов как характеристики ребер графа появляется ряд интересных и сложных вопросов. Что представляет собой минимальное остовное дерево взвешенного графа? Как вычислить наименьшее (или наименее затратное) расстояние от одной вершины к другой? Такие вопросы находят важное практическое применение в реальном мире.

Глава начинается с рассмотрения взвешенных, но не направленных графов и их минимальных остовных деревьев. Вторая половина главы посвящена графам, которые одновременно являются направленными и взвешенными, в контексте известного алгоритма Дейкстры, предназначенного для определения кратчайшего пути от одной вершины к другой.

Минимальное остовное дерево во взвешенных графах

Чтобы дать представление об особенностях взвешенных графов, мы вернемся к теме минимальных остовных деревьев. Для взвешенного графа построить такое дерево несколько сложнее, чем для невзвешенного. Когда все ребра имеют одинаковый вес, задача решается относительно тривиально (как было показано в главе 13, «Графы») — алгоритм легко выбирает ребро, включаемое в минимальное остовное дерево. Но если ребра имеют разные веса, правильный выбор потребует некоторых вычислений.

Пример: кабельное телевидение в джунглях

Предположим, мы хотим проложить линию кабельного телевидения, соединяющую шесть городов некоей вымышленной страны. Для соединения шести городов потребуется пять сегментов, но каких именно? Затраты на соединение разных

пар городов зависят от конкретной пары, поэтому мы должны тщательно выбрать маршрут для минимизации общих затрат.

На рис. 14.1 изображен взвешенный граф с шестью вершинами, представляющими города. Рядом с каждым ребром указан его вес. Будем считать, что эти числа представляют затраты на прокладку кабеля между городами. Обратите внимание: для некоторых пар прокладка кабеля невозможна из-за сложного рельефа или большого расстояния. Мы будем считать, что от A до C или от D до E слишком далеко, поэтому такие сегменты не нужно ни рассматривать, ни отображать их на графе.

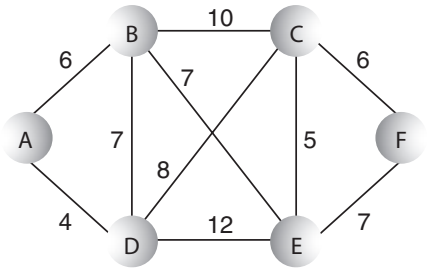

Рис. 14.1. Взвешенный граф

Как выбрать маршрут с минимальными общими затратами на прокладку кабеля? Для этого следует построить минимальное остовное дерево. Оно состоит из пяти сегментов (на единицу меньше количества городов), соединяет все шесть городов, а также обладает минимальными суммарными затратами. Сможете ли вы предложить такой маршрут по внешнему виду графа на рис. 14.1? Если не сможете, для решения задачи можно воспользоваться приложением GraphW Workshop.

Приложение GraphW Workshop

Приложение GraphW Workshop похоже на GraphN и GraphD, но оно создает взвешенные ненаправленные графы. Перед созданием ребра необходимо ввести его вес в текстовом поле, находящемся в правом верхнем углу. Приложение реализует только один алгоритм: при повторных нажатиях кнопки `Tree` оно определяет минимальное остовное дерево для созданного графа. Кнопки `New` и `View`, как и в предыдущих приложениях, предназначены для уничтожения старого графа и просмотра матрицы смежности.

Поэкспериментируйте с приложением, создайте небольшие графы и определите их минимальные остовные деревья. (В некоторых конфигурациях придется повозиться с расположением вершин, чтобы веса не накладывались друг на друга.)

В процессе работы алгоритма вершины, включаемые в минимальное остовное дерево, выделяются красным контуром, а ребра становятся толще. Вершины, находящиеся в дереве, также перечисляются слева под графом. Справа выводится содержимое приоритетной очереди. Например, элемент AB6 в очереди обозначает ребро из A в B с весом 6. Функции приоритетной очереди будут рассмотрены после примера алгоритма.

Создайте в приложении GraphW Workshop граф на рис. 14.1. Примерный вид результата показан на рис. 14.2.

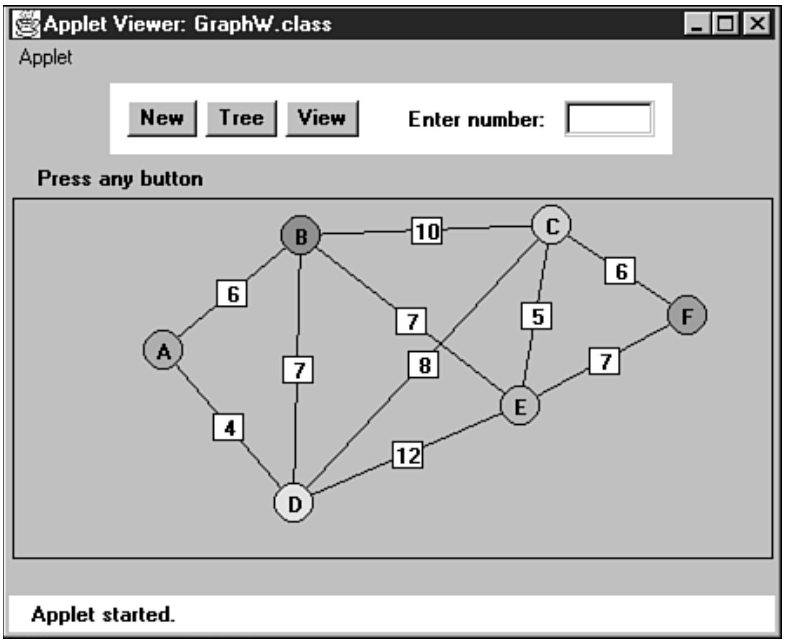

Рис. 14.2. Приложение GraphW Workshop

Постройте минимальное остовное дерево графа, выполняя алгоритм в пошаговом режиме (кнопка `Tree`). В результате должно получиться дерево, изображенное на рис. 14.3.

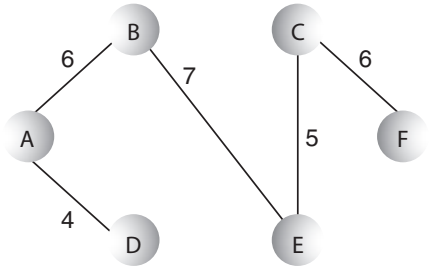

Рис. 14.3. Минимальное остовное дерево

Приложение должно обнаружить, что минимальное остовное дерево состоит из ребер AD, AB, BE, EC и CF с суммарным весом 28. Порядок перечисления ребер не важен. Если начать с другой вершины, получится дерево из тех же ребер, но в другом порядке.

Рассылка инспекторов

Алгоритм построения минимального остовного дерева непрост, поэтому для его рассмотрения мы воспользуемся аналогией с кабельным телевидением. В фирме кроме начальника (это, конечно, вы) работают еще несколько топографов-инспекторов.

Компьютерный алгоритм не знает всех данных задачи и не умеет представить себе общую картину. Он накапливает данные постепенно, изменяя свое представление о ситуации по ходу дела. При работе с графами алгоритмы обычно начи-

нают с определенной вершины и продвигаются «наружу», получая информацию о близлежащих вершинах раньше информации о дальних вершинах. Примеры такого подхода уже встречались нам в описаниях обхода в глубину и в ширину (см. предыдущую главу).

Аналогичным образом мы будем считать, что затраты на прокладку кабеля между всеми городами неизвестны заранее. Получение этой информации требует времени, и в этом нам помогут инспекторы.

Начинаем с вершины A

Для начала мы открываем офис в городе A (впрочем, начать можно с любого города). Из A доступны только два города: B и D (см. рис. 14.1). Вы отправляете двух инспекторов по опасным лесным тропам: первый идет в B, а второй в D. Задача инспекторов — определить стоимость прокладки кабеля по этим маршрутам.

Первый инспектор прибывает в B, завершив осмотр местности, и звонит вам; он говорит, что прокладка кабеля между A и B обойдется в 6 миллионов долларов. Второй инспектор чуть позднее сообщает, что сегмент A-D, проходящий по равнинной части страны, будет стоить только 4 миллиона. Вы составляете список:

- A-D, 4 миллиона.
- A-B, 6 миллионов.

Сегменты всегда перечисляются в порядке возрастания стоимости; причины вскоре будут понятны.

Прокладка сегмента A-D

На этой стадии можно отправлять строителей на прокладку кабеля из A в D. Почему мы уверены в том, что маршрут A-D станет частью самого экономичного решения (минимального остовного дерева)? Пока нам известна стоимость всего двух сегментов системы. Разве нам не нужна дополнительная информация?

Попробуйте представить, что какой-то другой маршрут, связывающий A с D, окажется дешевле прямого соединения. Если он не идет непосредственно в D, то этот другой маршрут должен пройти через B, а может быть, и через другие города. Но мы уже знаем, что сегмент в B (6) стоит дороже сегмента в D (4). Таким образом, даже если остальные сегменты этого гипотетического кругового маршрута будут дешевыми (рис. 14.4), переход в D через B все равно обойдется дороже. Кроме того, переход к городам на обходном пути (X) через B обойдется дороже, чем переход через D.

Итак, мы делаем вывод, что сегмент A-D должен быть частью минимального остовного дерева. Заодно это наводит на мысль, что в качестве отправной точки лучше всего выбрать самый дешевый сегмент (формальное доказательство выходит за рамки книги). Мы строим сегмент A-D и открываем офис в D.

Зачем нужен офис? По местному законодательству отправка инспекторов в близлежащие города возможна только при наличии офиса. А в контексте теории графов это означает, что вершина должна быть включена в дерево перед определением весов ребер, выходящих из этой вершины. Все города с офисами соединены друг с другом проложенными кабелями; города без офисов еще не подключены к сети.

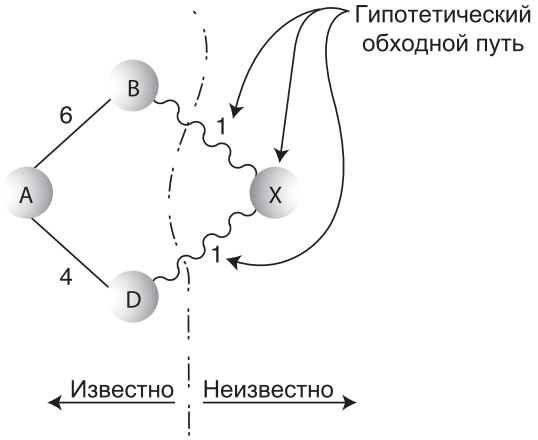

Рис. 14.4. Гипотетический обходной путь

Прокладка сегмента A-B

После завершения сегмента A-D и открытия офиса в D можно отправить инспекторов по всем городам, доступным из D. Это города B, C и E. Инспекторы добираются до места назначения и сообщают стоимость работ: 7, 8 и 12 миллионов соответственно. (Конечно, отправлять инспектора в A не нужно, потому что мы уже проанализировали маршрут A-D и проложили кабель.)

Теперь мы знаем стоимость прокладки четырех сегментов из городов с офисами в города без офисов:

- A-B, 6 миллионов.
- D-B, 7 миллионов.
- D-C, 8 миллионов.
- D-E, 12 миллионов.

Почему в списке нет сегмента A-D? Потому что кабель между этими городами уже проложен; этот участок сети можно больше не рассматривать. Маршрут, по которому был проложен канал, всегда удаляется из списка.

Что же делать дальше? Имеется множество потенциальных сегментов. Ваши дальнейшие действия должны определяться следующим правилом.

ПРАВИЛО

Из списка всегда выбирается ребро с наименьшим весом.

Вообще говоря, мы уже следовали этому правилу при выборе маршрута из A; ребро A-D было самым «дешевым». В текущей ситуации наименьшим весом обладает ребро A-B, поэтому мы прокладываем кабель из A в B за 6 миллионов и открываем офис в B.

Здесь необходимо сделать одно общее замечание. В любой момент при проектировании кабельной сети существуют три категории городов:

1. Города, в которых имеются офисы, связанные кабелем. (В терминологии графов — вершины, входящие в минимальное остовное дерево.)

2. Города, еще не подключенные к сети и не имеющие офисов, но для которых известна стоимость соединения как минимум с одним городом, в котором существует офис. Мы будем называть их «периферийными» городами.
3. Города, о которых ничего не известно.

На данный момент A, D и B относятся к категории 1, C и E — к категории 2, а F — к категории 3 (рис. 14.5). В ходе работы алгоритма города переходят из категории 3 в 2, а из категории 2 в категорию 1.

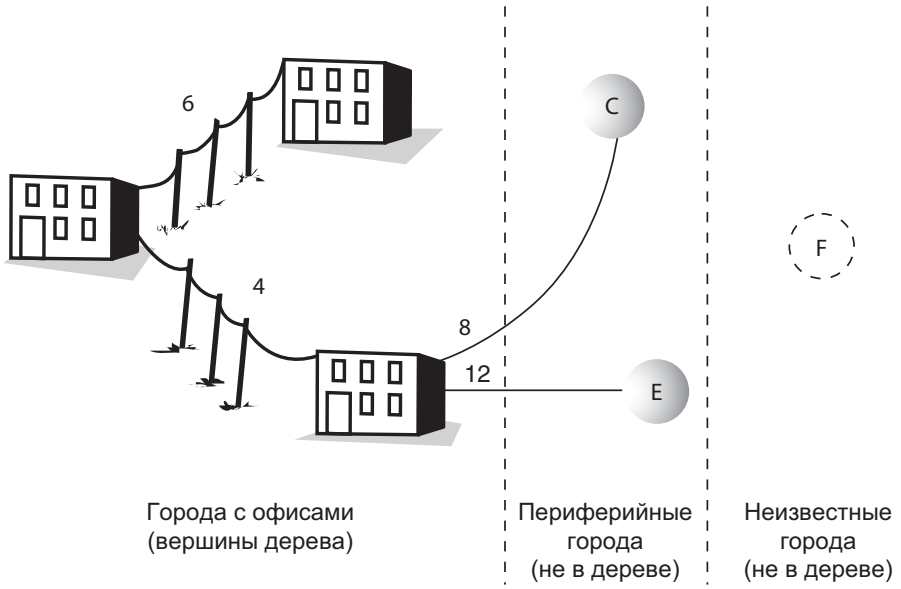

Рис. 14.5. Промежуточная стадия построения минимального остовного дерева

Прокладка сегмента B-E

К этому моменту города A, D и B подключены к кабельной сети и в них открыты офисы. Мы уже знаем стоимость прокладки кабеля из A и D в города категории 2, но для B затраты пока неизвестны. Инспекторы из B отправляются в C и E. От них приходит информация: прокладка кабеля в C обойдется в 10 миллионов, а в E — 7 миллионов. Новый список выглядит так:

- B-E, 7 миллионов.
- D-C, 8 миллионов.
- B-C, 10 миллионов.
- D-E, 12 миллионов.

Сегмент D-B был в предыдущем списке, но в новом варианте списка его нет. Как упоминалось ранее, бессмысленно прокладывать сегменты к уже подключенным городам.

Из списка видно, что наименьшим весом обладает маршрут B-E (7). Вы отправляете рабочих на прокладку кабеля и открываете офис в E (см. рис. 14.3).

Прокладка сегмента E-C

Из E инспекторы сообщают, что стоимость прокладки кабеля в C составляет 5 миллионов, а в F — 7 миллионов. Сегмент D-E удаляется из списка, так как город E уже подключен. Обновленный список выглядит так:

- E-C, 5 миллионов.
- E-F, 7 миллионов.
- D-C, 8 миллионов.
- B-C, 10 миллионов.

Наименьшей стоимостью обладает сегмент E-C. Вы строите его и открываете офис в C.

Последний сегмент C-F

Количество вариантов постоянно сокращается. После удаления подключенных городов в списке остаются только два сегмента:

- C-F, 6 миллионов.
- E-F, 7 миллионов.

Вы прокладываете последний сегмент кабеля из C в F, открываете офис в F — работа завершена. В каждом городе имеется офис, а между городами проложена кабельная сеть A-D, A-B, B-E, E-C и C-F (см. рис. 14.3). У нас получилась сеть, соединяющая все шесть городов с минимальными затратами.

Создание алгоритма

На примере прокладки сети кабельного телевидения были представлены основные принципы построения минимальных остовных деревьев для взвешенных графов. Теперь посмотрим, как алгоритмизируется этот процесс.

Приоритетная очередь

Как видно из примера с кабельной сетью, основной операцией в этом алгоритме является ведение списка стоимости сегментов между парами городов. Выбор сегмента с минимальной стоимостью определял место прокладки следующего участка.

Для списка, из которого требуется раз за разом выбирать наименьшее значение, логично использовать приоритетную очередь. Оказывается, эта структура данных весьма эффективна при построении минимального остовного дерева. В серьезной программе приоритетная очередь может быть реализована на базе пирамиды (см. главу 12, «Пирамиды»); это ускорит выполнение операций с большими очередями. В нашей демонстрационной программе будет использоваться приоритетная очередь на базе простого массива.

Общая схема алгоритма

Сформулируем алгоритм в терминологии графов (вместо терминологии кабельной сети).

Начать с вершины, включить ее в дерево. Затем многократно выполнять следующие действия:

1. Найти все ребра от самой новой вершины к другим вершинам, не входящим в дерево. Поместить эти ребра в приоритетную очередь.
2. Выбрать ребро с наименьшим весом. Включить его и вершину, к которой оно ведет, в дерево.

Действия повторяются до тех пор, пока все вершины не будут включены в дерево. При выполнении этого условия работа алгоритма завершается.

На шаге 1 под «самой новой» понимается вершина, которая была последней добавлена в дерево. Поиск ребер для этого шага осуществляется в матрице смежности. После шага 1 в списке содержатся все ребра, ведущие от вершин в дереве к периферийным вершинам.

Избыточные ребра

При составлении списка мы позаботились об удалении сегментов, ведущих к уже соединенным городам. Без этого в сети могли бы появиться лишние сегменты.

В реализации алгоритма также следует проследить за исключением из приоритетной очереди ребер, ведущих к вершинам, которые уже включены в дерево. Можно просматривать содержимое очереди и удалять все такие ребра каждый раз, когда в дерево добавляется новая вершина. Но как выясняется, проще в любой момент времени хранить в приоритетной очереди только одно ребро к заданной периферийной вершине. Другими словами, очередь должна содержать только одно ребро, ведущее к каждой вершине категории 2.

Именно это решение используется в приложении GraphW Workshop. Приоритетная очередь содержит меньше ребер, чем можно ожидать — всего один элемент для каждой вершины категории 2. Пройдите по минимальному остовному дереву для рис. 14.1 и убедитесь в том, что это действительно так. В таблице 14.1 показано, как ребра с повторяющимися «конечными точками» удаляются из приоритетной очереди.

Таблица 14.1. Отсечение ребер

Номер шага	Полный список ребер	Усеченный список (в приоритетной очереди)	Дубликаты, удаленные из приоритетной очереди
1	AB6, AD4	AB6, AD4	
2	DE12, DC8, DB7, AB6	DE12, DC8, AB6	DB7(AB6)
3	DE12, BC10, DC8, BE7	DC8, BE7	DE12(BE7), BC10(DC8)
4	BC10, DC8, EF7, EC5	EF7, EC5	BC10(EC5), DC8(EC5)
5	EF7, CF6	CF6	EF7

Напомним, что ребро определяется начальной вершиной, конечной вершиной и весом. Второй столбец таблицы соответствует тем спискам, которые мы вели при построении кабельной сети. В нем перечисляются все ребра от вершин категории 1 (входящих в дерево) к вершинам категории 2 (для которых известно хотя бы одно ребро, ведущее от вершины категории 1).

В третьем столбце приводится содержимое приоритетной очереди при выполнении приложения GraphW. Из очереди удаляются все ребра с такой же конечной вершиной, как у другого ребра, но с бóльшим весом.

В четвертом столбце перечислены удаленные ребра; в круглых скобках указаны ребра с меньшим весом, которые остались в очереди вместо них. При переходе к следующему шагу из списка всегда исключается последний элемент, потому что соответствующее ребро включается в очередь.

Поиск дубликатов в приоритетной очереди

Как сделать так, чтобы для каждой вершины категории 2 в очереди было только одно ребро? Каждый раз, когда ребро включается в очередь, мы убеждаемся в отсутствии другого ребра, ведущего к той же вершине. Если такое ребро существует, то из двух ребер остается только одно, имеющее минимальный вес.

Из этого следует, что алгоритм должен проверять приоритетную очередь элемент за элементом в поисках ребер-дубликатов. Произвольный доступ в приоритетных очередях выполняется неэффективно, однако в данной ситуации эта операция, противоречащая «духу» приоритетной очереди, необходима.

Реализация на языке Java

Метод построения минимального остовного дерева для взвешенного графа `mstw()` работает по описанному выше алгоритму. Как и в других программах, работающих с графами, список вершин хранится в массиве `vertexList[]`, а первая вершина хранится в ячейке с индексом 0. Переменная `currentVert` представляет вершину, которая была последней включена в дерево.

```
public void mstw()           // Построение минимального остовного дерева
   {
   currentVert = 0;          // Начиная с ячейки 0

   while(nTree < nVerts-1)   // Пока не все вершины включены в дерево
      {                      // Включение currentVert в дерево
      vertexList[currentVert].isInTree = true;
      nTree++;

      // Вставка в приоритетную очередь ребер, смежных с currentVert
      for(int j=0; j<nVerts; j++)   // Для каждой вершины
         {
         if(j==currentVert)         // Пропустить, если текущая вершина
            continue;
         if(vertexList[j].isInTree) // Пропустить, если уже в дереве
```

```
            continue;
         int distance = adjMat[currentVert][j];
         if( distance == INFINITY)  // Пропустить, если нет ребер
            continue;
         putInPQ(j, distance);      // Поместить в приоритетную очередь
         }
      if(thePQ.size()==0)           // Очередь не содержит вершин?
         {
         System.out.println(" GRAPH NOT CONNECTED");
         return;
         }
         // Удаление ребра с минимальным расстоянием из очереди
         Edge theEdge = thePQ.removeMin();
         int sourceVert = theEdge.srcVert;
         currentVert = theEdge.destVert;

         // Вывод ребра от начальной до текущей вершины
         System.out.print( vertexList[sourceVert].label );
         System.out.print( vertexList[currentVert].label );
         System.out.print(" ");
         }

      // Минимальное остовное дерево построено
      for(int j=0; j<nVerts; j++)     // Снятие пометки с вершин
         vertexList[j].isInTree = false;
      }
```

Алгоритм выполняется в цикле `while`, который завершается с включением всех вершин в дерево. В цикле выполняются следующие действия:

1. Текущая вершина включается в дерево.
2. Ребра, смежные с этой вершиной, включаются в приоритетную очередь (если требуется).
3. Ребро с минимальным весом исключается из приоритетной очереди. Конечная вершина этого ребра становится текущей вершиной.

Рассмотрим все эти действия более подробно. На шаге 1 в дерево включается вершина `currentVert` (посредством установки флага `isInTree`).

На шаге 2 вершины, смежные с этой вершиной, анализируются в качестве кандидатов на вставку в приоритетную очередь. Ребра перебираются по элементам строки матрицы смежности, номер которой равен `currentVert`. Ребро включается в очередь, если не выполняется ни одно из следующих условий:

- Начальная вершина совпадает с конечной.
- Конечная вершина включена в дерево.
- Не существует ребра, ведущего к этой вершине.

Если ни одно из этих условий не выполняется, то вызывается метод `putInPQ()` для включения ребра в приоритетную очередь. Впрочем, как мы вскоре убедимся, этот метод тоже не всегда включает ребро в очередь.

На шаге 3 ребро с минимальным весом исключается из приоритетной очереди. Это ребро и его конечная вершина включаются в дерево, а начальная (currentVert) и приемная вершины выводятся приложением.

В конце выполнения mstw() вершины исключаются из дерева, для чего метод сбрасывает их переменные isInTree. Этот шаг не является строго обязательным, потому что на основе данных строится только одно дерево. И все же хороший стиль программирования рекомендует возвращать данные к исходному состоянию после их использования.

Как упоминалось ранее, приоритетная очередь должна содержать только одно ребро с заданной конечной вершиной. Метод putInPQ() обеспечивает выполнение этого условия. Он вызывает метод find() класса PriorityQ, адаптированный для поиска ребра с заданной конечной вершиной. Если такого ребра нет, а метод find() соответственно возвращает –1, то метод putInPQ() просто вставляет ребро в приоритетную очередь. Но если такое ребро существует, putInPQ() проверяет, какое из двух ребер — существующее или новое — обладает меньшим весом. Если вес меньше у старого ребра, изменения не требуются, а если у нового — старое ребро выводится из очереди, а новое ребро включается в нее. Код метода putInPQ():

```
public void putInPQ(int newVert, int newDist)
   {
   // Существует ли другое ребро с той же конечной вершиной?
   int queueIndex = thePQ.find(newVert);  // Получение индекса
   if(queueIndex != -1)              // Если ребро существует,
      {                              // получить его
      Edge tempEdge = thePQ.peekN(queueIndex);
      int oldDist = tempEdge.distance;
      if(oldDist > newDist)          // Если новое ребро короче,
         {
         thePQ.removeN(queueIndex);  // удалить старое ребро
         Edge theEdge = new Edge(currentVert, newVert, newDist);
         thePQ.insert(theEdge);      // Вставка нового ребра
         }
      // Иначе ничего не делается; оставляем старую вершину
      }
   else  // Ребра с той же конечной вершиной не существует
      {                              // Вставка нового ребра
      Edge theEdge = new Edge(currentVert, newVert, newDist);
      thePQ.insert(theEdge);
      }
   }
```

Программа mstw.java

Класс PriorityQ использует массив для хранения своих элементов. Как упоминалось ранее, в программе, работающей с большими графами, вместо массива лучше использовать пирамиду. Класс PriorityQ был дополнен вспомогательными методами. Уже упоминавшийся метод find() ищет ребро с заданной конечной вершиной. Метод peekN() читает значение произвольного элемента, а метод removeN() удаляет

произвольный элемент. В остальном код класса выглядит уже знакомо. В листинге 14.1 приведен полный код программы `mstw.java`.

Листинг 14.1. Программа mstw.java

```
// mstw.java
// Построение минимального остовного дерева для взвешенного графа
// Запуск программы: C>java MSTWApp
////////////////////////////////////////////////////////////////
class Edge
   {
   public int srcVert;   // Индекс начальной вершины ребра
   public int destVert;  // Индекс конечной вершины ребра
   public int distance;  // Расстояние от начала до конца
// --------------------------------------------------------------
   public Edge(int sv, int dv, int d)  // Конструктор
      {
      srcVert = sv;
      destVert = dv;
      distance = d;
      }
// --------------------------------------------------------------
   }  // Конец класса Edge
////////////////////////////////////////////////////////////////
   class PriorityQ
      {
      // Массив отсортирован по убыванию от ячейки 0 до size-1
      private final int SIZE = 20;
      private Edge[] queArray;
      private int size;
// --------------------------------------------------------------
   public PriorityQ()            // Конструктор
      {
      queArray = new Edge[SIZE];
      size = 0;
      }
// --------------------------------------------------------------
   public void insert(Edge item)  // Вставка элемента в порядке сортировки
      {
      int j;

      for(j=0; j<size; j++)          // Поиск места для вставки
         if( item.distance >= queArray[j].distance )
            break;

      for(int k=size-1; k>=j; k--)    // Перемещение элементов вверх
         queArray[k+1] = queArray[k];

      queArray[j] = item;             // Вставка элемента
      size++;
      }
```

продолжение ⇩

Листинг 14.1 (*продолжение*)

```
// ---------------------------------------------------------
   public Edge removeMin()            // Извлечение наименьшего элемента
      { return queArray[--size]; }
// ---------------------------------------------------------
   public void removeN(int n)         // Удаление элемента в позиции N
      {
      for(int j=n; j<size-1; j++)     // Перемещение элементов вниз
         queArray[j] = queArray[j+1];
      size--;
      }
// ---------------------------------------------------------
   public Edge peekMin()          // Чтение наименьшего элемента
      { return queArray[size-1]; }
// ---------------------------------------------------------
   public int size()              // Получение количества элементов
      { return size; }
// ---------------------------------------------------------
   public boolean isEmpty()      // true, если очередь пуста
      { return (size==0); }
// ---------------------------------------------------------
   public Edge peekN(int n)      // Чтение элемента в позиции N
      { return queArray[n]; }
// ---------------------------------------------------------
   public int find(int findDex)  // Поиск элемента с заданным
      {                          // значением destVert
      for(int j=0; j<size; j++)
         if(queArray[j].destVert == findDex)
            return j;
      return -1;
      }
// ---------------------------------------------------------
   }  // Конец класса PriorityQ
////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean isInTree;
// ---------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      isInTree = false;
      }
// ---------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
```

```
   private final int INFINITY = 1000000;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private int currentVert;
   private PriorityQ thePQ;
   private int nTree;           // Количество вершин в дереве
// -------------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)   // заполняется нулями
            adjMat[j][k] = INFINITY;
      thePQ = new PriorityQ();
   }
// -------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end, int weight)
      {
      adjMat[start][end] = weight;
      adjMat[end][start] = weight;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void mstw()        // Построение минимального остовного дерева
      {
      currentVert = 0;          // Начиная с ячейки 0

      while(nTree < nVerts-1)   // Пока не все вершины включены в дерево
         {                      // Включение currentVert в дерево
         vertexList[currentVert].isInTree = true;
         nTree++;

         // Вставка в приоритетную очередь ребер, смежных с currentVert
         for(int j=0; j<nVerts; j++)   // Для каждой вершины
            {
            if(j==currentVert)         // Пропустить, если текущая вершина
```

продолжение ⇩

Листинг 14.1 (*продолжение*)

```
            continue;
         if(vertexList[j].isInTree) // Пропустить, если уже в дереве
            continue;
         int distance = adjMat[currentVert][j];
         if( distance == INFINITY)  // Пропустить, если нет ребер
            continue;
         putInPQ(j, distance);      // Поместить в приоритетную очередь
         }
      if(thePQ.size()==0)           // Очередь не содержит вершин?
         {
         System.out.println(" GRAPH NOT CONNECTED");
         return;
         }
         // Удаление ребра с минимальным расстоянием из очереди
         Edge theEdge = thePQ.removeMin();
         int sourceVert = theEdge.srcVert;
         currentVert = theEdge.destVert;

         // Вывод ребра от начальной до текущей вершины
         System.out.print( vertexList[sourceVert].label );
         System.out.print( vertexList[currentVert].label );
         System.out.print(" ");
         }

      // Минимальное остовное дерево построено
      for(int j=0; j<nVerts; j++)     // Снятие пометки с вершин
         vertexList[j].isInTree = false;
      }
// --------------------------------------------------------------
   public void putInPQ(int newVert, int newDist)
      {
      // Существует ли другое ребро с той же конечной вершиной?
      int queueIndex = thePQ.find(newVert);  // Получение индекса
      if(queueIndex != -1)              // Если ребро существует,
         {                              // получить его
         Edge tempEdge = thePQ.peekN(queueIndex);
         int oldDist = tempEdge.distance;
         if(oldDist > newDist)          // Если новое ребро короче,
            {
            thePQ.removeN(queueIndex);  // удалить старое ребро
            Edge theEdge = new Edge(currentVert, newVert, newDist);
            thePQ.insert(theEdge);      // Вставка нового ребра
            }
         // Иначе ничего не делается; оставляем старую вершину
         }
      else  // Ребра с той же конечной вершиной не существует
         {                              // Вставка нового ребра
         Edge theEdge = new Edge(currentVert, newVert, newDist);
```

```
            thePQ.insert(theEdge);
            }
         }
// -----------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class MSTWApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (исходная вершина)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addVertex('F');    // 5

      theGraph.addEdge(0, 1, 6);  // AB  6
      theGraph.addEdge(0, 3, 4);  // AD  4
      theGraph.addEdge(1, 2, 10); // BC 10
      theGraph.addEdge(1, 3, 7);  // BD  7
      theGraph.addEdge(1, 4, 7);  // BE  7
      theGraph.addEdge(2, 3, 8);  // CD  8
      theGraph.addEdge(2, 4, 5);  // CE  5
      theGraph.addEdge(2, 5, 6);  // CF  6
      theGraph.addEdge(3, 4, 12); // DE 12
      theGraph.addEdge(4, 5, 7);  // EF  7

      System.out.print("Minimum spanning tree: ");
      theGraph.mstw();            // Минимальное остовное дерево
      System.out.println();
      }
   }  // Конец класса MSTWApp
////////////////////////////////////////////////////////////////
```

Метод `main()` класса `MSTWApp` создает дерево, изображенное на рис. 14.1. Выходные данные выглядят так:

```
Minimum spanning tree: AD AB BE EC CF
```

Задача выбора кратчайшего пути

Пожалуй, самой распространенной задачей, связанной с взвешенными графами, является задача выбора кратчайшего пути между двумя вершинами. Решение этой задачи находит практическое применение во множестве реальных ситуаций, от проектирования печатных плат до планирования проектов. Эта задача превосходит по сложности то, что мы делали раньше, поэтому мы начнем с рассмотрения примера.

Железная дорога

Действие происходит все в той же вымышленной стране, но на этот раз нас интересует не кабельное телевидение, а железная дорога. Впрочем, новый проект не столь грандиозен, как предыдущий — железную дорогу строить не нужно; она уже построена. Мы всего лишь хотим найти самый экономичный маршрут из одного города в другой.

Стоимость билета между любыми двумя городами является фиксированной величиной. Данные представлены на рис. 14.6. Таким образом, поездка из A в B будет стоить $50, поездка из B в D обойдется в $90 и т. д. Цена не зависит от того, является ли поездка сегментом более длинного маршрута или нет (в отличие от современных авиаперелетов).

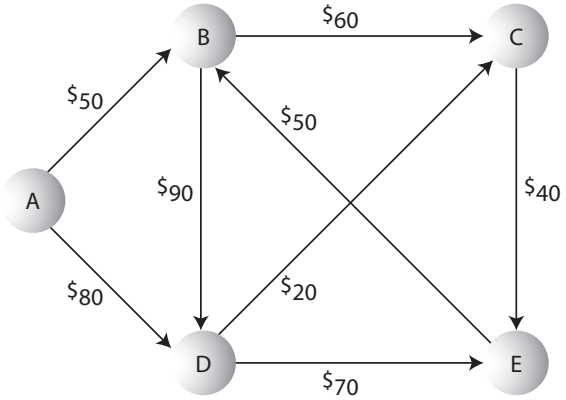

Рис. 14.6. Стоимость поездок

Граф на рис. 14.6 имеет направленные ребра. Они представляют одноколейные железнодорожные линии, по которым (в интересах безопасности) движение осуществляется только в одну сторону. Например, из A в B можно проехать напрямую, а из B в A прямой дороги нет. Хотя в данной ситуации нужно найти самый экономичный маршрут, задача называется задачей выбора кратчайшего пути. Под «кратчайшим» подразумевается не только самый короткий, но и самый экономичный, самый быстрый или оптимальный по какой-то другой характеристике маршрут.

Между любыми двумя городами существует несколько маршрутов. Например, из A в E можно проехать через D, через B и C, через D и C или каким-нибудь другим путем. (Добраться до города F по железной дороге невозможно, поэтому на графе он не представлен. И это хорошо, потому что сокращение количества вершин сокращает размер некоторых списков, которые нам предстоит построить.)

Задача выбора кратчайшего пути в нашем случае формулируется так: какой маршрут будет самым экономичным для заданной начальной и конечной точек? При взгляде на рис. 14.6 нетрудно представить, что самый экономичный маршрут из A в E проходит через D и C и стоит $140.

Направленный взвешенный граф

Как упоминалось ранее, в стране проложена одноколейная железная дорога, поэтому перемещение между любыми двумя городами возможно только в одном направлении. Такая железнодорожная сеть моделируется направленным графом. В более реалистичной ситуации между любыми двумя городами можно было бы перемещаться в обоих направлениях за одинаковую цену; такая модель соответствовала бы ненаправленному графу. Впрочем, задача выбора кратчайшего пути в этих случаях решается сходным образом, поэтому для разнообразия мы покажем, как выглядит решение для направленного графа.

Алгоритм Дейкстры

Следующее решение задачи кратчайшего пути называется *алгоритмом Дейкстры* (по имени Э. Дейкстры, описавшего его в 1959 г.) Алгоритм основан на представлении графа в виде матрицы смежности. Интересно, что алгоритм Дейкстры находит не только кратчайший путь от одной заданной вершины к другой, но и кратчайшие пути от заданной вершины ко всем остальным вершинам.

Агенты и поездки

Чтобы понять, как работает алгоритм Дейкстры, представьте, что вы ищете самый дешевый вариант поездки из A в другие города. Вы (и нанятые вами агенты) будете выполнять функции компьютерной программы, реализующей алгоритм Дейкстры. Конечно, в реальной жизни вы попросту купили бы железнодорожный справочник со всеми ценами. Однако алгоритм должен получать информацию последовательно, поэтому (по аналогии с предыдущим разделом) мы будем считать, что полная информация изначально недоступна.

В каждом городе кассир может сказать, сколько будет стоить прямая поездка в другие города (то есть поездка, при которой не приходится делать пересадку). К сожалению, кассир не знает стоимости билетов в города, находящиеся на расстоянии более одного сегмента. Вы рисуете в блокноте таблицу со столбцом для каждого города. В конце работы алгоритма в каждом столбце должен быть построен самый дешевый маршрут от начальной точки до этого города.

Первый агент: A

В каждом городе должен появиться ваш агент, задача которого — получить информацию о стоимости билетов в другие города. В городе A таким агентом являетесь вы сами. Кассир в A знает лишь то, что поездка в B стоит $50, а поездка в D стоит $80. Вы записываете эту информацию в блокнот (табл. 14.2).

Таблица 14.2. Шаг 1: агент в A

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск

Сокращение «Бск» означает «бесконечность»; из A нельзя попасть в город, указанный в заголовке столбца — или по крайней мере вы еще не знаете, как туда попасть. (Как вы вскоре увидите, в алгоритме бесконечность представляется очень большим числом для упрощения расчетов.) В круглых скобках указывается последний город, посещенный перед прибытием в разные конечные точки. Вскоре вы увидите, для чего нужна эта информация. Что делать теперь? Действуйте по следующему правилу.

ПРАВИЛО

Всегда отправляйте агента в город с минимальной общей стоимостью билета от начальной точки (A).

Города, в которых уже имеется агент, не рассматриваются. Обратите внимание на отличие этого правила от того, которое использовалось при построении минимального остовного дерева (прокладка кабельной сети). Там мы выбирали самый дешевый сегмент (ребро) от подключенных городов к неподключенным, а сейчас выбирается самый экономичный общий маршрут от A к городу без агента. В этой конкретной фазе нашего исследования эти два принципа приводят к одинаковым результатам, потому что все известные маршруты от A состоят лишь из одного ребра, но с рассылкой агентов по другим городам маршруты от A превратятся в набор сегментов.

Второй агент: B

Самый дешевый маршрут из A ведет в B ($50). Вы нанимаете работника и отправляете его в B, где он будет вашим агентом. Он звонит вам и сообщает, что по словам кассира B поездка в C будет стоить $60, а поездка в D обойдется в $90.

Несложные вычисления показывают, что перемещение из A в C через B будет стоить $50 + $60 = $110, поэтому вы изменяете запись для C. Также видно, что перемещение из A в D через B стоит $50 + $90 = $140. Однако — и это очень важный момент — мы уже знаем, что прямая поездка из A в D стоит всего $80. Нас интересует только самый экономичный маршрут из A, поэтому более дорогой маршрут игнорируется, а запись в таблице остается без изменений. В таблице 14.3 приведено текущее состояние таблицы, а на рис. 14.7 изображено его географическое представление.

Таблица 14.3. Шаг 2: агенты в A и B

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск

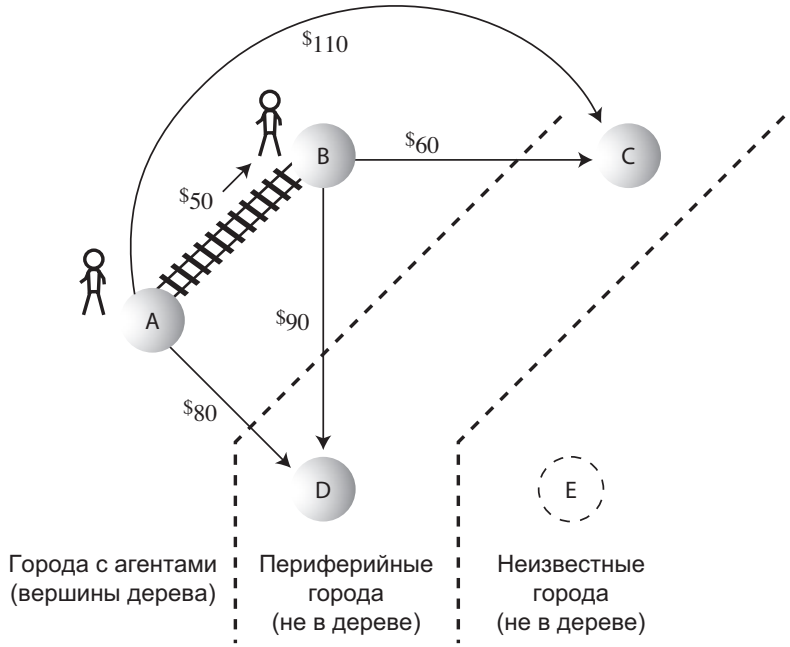

Рис. 14.7. Шаг 2 в алгоритме выбора кратчайшего пути

После того как в городе появится агент, мы точно знаем, что маршрут, выбранный этим агентом для перемещения в город, является самым экономичным. Почему? Возьмем текущий пример. Если бы существовал более дешевый маршрут для перемещения из A в B, то он должен был бы проходить через другой город. Но из A можно перейти только в D, а уже одна эта поездка обойдется дороже прямого перемещения в B. Добавление дополнительных маршрутов для перемещения из D в B только повысит стоимость маршрута в D. Из этого следует, что в дальнейшем самый экономичный маршрут из A в B не должен обновляться. Его стоимость останется неизменной, какую бы информацию мы ни получили о других городах. Маршрут помечается звездочкой (*) — это означает, что в городе имеется агент, а самый дешевый маршрут в него зафиксирован.

Три категории городов

В алгоритме минимального остовного дерева все города делятся на три категории:

1. Города с агентом (входящие в дерево).
2. Города с известной стоимостью перемещения из городов с агентами (периферийные города).
3. Неизвестные города.

В текущем состоянии города A и B относятся к категории 1, потому что там уже присутствуют агенты. Города категории 1 образуют дерево из путей, которые идут от начальной вершины к разным конечным вершинам. (Конечно, это дерево не совпадает с минимальным остовным деревом.)

В некоторых других городах агентов нет, но стоимость перемещения в них известна, потому что агенты находятся в смежных городах категории 1. Мы знаем, что поездка из A в D стоит $80, а поездка из B в C обойдется в $60. Так как стоимость

поездки в D и C известна, эти города относятся к категории 2 (периферийные города).

Пока мы не располагаем информацией о городе E, поэтому этот город относится к категории «неизвестных». На рис. 14.7 показано распределение городов по категориям для текущего состояния алгоритма.

Как и при построении минимального остовного дерева, этот алгоритм в процессе своей работы постепенно перемещает города из категории неизвестных в категорию периферийных, а из категории периферийных — в дерево.

Третий агент: D

На данный момент самым экономичным из известных маршрутов из A в любой город без агента является прямой маршрут A-D за $80. Маршруты A-B-C ($110) и A-B-D ($140) стоят дороже.

Вы нанимаете очередного агента и отправляете его в D за $80. Агент сообщает, что из D можно добраться в C ($20) или в E ($80). Теперь можно изменить данные для C. Ранее добраться до этого города из A можно было через B за $110. Теперь мы также видим, что до C можно добраться всего за $100 через D. Также определилась стоимость поездки из A в ранее неизвестный город E — $150 через D. Вы отмечаете внесенные изменения (табл. 14.4 и рис. 14.8).

Таблица 14.4. Шаг 3: агенты в A, B и D

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск
Шаг 3	50 (через A)*	100 (через D)	80 (через A)*	150 (через D)

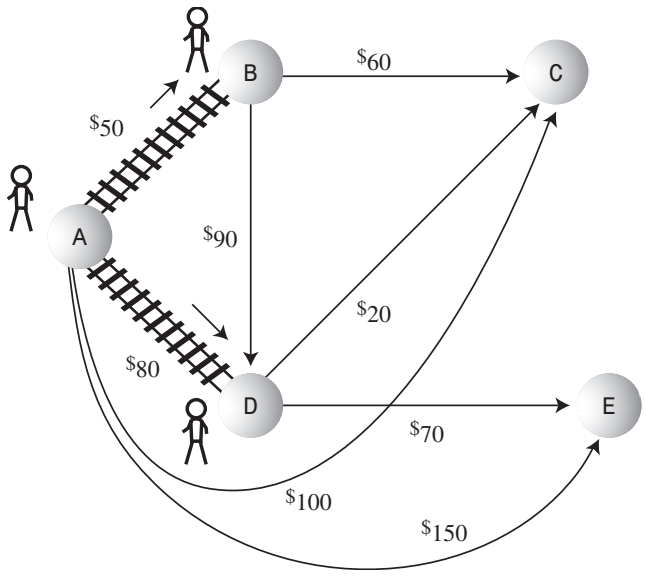

Рис. 14.8. Шаг 3 в алгоритме выбора кратчайшего пути

Четвертый агент: C

Теперь самым экономичным путем в любой город без агента становится путь из A в C через D за $100. Соответственно вы отправляете агента по этому маршруту в C. Агент сообщает, что из C в E можно добраться за $40. А раз C находится на расстоянии $100 из A (через D), а E находится на расстоянии $40 от C, минимальная стоимость поездки из A в E сокращается с $150 (маршрут A-D-E) до $140 (маршрут A-D-C-E). Обновленная информация представлена в табл. 14.5 и на рис. 14.9.

Таблица 14.5. Шаг 4: агенты в A, B, D и C

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск
Шаг 3	50 (через A)*	100 (через D)	80 (через A)*	150 (через D)
Шаг 4	50 (через A)*	100 (через D)*	80 (через A)*	140 (через C)

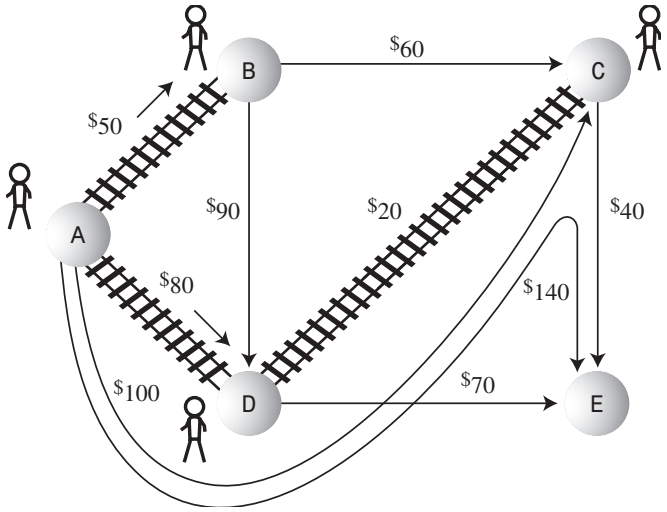

Рис. 14.9. Шаг 4 в алгоритме выбора кратчайшего пути

Последний агент: E

Самым экономичным путем из A в любой другой город, в котором пока нет своего агента, становится путь в E через D и C ($140). Вы отправляете агента в E, но агент сообщает, что из E нет ни одного маршрута в город без агентов (есть маршрут в B, но в B уже есть агент). Таблица 14.6 дополняется последней строкой; мы всего лишь добавляем звездочку в запись E, чтобы обозначить присутствие агента в этом городе.

Таблица 14.6. Шаг 4: агенты в A, B, D, C и E

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск

продолжение ⇩

Таблица 14.6 (*продолжение*)

Из города A	B	C	D	E
Шаг 3	50 (через A)*	100 (через D)	80 (через A)*	150 (через D)
Шаг 4	50 (через A)*	100 (через D)*	80 (через A)*	140 (через C)
Шаг 5	50 (через A)*	100 (через D)*	80 (через A)*	140 (через C)*

Когда в каждом городе появится свой агент, мы узнаем стоимость проезда из A в каждый из остальных городов. Работа алгоритма закончена. Без каких-либо дополнительных вычислений мы получили стоимости самых экономичных маршрутов из A во все остальные города.

Это описание демонстрирует основные принципы алгоритма Дейкстры. Ключевые моменты:

- Каждый раз при отправке агента в новый город вы используете информацию, полученную от него, для обновления списка стоимости проезда. В списке сохраняется только самый экономичный маршрут (из известных вам) от начальной точки до заданного города.
- Новый агент всегда отправляется в город с самым экономичным маршрутом от начальной точки (а не с самым дешевым сегментом от любого города, в котором уже есть агент, как при построении минимального остовного дерева).

Приложение GraphDW Workshop

За тем, как работает алгоритм Дейкстры, можно понаблюдать в приложении GraphDW Workshop. Создайте в этом приложении граф, изображенный на рис. 14.6. Результат должен выглядеть так, как показано на рис. 14.10. (О том, как вывести под графом таблицу, будет рассказано ниже.) Граф является взвешенным и направленным, поэтому при создании ребра необходимо ввести вес, а указатель мыши необходимо перетаскивать в правильном направлении, от начала к концу.

Когда построение графа будет завершено, щелкните на кнопке `Path`, а затем по запросу щелкните на вершине A. Еще несколько щелчков на кнопке `Path` включают A в дерево (вершина выделяется красным контуром).

Массив кратчайшего пути

При следующем нажатии кнопки `Path` под графом появляется таблица (рис. 14.10). Алгоритм Дейкстры начинает работу с копирования соответствующей строки матрицы смежности (то есть строки начальной вершины) в массив. (Не забудьте, что матрицу смежности можно просмотреть в любой момент при помощи кнопки `View`.) Этот массив называется «массивом кратчайших путей». Он соответствует последней строке таблицы в блокноте в нашем примере с вычислением самого экономичного маршрута. В массиве хранятся текущие версии кратчайших путей к другим вершинам, которые мы будем называть *конечными* вершинами. В таблице 14.7 конечные вершины представлены заголовками столбцов.

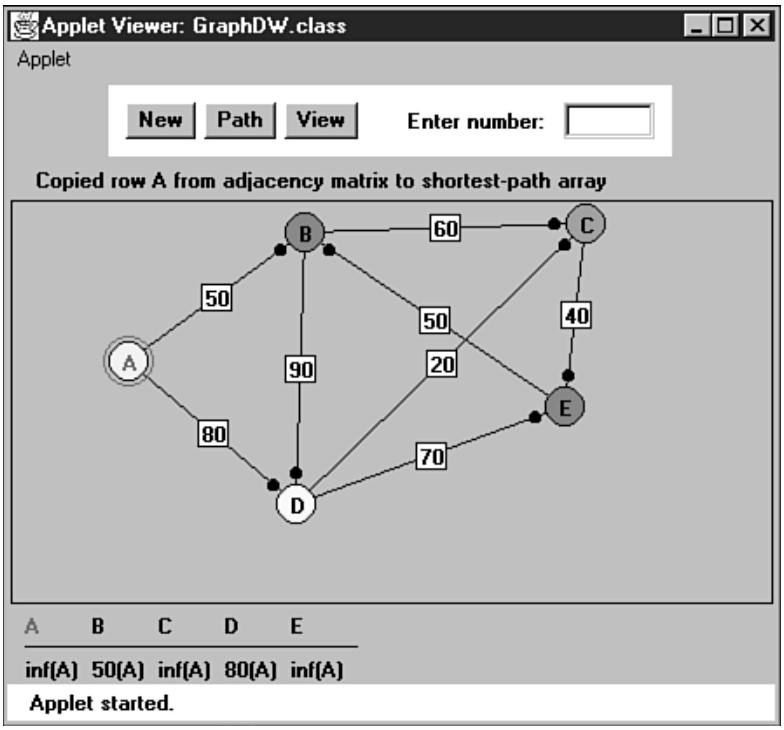

Рис. 14.10. Железнодорожный граф в приложении GraphDW

Таблица 14.7. Шаг 1: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	Бск (A)	80 (A)	Бск (A)

В приложении за длинами кратчайших путей в массиве указываются *родительские* вершины в круглых скобках. Родительской называется вершина, достигнутая непосредственно перед достижением конечной вершины. В данном примере в качестве родителя везде указывается вершина A, потому что алгоритм переместился от A всего на одно ребро.

Если стоимость проезда неизвестна (или не имеет смысла, как для маршрута A-A), она представляется в виде бесконечной величины («бск» в таблице маршрутов). Обратите внимание: заголовки столбцов тех вершин, которые уже были добавлены в дерево, выделены красным цветом. Данные этих столбцов остаются неизменными.

Минимальное расстояние

Изначально алгоритм знает расстояния от A до других вершин, находящихся на расстоянии ровно одного ребра от A. Смежными с A являются только B и D, поэтому расстояния приводятся только для них. Алгоритм выбирает минимальное расстояние. При следующем нажатии кнопки `Path` появляется сообщение о том, что на минимальном расстоянии от A (50) находится вершина B. Алгоритм добавляет вершину в дерево, поэтому при следующем щелчке выводится сообщение о включении вершины B в дерево.

На графе вершина B выделяется контуром, а заголовок столбца B окрашивается в красный цвет. Ребро от A к B темнеет, что указывает на его принадлежность к дереву.

Перебор столбцов в массиве кратчайших путей

Теперь алгоритму известны не только все ребра, ведущие из A, но и все ребра из B. Он перебирает массив кратчайших путей столбец за столбцом, проверяя, нельзя ли вычислить более короткий путь на основании новой информации. Вершины, уже находящиеся в дереве (A и B), пропускаются. Сначала проверяется столбец C. Мы видим сообщение о том, что суммарный вес ребер A-B (50) и B-C (60) меньше веса ребра A-C (бесконечность).

Алгоритм обнаружил путь к C, который короче пути в массиве (бесконечность в столбце C). Вес ребра от A к B равен 50 (алгоритм находит это значение в столбце B массива кратчайших путей), а вес ребра от B к C равен 60 (находится на пересечении строки B и столбца C матрицы смежности). Сумма равна 110. Расстояние 110 меньше бесконечности, поэтому алгоритм обновляет массив кратчайших путей для столбца C и вставляет в него значение 110. За ним указывается B в круглых скобках — последняя вершина перед C; B является родителем C.

Затем анализируется столбец D. В сообщении говорится, что сумма A-B (50) и B-D (90) больше либо равна весу A-D (80). Алгоритм сравнивает ранее приводившееся расстояние от A до D, равное 80 (прямой маршрут) с потенциальным маршрутом через B (то есть A-B-D). Но суммарный вес последнего маршрута равен 140 (50 + 90). Результат больше 80, поэтому значение 80 не изменяется.

Для столбца E выводится сообщение о том, что суммарный вес сегментов A-B (50) и B-E (бесконечность) больше либо равен весу A-E (бесконечность). Соответственно столбец E не изменяется, а массив кратчайших путей теперь выглядит так, как показано в табл. 14.8.

Таблица 14.8. Шаг 2: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	110 (B)	80 (A)	Бск (A)

Теперь проясняется роль родительской вершины, указываемой в круглых скобках после каждого расстояния. В каждом столбце содержится расстояние от A до конечной вершины. Родителем является непосредственный предшественник конечной вершины на пути от A. В столбце C родительской вершиной является B; это означает, что кратчайший путь от A к C проходит через вершину B непосредственно перед попаданием в C. Эта информация используется алгоритмом для включения в дерево соответствующей вершины. (При бесконечном расстоянии родительская вершина не имеет смысла и вместо нее выводится A.)

Новое минимальное расстояние

После обновления массива кратчайших путей алгоритм ищет кратчайшее расстояние в массиве (еще раз нажмите кнопку `Path`). Сообщение гласит, что минимальное расстояние от A равно 80 — для вершины D. Соответственно в дерево включается новая вершина D и ребро.

Снова и снова

Алгоритм снова перебирает массив кратчайших путей, проверяя и обновляя расстояния для конечных вершин, не входящих в дерево; в этой категории остаются только вершины C и E. Результат обновления столбцов C и E приведен в табл. 14.9.

Таблица 14.9. Шаг 3: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	100 (D)	80 (A)	150 (D)

Кратчайший путь от A к вершине, не принадлежащей дереву, имеет вес 100 для вершины C. Соответственно вершина C включается в дерево.

При следующем просмотре массива кратчайших путей рассматривается только расстояние до E. Оно может быть сокращено прохождением через C; обновленные данные приведены в табл. 14.10.

Таблица 14.10. Шаг 3: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	100 (D)	80 (A)	140 (C)

Последняя вершина E включена в дерево, а работа алгоритма закончена. В массиве кратчайших путей содержатся наименьшие расстояния от A до всех остальных вершин. Дерево состоит из вершин и ребер AB, AD, DC и CE, выделенных толстыми линиями.

Последовательность вершин по кратчайшему пути к любой вершине. Например, в кратчайшем пути к E родителем вершины E является вершина C, приведенная в массиве в круглых скобках. Родителем C (также согласно данным из массива) является вершина D, а вершине D предшествует A. Таким образом, кратчайший путь от A к E проходит по маршруту A-D-C-E.

Поэкспериментируйте с приложением GraphDW; начните с небольших графов. Через некоторое время вы поймете суть алгоритма Дейкстры и сможете предсказать, что он сделает на следующем шаге.

Реализация на языке Java

Алгоритм выбора кратчайшего пути — один из самых сложных в этой книге, но он вполне по силам простым смертным. Сначала мы рассмотрим вспомогательный класс, затем основной метод выполнения алгоритма `path()` и в завершение два метода, вызываемых `path()` для выполнения специализированных задач.

Массив sPath и класс DistPar

Как мы уже видели, основной структурой данных в алгоритме кратчайшего пути является массив с минимальными расстояниями от начальной вершины до других (конечных) вершин. В ходе выполнения алгоритма эти расстояния изменяются, а на последнем шаге элементы массива содержат фактические кратчайшие расстояния от начала. В приведенном коде этот массив называется `sPath[]`.

Как мы уже видели, очень важно знать не только минимальное расстояние от начальной вершины до конечной, но и путь, на котором это расстояние достигается. К счастью, хранить весь путь в явном виде не обязательно. Достаточно хранить в массиве родителя конечную вершину (то есть вершину, предшествующую конечной). Если в столбце C хранится запись 100(D), это означает, что длина кратчайшего пути от A к C равна 100, и вершина D непосредственно предшествует C на этом пути.

Существует много разных способов хранения информации о родительской вершине. Мы объединили родительскую вершину с расстоянием в классе `DistPar`; объекты этого класса хранятся в элементах массива `sPath[]`.

```
class DistPar             // Расстояние и родительская вершина
   {                      // Объекты сохраняются в массиве sPath
   public int distance;   // Расстояние от начальной вершины до текущей
   public int parentVert; // Текущий родитель вершины
   public DistPar(int pv, int d)  // Конструктор
      {
      distance = d;
      parentVert = pv;
      }
   }
```

Метод path()

Метод `path()` непосредственно реализует алгоритм выбора кратчайшего пути. В нем используется класс `DistPar` и класс `Vertex`, уже знакомый по программе `mstw.java` (см. листинг 14.1). Метод `path()` является методом класса `Graph`, который тоже встречался нам в несколько измененном виде в программе `mstw.java`.

```
public void path()               // Выбор кратчайших путей
   {
   int startTree = 0;            // Начиная с вершины 0
   vertexList[startTree].isInTree = true;
```

```
nTree = 1;                      // Включение в дерево

// Перемещение строки расстояний из adjMat в sPath
for(int j=0; j<nVerts; j++)
   {
   int tempDist = adjMat[startTree][j];
   sPath[j] = new DistPar(startTree, tempDist);
   }

// Пока все вершины не окажутся в дереве
while(nTree < nVerts)
   {
   int indexMin = getMin();    // Получение минимума из sPath
   int minDist = sPath[indexMin].distance;

   if(minDist == INFINITY)     // Если все расстояния бесконечны
      {                        // или уже находятся в дереве,
      System.out.println("There are unreachable vertices");
      break;                   // построение sPath завершено
      }
   else
      {                        // Возврат currentVert
      currentVert = indexMin;  // к ближайшей вершине
      startToCurrent = sPath[indexMin].distance;
      // Минимальное расстояние от startTree
      // до currentVert равно startToCurrent
      }
   // Включение текущей вершины в дерево
   vertexList[currentVert].isInTree = true;
   nTree++;
   adjust_sPath();             // Обновление массива sPath[]
   }
displayPaths();                // Вывод содержимого sPath[]

nTree = 0;                     // Очистка дерева
for(int j=0; j<nVerts; j++)
   vertexList[j].isInTree = false;
}
```

Начальная вершина всегда хранится в ячейке 0 массива `vertexList[]`. Выполнение метода `path()` начинается с включения этой вершины в дерево. По ходу работы алгоритма другие вершины также будут включаться в дерево. Класс `Vertex` содержит флаг, обозначающий принадлежность объекта вершины к дереву. Чтобы включить вершину в дерево, необходимо установить флаг и увеличить `nTree` — счетчик вершин в дереве. Затем `path()` копирует расстояния из соответствующей строки матрицы смежности в `sPath[]`. Ей всегда является строка 0, потому что для простоты мы считаем, что индекс начальной вершины всегда равен 0. Изначально в поле родительской вершины всех элементов `sPath[]` хранится начальная вершина A.

Далее метод входит в основной цикл while алгоритма. Цикл завершается после того, как все вершины будут включены в дерево. Фактически в цикле выполняются три действия:

1. Выбор элемента sPath[] с наименьшим расстоянием.
2. Включение соответствующей вершины (заголовка столбца для этого элемента) в дерево. Вершина становится «текущей», то есть currentVert.
3. Обновление всех элементов sPath[] в соответствии с расстояниями от currentVert.

Если метод path() обнаруживает, что минимальное расстояние равно бесконечности, то он знает, что некоторые вершины недостижимы от начальной точки. Почему? Потому что не все вершины были включены в дерево (цикл while еще не завершился), но путь перехода к этим вершинам не был найден; в противном случае расстояние не было бы бесконечным.

Прежде чем возвращать управление, path() выводит окончательное содержимое sPath[] вызовом метода displayPaths(). Никаких других данных программа не выводит. Кроме того, path() обнуляет nTree и удаляет флаги isInTree из всех вершин на случай, если они будут использоваться другим алгоритмом (хотя к нашей программе это не относится).

Определение минимального расстояния методом getMin()

Чтобы найти элемент sPath[] с минимальным расстоянием, path() вызывает метод getMin(). Метод вполне тривиален — он перебирает элементы sPath[] и возвращает номер столбца (индекс массива) элемента с наименьшим расстоянием.

```
public int getMin()              // Поиск в sPath элемента
   {                             // с наименьшим расстоянием
   int minDist = INFINITY;       // Исходный высокий "минимум"
   int indexMin = 0;
   for(int j=1; j<nVerts; j++)   // Для каждой вершины
      {                          // Если она не включена в дерево
      if( !vertexList[j].isInTree &&  // и ее расстояние меньше
                  sPath[j].distance < minDist )  // старого минимума
         {
            minDist = sPath[j].distance;
            indexMin = j;            // Обновление минимума
            }
      }
   return indexMin;              // Метод возвращает индекс
   }                             // элемента с наименьшим расстоянием
```

В основу алгоритма выбора кратчайшего пути можно было бы заложить приоритетную очередь, как было сделано в предыдущем разделе при поиске минимального остовного дерева. В этом случае метод getMin() был бы лишним; ребро с минимальным весом автоматически помещалось бы в начало массива. Тем не менее при использовании массива проще понять, что происходит в алгоритме.

Обновление sPath[] методом adjust_sPath()

Метод adjust_sPath() используется для обновления sPath[] новой информацией о вершинах, только что вставленных в дерево. При вызове этого метода вершина currentVert только что была вставлена в дерево, а startToCurrent содержит текущие данные из sPath[] для этой вершины. Метод adjust_sPath() проверяет каждую вершину в sPath[], последовательно перебирая вершины в переменной цикла column. Для каждого элемента sPath[] при условии, что вершина не включена в дерево, метод выполняет три операции:

1. Расстояние до текущей вершины (уже вычисленное и хранящееся в startToCurrent) суммируется с весом ребра от currentVert до вершины column. Результат сохраняется в переменной startToFringe.
2. Значение startToFringe сравнивается с текущими данными в sPath[].
3. Если значение startToFringe меньше, оно заменяет элемент в sPath[].

В этих операциях заключена суть алгоритма Дейкстры. Массив sPath[] обновляется кратчайшими расстояниями для всех вершин, известных в настоящее время. Код метода adjust_sPath():

```
public void adjust_sPath()
   {
   // Обновление данных в массиве кратчайших путей sPath
   int column = 1;                 // Начальная вершина пропускается
   while(column < nVerts)          // Перебор столбцов
      {
      // Если вершина column уже включена в дерево, она пропускается
      if( vertexList[column].isInTree )
         {
         column++;
         continue;
         }
      // Вычисление расстояния для одного элемента sPath
                     // Получение ребра от currentVert к column
      int currentToFringe = adjMat[currentVert][column];
                     // Суммирование расстояний
      int startToFringe = startToCurrent + currentToFringe;
                     // Определение расстояния текущего элемента sPath
      int sPathDist = sPath[column].distance;

      // Сравнение расстояния от начальной вершины с элементом sPath
      if(startToFringe < sPathDist)   // Если меньше,
         {                            // данные sPath обновляются
         sPath[column].parentVert = currentVert;
         sPath[column].distance = startToFringe;
         }
      column++;
      }
}
```

Метод `main()` в программе `path.java` строит дерево, изображенное на рис. 14.6, и выводит для него массив кратчайших путей. Код метода выглядит так:

```
public static void main(String[] args)
   {
   Graph theGraph = new Graph();
   theGraph.addVertex('A');     // 0  (исходная вершина)
   theGraph.addVertex('B');     // 1
   theGraph.addVertex('C');     // 2
   theGraph.addVertex('D');     // 3
   theGraph.addVertex('E');     // 4

   theGraph.addEdge(0, 1, 50);  // AB 50
   theGraph.addEdge(0, 3, 80);  // AD 80
   theGraph.addEdge(1, 2, 60);  // BC 60
   theGraph.addEdge(1, 3, 90);  // BD 90
   theGraph.addEdge(2, 4, 40);  // CE 40
   theGraph.addEdge(3, 2, 20);  // DC 20
   theGraph.addEdge(3, 4, 70);  // DE 70
   theGraph.addEdge(4, 1, 50);  // EB 50

   System.out.println("Shortest paths");
   theGraph.path();             // Кратчайшие пути
   System.out.println();
   }
```

Программа выводит следующий результат:

```
A=inf(A) B=50(A) C=100(D) D=80(A) E=140(C)
```

Программа path.java

В листинге 14.2 приведен полный код программы `path.java`. Все ее компоненты были рассмотрены выше.

Листинг 14.2. Программа path.java

```
// path.java
// Выбор кратчайшего пути в направленном взвешенном графе
// Запуск программы: C>java PathApp
////////////////////////////////////////////////////////////
class DistPar             // Расстояние и родительская вершина
   {                      // Объекты сохраняются в массиве sPath
   public int distance;   // Расстояние от начальной вершины до текущей
   public int parentVert; // Текущий родитель вершины
// --------------------------------------------------------------
   public DistPar(int pv, int d)  // Конструктор
      {
      distance = d;
      parentVert = pv;
      }
```

```
   }  // Конец класса DistPar
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean isInTree;
// ----------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      isInTree = false;
      }
// ----------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private final int INFINITY = 1000000;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private int nTree;           // Количество вершин в дереве
   private DistPar sPath[];     // Массив данных кратчайших путей
   private int currentVert;     // Текущая вершина
   private int startToCurrent;  // Расстояние до currentVert
// ----------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      nTree = 0;
      for(int j=0; j<MAX_VERTS; j++)     // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)  // заполняется
            adjMat[j][k] = INFINITY;     // бесконечными расстояниями
      sPath = new DistPar[MAX_VERTS];    // shortest paths
   }
// ----------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// ----------------------------------------------------------
   public void addEdge(int start, int end, int weight)
      {
      adjMat[start][end] = weight;  // (направленный граф)
      }
```

продолжение ⇩

Листинг 14.2 (*продолжение*)

```
// -------------------------------------------------------------
   public void path()               // Выбор кратчайших путей
      {
      int startTree = 0;            // Начиная с вершины 0
      vertexList[startTree].isInTree = true;
      nTree = 1;                    // Включение в дерево

      // Перемещение строки расстояний из adjMat в sPath
      for(int j=0; j<nVerts; j++)
         {
         int tempDist = adjMat[startTree][j];
         sPath[j] = new DistPar(startTree, tempDist);
         }

      // Пока все вершины не окажутся в дереве
      while(nTree < nVerts)
         {
         int indexMin = getMin();    // Получение минимума из sPath
         int minDist = sPath[indexMin].distance;

         if(minDist == INFINITY)     // Если все расстояния бесконечны
            {                        // или уже находятся в дереве,
            System.out.println("There are unreachable vertices");
            break;                   // построение sPath завершено
            }
         else
            {                        // Возврат currentVert
            currentVert = indexMin;  // к ближайшей вершине
            startToCurrent = sPath[indexMin].distance;
            // Минимальное расстояние от startTree
            // до currentVert равно startToCurrent
            }
         // Включение текущей вершины в дерево
         vertexList[currentVert].isInTree = true;
         nTree++;
         adjust_sPath();             // Обновление массива sPath[]
         }

      displayPaths();                // Вывод содержимого sPath[]

      nTree = 0;                     // Очистка дерева
      for(int j=0; j<nVerts; j++)
         vertexList[j].isInTree = false;
      }
// -------------------------------------------------------------
   public int getMin()              // Поиск в sPath элемента
      {                             //  с наименьшим расстоянием
      int minDist = INFINITY;       // Исходный высокий "минимум"
      int indexMin = 0;
```

```
      for(int j=1; j<nVerts; j++)    // Для каждой вершины
         {                           // Если она не включена в дерево
         if( !vertexList[j].isInTree &&  // и ее расстояние меньше
                   sPath[j].distance < minDist )  // старого минимума
            {
            minDist = sPath[j].distance;
            indexMin = j;            // Обновление минимума
            }
         }
      return indexMin;               // Метод возвращает индекс
      }                              // элемента с наименьшим расстоянием
// -------------------------------------------------------------
   public void adjust_sPath()
      {
      // Обновление данных в массиве кратчайших путей sPath
      int column = 1;                // Начальная вершина пропускается
      while(column < nVerts)         // Перебор столбцов
         {
         // Если вершина column уже включена в дерево, она пропускается
         if( vertexList[column].isInTree )
            {
            column++;
            continue;
            }
         // Вычисление расстояния для одного элемента sPath
                        // Получение ребра от currentVert к column
         int currentToFringe = adjMat[currentVert][column];
                        // Суммирование расстояний
         int startToFringe = startToCurrent + currentToFringe;
                        // Определение расстояния текущего элемента sPath
         int sPathDist = sPath[column].distance;

         // Сравнение расстояния от начальной вершины с элементом sPath
         if(startToFringe < sPathDist)   // Если меньше,
            {                            // данные sPath обновляются
            sPath[column].parentVert = currentVert;
            sPath[column].distance = startToFringe;
            }
         column++;
         }
   }
// -------------------------------------------------------------
   public void displayPaths()
      {
      for(int j=0; j<nVerts; j++) // display contents of sPath[]
         {
         System.out.print(vertexList[j].label + "="); // B=
         if(sPath[j].distance == INFINITY)
            System.out.print("inf");                 // inf
```

продолжение ⇩

Листинг 14.2 (*продолжение*)

```
         else
            System.out.print(sPath[j].distance);      // 50
         char parent = vertexList[ sPath[j].parentVert ].label;
         System.out.print("(" + parent + ") ");       // (A)
         }
      System.out.println("");
      }
// --------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class PathApp
{
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');     // 0  (исходная вершина)
      theGraph.addVertex('B');     // 1
      theGraph.addVertex('C');     // 2
      theGraph.addVertex('D');     // 3
      theGraph.addVertex('E');     // 4

      theGraph.addEdge(0, 1, 50);  // AB 50
      theGraph.addEdge(0, 3, 80);  // AD 80
      theGraph.addEdge(1, 2, 60);  // BC 60
      theGraph.addEdge(1, 3, 90);  // BD 90
      theGraph.addEdge(2, 4, 40);  // CE 40
      theGraph.addEdge(3, 2, 20);  // DC 20
      theGraph.addEdge(3, 4, 70);  // DE 70
      theGraph.addEdge(4, 1, 50);  // EB 50

      System.out.println("Shortest paths");
      theGraph.path();             // Кратчайшие пути
      System.out.println();
      }
   }  // Конец класса PathApp
////////////////////////////////////////////////////////////////
```

Поиск кратчайших путей между всеми парами вершин

Когда в главе 13 речь шла о связности графов, мы хотели узнать, возможен ли перелет от Афин до Мурманска при любом количестве пересадок. Когда вы имеете дело со взвешенным графом, появляется второй разумный вопрос: сколько будет стоить такое путешествие?

Чтобы узнать, возможно ли перемещение между вершинами, мы строили таблицу связности. Для взвешенного графа аналогичная таблица должна содержать

минимальные стоимости перемещения между всеми парами вершин по разным ребрам. Задача построения такой таблицы называется задачей *поиска кратчайших путей между всеми парами вершин*.

Чтобы построить такую таблицу, можно выполнить программу `path.java` для всех вершин попеременно. Примерный результат показан на рис. 14.11.

Таблица 14.11. Таблица кратчайших путей между всеми парами вершин

	A	B	C	D	E
A	–	50	100	80	140
B	–	–	60	90	100
C	–	90	–	180	40
D	–	110	20	–	60
E	–	50	110	140	–

В предыдущей главе было показано, что для создания таблицы с информацией о достижимости вершин от конкретной вершины (за один или несколько шагов) существует более быстрый способ — алгоритм Уоршелла. Аналогичное решение для взвешенных графов использует алгоритм Флойда, предложенный Робертом Флойдом в 1962 году. Это другой способ построения данных из табл. 14.11.

Рассмотрим работу алгоритма Флойда на примере упрощенного графа. На рис. 14.11 изображен взвешенный направленный граф с матрицей смежности.

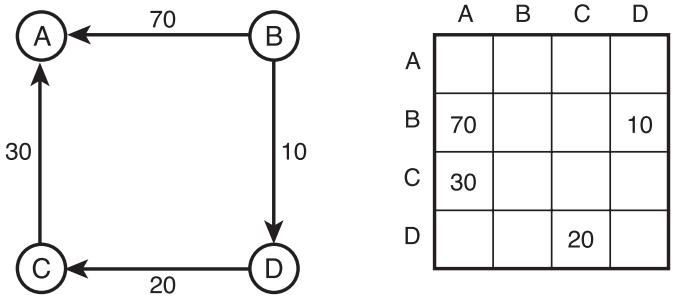

Рис. 14.11. Взвешенный граф и его матрица смежности

В матрице смежности указана стоимость всех путей, состоящих из одного ребра. Мы хотим расширить матрицу таким образом, чтобы в ней указывалась стоимость всех путей независимо от их длины. Например, из рис. 14.11 видно, что стоимость перехода от B к C равна 30 (10 от B к D плюс 20 от D к C).

Как и в алгоритме Уоршелла, мы будем систематически изменять матрицу смежности. Алгоритм последовательно проверяет каждую ячейку каждой строки. Если вес отличен от нуля (допустим, 30 для строки C и столбца A), мы просматриваем содержимое столбца C (потому что значение 30 находится в строке C). При обнаружении данных в столбце C (40 на пересечении со строкой D) мы знаем, что от C к A ведет путь с весом 30, а от D к C — путь с весом 40. Из этого можно сделать вывод, что от D к A существует двухшаговый путь с суммарным весом 70. На рис. 14.12 показан результат применения алгоритма Флойда к графу на рис. 14.11.

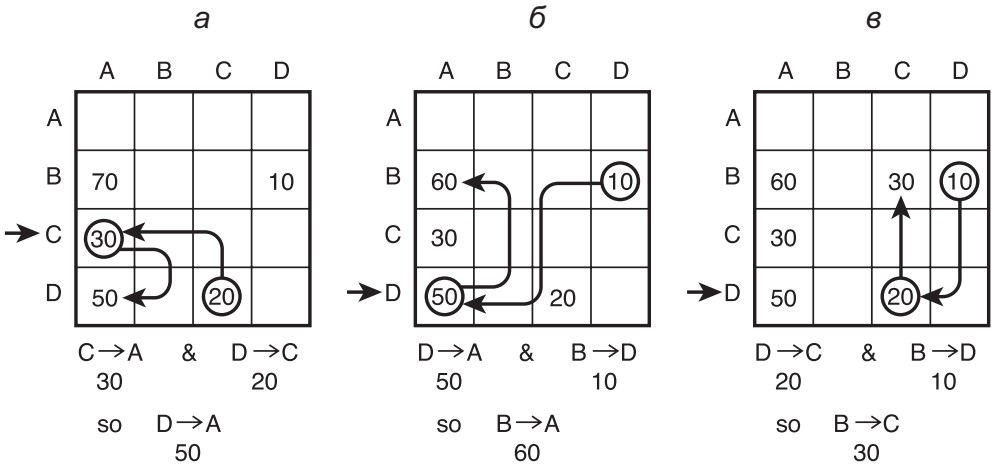

Рис. 14.12. Алгоритм Флойда: *а*) $y = 2$, $x = 0$, $z = 3$; *б*) $y = 3$, $x = 0$, $z = 1$; *в*) $y = 3$, $x = 2$, $z = 1$

Строка A пуста, делать здесь нечего. В строке B столбец A содержит значение 70, а столбец D — значение 10, но в столбце B данных нет, поэтому элементы строки B не удастся объединить с ребрами, заканчивающимися в B.

Однако в строке C мы находим значение 30 на пересечении со столбцом A. При обращении к столбцу C в строке D обнаруживается значение 20. Теперь нам известны пути от C к A (вес 30) и от D к C с весом 20; следовательно, от D к A существует путь с весом 50.

В строке D возникает интересная ситуация со снижением текущей стоимости. Столбец A содержит значение 50. Кроме того, на пересечении строки B и столбца D находится значение 10; следовательно, от B к A ведет путь со стоимостью 60. Однако в этой ячейке уже хранится значение 70. Так как 60 меньше 70, мы заменяем 70 новым значением 60. При наличии нескольких путей между вершинами в таблице должна храниться стоимость пути с наименьшим суммарным весом.

Реализация алгоритма Флойда сходна с реализацией алгоритма Уоршелла, но вместо того, чтобы просто вставлять единицу в таблицу при обнаружении пути из двух ребер, мы суммируем веса двух одношаговых путей и вставляем сумму. Подробности реализации предоставляются читателю для самостоятельной работы.

Эффективность

До настоящего момента мы не рассматривали эффективность различных алгоритмов, работающих с графами. Ситуация усложняется наличием двух способов представления графов: матрицы смежности и списков смежности.

При использовании матрицы смежности описанные алгоритмы в основном выполняются со сложностью $O(V^2)$, где V — количество вершин. Почему? Анализ алгоритмов показывает, что они основаны на однократной проверке каждой вершины, после чего для каждой вершины просматривается соответствующая строка матрицы смежности. Иначе говоря, просматривается каждая ячейка матрицы смежности, состоящей из V^2 ячеек.

Для больших матриц сложность $O(V^2)$ оставляет желать лучшего. Для насыщенных графов возможности улучшения эффективности ограничены. (*Насыщенным* называется граф с большим количеством ребер, для которого заполнено большинство ячеек матрицы смежности.)

Однако большинство графов относится к категории разреженных. Не существует четкого определения того, при каком количестве ребер граф может считаться насыщенным или разреженным, но если каждая вершина большого графа соединяется с соседними вершинами всего несколькими ребрами, такой граф обычно относится к разреженным.

В разреженном графе время работы алгоритма обычно улучшается при переходе от представления в виде матрицы смежности к спискам смежности. Причины понятны: алгоритм не тратит время на проверку ячеек матрицы смежности, не содержащих ребер.

Для невзвешенных графов алгоритм обхода в глубину со списками смежности выполняется за время $O(V + E)$, где V — количество вершин, а E — количество ребер. Для взвешенных графов алгоритм построения минимального остовного дерева и алгоритм кратчайшего пути выполняются за время $O((E + V) \log V)$. В больших разреженных графах эти показатели могут стать значительным улучшением по сравнению с матрицей смежности. С другой стороны, реализация алгоритмов несколько усложняется, поэтому в этой главе использовалось исключительно представление с матрицей смежности. За примерами реализации алгоритмов графов на основе списков смежности обращайтесь к Седжвику (см. приложение Б) и другим источникам.

Алгоритмы Уоршелла и Флойда работают медленнее других алгоритмов, представленных в книге. Оба алгоритма требуют времени $O(V^3)$, так как в их реализации используется тройной вложенный цикл.

Неразрешимые задачи

В книге нам встречались алгоритмы с разной сложностью: от $O(1)$ к $O(N)$, $O(N*\log N)$, $O(N^2)$ и (для алгоритмов Уоршелла и Флойда) $O(N^3)$. Даже при сложности $O(N^3)$ при тысячных значениях N вычисления могут быть выполнены за приемлемое время. Алгоритмы с такой сложностью могут использоваться для поиска решений большинства реальных задач.

С другой стороны, у некоторых алгоритмов сложность оказывается настолько большой, что они могут применяться только для относительно небольших значений N. Многие реальные задачи, требующие применения таких алгоритмов, просто не могут быть решены за сколько-нибудь разумный промежуток времени. Такие задачи называются неразрешимыми. Также встречается термин «НП-полные задачи», где сокращение «НП» означает «недетерминированный полиномиальный» (к сожалению, подробные объяснения выходят за рамки книги).

Обход доски ходом шахматного коня

Задача обхода доски ходом шахматного коня (программный проект 13.5 в главе 13) относится к категории неразрешимых из-за слишком большого количества возможных ходов. Общее количество возможных последовательностей ходов плохо поддается точным вычислениям, но приблизительная оценка возможна. Максимальное количество полей, на которые может быть сделан ход с каждого поля, равно 8. Это число сокращается за счет ходов, ведущих за край доски, а также ходов на поля, которые уже были посещены ранее. На ранней стадии обхода количество возможных ходов будет оставаться близким к 8, но будет постепенно сокращаться по мере заполнения доски. Предположим (более чем консервативно), что из каждой позиции возможно в среднем только два хода. После первого хода конь может посетить еще 63 оставшихся поля. Таким образом, общее количество возможных ходов равно 2^{63}, или около 10^{19}. Компьютер обрабатывает до миллиона ходов в секунду, или 10^6. Год состоит примерно из 10^7 секунд, поэтому за год компьютер обработает 10^{13} ходов. Таким образом, решение задачи методом «грубой силы» займет 10^6 (около миллиона) лет.

Разрешимость этой конкретной задачи можно несколько улучшить за счет применения стратегий «усечения» игрового дерева. Одна из них — эвристическое правило Уорнсдорфа (Г. К. фон Уорнсдорф, 1823) — указывает, что перемещение всегда должно вести к полю с минимальным количеством возможных выходов.

Задача коммивояжера

Другая известная неразрешимая задача: представьте, что вы — коммивояжер и вам нужно объехать все города, в которых у вас имеются клиенты. Вам хотелось бы свести к минимуму пройденное расстояние. Известно расстояние между любыми двумя городами. Вы хотите начать поездку со своего города, посетить каждый город с клиентом ровно один раз и вернуться в исходный город. В какой последовательности следует посещать города, чтобы свести к минимуму пройденное расстояние? В теории графов эта задача называется *задачей коммивояжера*.

На рис. 14.13 представлено расположение городов и расстояния между ними. Как будет выглядеть кратчайший путь перемещения от A по всем городам и обратно в A? Обратите внимание: в задаче не требуется, чтобы каждая пара городов была соединена ребром.

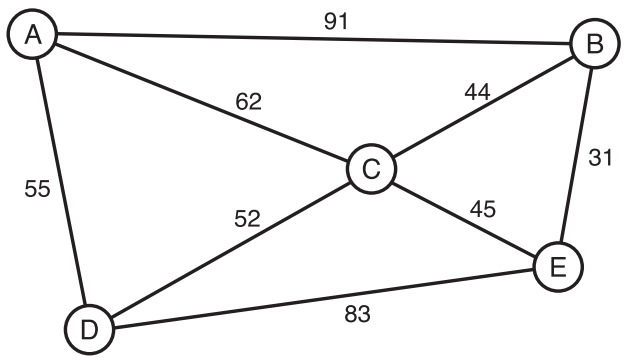

Рис. 14.13. Города и расстояния

Чтобы найти кратчайший маршрут, следует составить список всех возможных перестановок городов (A-B-C, B-C-A, C-B-A и т. д.), а также вычислить общее расстояние для каждой перестановки. Общая продолжительность маршрута ABCEDA равна 318. Маршрут ABCDEA невозможен из-за отсутствия ребра из E в A.

К сожалению, количество перестановок может быть очень большим: оно вычисляется как факториал количества городов (не считая исходного). Если требуется посетить 6 городов, то первый город выбирается из 6 вариантов, второй — из 5 вариантов, третий — из 4 и т. д; итого 6*5*4*3*2*1, или 720 возможных маршрутов. Даже для 50 городов решить задачу простым перебором нереально. Некоторые стратегии позволяют сократить количество проверяемых последовательностей, но они помогают лишь незначительно. Задача реализуется с использованием взвешенного графа: веса представляют расстояния между городами, а вершины — сами города. Граф может быть ненаправленным, если расстояние от A до B совпадает с расстоянием от B до A, как это обычно бывает при переездах. Если веса представляют стоимость авиабилетов, они могут изменяться в зависимости от направления; в таком случае используется направленный граф.

Гамильтоновы циклы

Задача, отчасти похожая на задачу коммивояжера, но более абстрактная — задача поиска гамильтонового цикла графа. Как упоминалось ранее, циклом называется путь, который начинается и заканчивается в одной вершине. Гамильтонов цикл посещает каждую вершину графа ровно один раз. В отличие от задачи коммивояжера, расстояния нас не интересуют; нужно лишь определить, существует ли такой цикл. На рис. 14.13 маршрут ABCEDA является гамильтоновым циклом, а маршрут ABCDEA — не является. Задача обхода доски ходом шахматного коня тоже является примером гамильтонова цикла (если предположить, что конь возвращается на исходную клетку).

Поиск гамильтонова цикла выполняется за то же время $O(N!)$, что и решение задачи коммивояжера. Высокая сложность в O-синтаксисе — например, 2^N и $N!$, возрастающая еще быстрее; 2^N — часто называется *экспоненциальной*.

Итоги

- Во взвешенном графе ребрам ставятся в соответствие числа, называемые весами. Они могут представлять расстояния, затраты, время и другие величины.
- Минимальное остовное дерево во взвешенном графе минимизирует вес ребер, необходимых для соединения всех вершин.
- Для построения минимального остовного дерева взвешенного графа может использоваться алгоритм на базе приоритетной очереди.
- Минимальное остовное дерево взвешенного графа моделирует реальные ситуации, например прокладку кабелей между городами.

- ✦ Задача выбора кратчайшего пути в невзвешенном графе заключается в определении минимального количества ребер между двумя вершинами.
- ✦ Для взвешенных графов при решении задачи выбора кратчайшего пути строится путь с минимальным суммарным весом ребер.
- ✦ Задача выбора кратчайшего пути для взвешенных графов может решаться при помощи алгоритма Дейкстры.
- ✦ Для больших разреженных графов алгоритмы обычно работают быстрее при представлении графа в формате списка смежности (вместо матрицы смежности).
- ✦ В задаче построения кратчайших путей для всех пар вершин определяется суммарный вес ребер между всеми парами вершин в графе. Для решения этой задачи может использоваться алгоритм Флойда.
- ✦ Некоторые алгоритмы выполняются за экспоненциальное время. Такие алгоритмы могут применяться лишь к графам с относительно небольшим количеством вершин.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Веса во взвешенном графе являются свойством _____ графа.
2. Во взвешенном графе минимальное остовное дерево стремится свести к минимуму:
   a) количество ребер от начальной до заданной вершины;
   b) количество ребер, соединяющих все вершины;
   c) суммарный вес ребер от начальной до заданной вершины;
   d) суммарный вес ребер, соединяющих все вершины.
3. Вес минимального остовного дерева зависит от выбора начальной вершины (Да/Нет).
4. Что удаляется из приоритетной очереди в алгоритме построения минимального остовного дерева?
5. В примере с кабельной сетью каждое ребро, добавляемое в минимальное остовное дерево, соединяет:
   a) начальную вершину со смежной вершиной;
   b) уже подключенный город с неподключенным городом;
   c) текущую вершину со смежной вершиной;
   d) два города с офисами.
6. Алгоритм построения минимального остовного дерева исключает ребро из списка, если оно ведет к вершине, которая _____ .

7. Задача выбора кратчайшего пути имеет смысл только для направленных графов (Да/Нет).
8. Алгоритм Дейкстры находит кратчайший путь:
   a) от одной заданной вершины до всех остальных вершин;
   b) от одной заданной вершины до другой заданной вершины;
   c) от всех вершин до всех остальных вершин, до которых можно добраться через одно ребро;
   d) от всех вершин до всех остальных вершин, до которых можно добраться через несколько ребер.
9. Правило алгоритма Дейкстры требует, чтобы в дерево всегда включалась вершина, ближайшая к начальной вершине (Да/Нет).
10. В примере с железнодорожными билетами периферийным является город:
    a) расстояние до которого известно, но расстояния от которого неизвестны;
    b) который включен в дерево;
    c) расстояние до которого известно и который включен в дерево;
    d) о котором ничего не известно.
11. В задаче построения кратчайших путей для всех пар вершин ищется кратчайший путь:
    a) от начальной вершины до всех остальных вершин;
    b) от каждой вершины до всех остальных вершин;
    c) от начальной вершины до всех остальных вершин, находящихся от нее на расстоянии одного ребра;
    d) от каждой вершины до всех остальных вершин, находящихся от нее на расстоянии одного и более ребра.
12. Алгоритм Флойда для взвешенных графов является тем, чем _____ является для невзвешенных графов.
13. Алгоритм Флойда использует представление графа в виде _____ .
14. За какое приблизительно время (в O-синтаксисе) решается задача обхода доски ходом шахматного коня?
15. Является ли маршрут ABCEDA на рис. 14.13 оптимальным решением задачи коммивояжера?

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Используйте приложение GraphW Workshop для построения минимального остовного дерева графа на рис. 14.6. Граф следует считать ненаправленным (то есть на стрелки не обращайте внимания).

2. Решите задачу выбора кратчайшего пути для графа на рис. 14.6 в приложении GraphDW Workshop, но вычислите для всех ребер новые веса, вычитая приведенные на рисунке веса из 100.
3. Нарисуйте граф из пяти вершин и пяти ребер. Реализуйте алгоритм Дейкстры для этого графа, используя карандаш и бумагу. На каждом шаге записывайте текущее состояние дерева и массива кратчайших путей.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

14.1. Измените программу `path.java` (см. листинг 14.2), чтобы она выводила таблицу минимальной стоимости перемещения от любой вершины до всех остальных вершин. Для этого придется внести изменения в методы, предполагающие, что начальной вершиной всегда является вершина A.

14.2. Мы рассмотрели варианты представления графов в виде матрицы смежности или списков смежности. Также для представления графа могут использоваться ссылки Java: в объекте `Vertex` хранится список ссылок на другие вершины, с которыми он соединен. Такое использование ссылок выглядит особенно логично в направленных графах, потому что ссылки «указывают» из одной вершины на другую. Напишите программу, реализующую эту схему. Метод `main()` пишется по аналогии с методом `main()` программы `path.java` (см. листинг 14.2): он тоже должен строить граф, изображенный на рис. 14.6, с использованием тех же вызовов `addVertex()` и `addEdge()`. Далее метод должен выводить таблицу связности графа, чтобы доказать, что граф построен верно. Вес каждого ребра необходимо где-то хранить. Одно из возможных решений — класс `Edge` с полями для веса ребра и вершины, на которой оно кончается. Для каждой вершины ведется список объектов `Edge` (то есть ребер, начинающихся от этой вершины).

14.3. Реализуйте алгоритм Флойда. Начните с программы `path.java` (см. листинг 14.2) и внесите в нее необходимые изменения. В частности, из программы можно удалить весь код поиска кратчайших путей. Недостижимые вершины, как и прежде, должны представляться бесконечностью. Это позволит вам обойтись без проверки на нуль при сравнении текущего веса с вычисленным. Веса всех возможных маршрутов должны быть меньше бесконечности. Метод `main()` должен поддерживать возможность графов любой сложности.

14.4. Реализуйте задачу коммивояжера (см. раздел «Неразрешимые задачи» этой главы). Несмотря на неразрешимость, вам удастся без особых проблем решить задачу для небольших значений *N*, например 10 и менее городов. Попробуйте использовать ненаправленный граф. Переберите все возможные последовательности городов методом «грубой силы». Для генерирования перестановок воспользуйтесь программой `anagram.java` (см. листинг 6.2) из главы 6, «Рекурсия». Для представления несуществующих ребер используйте бесконечные веса. В этом случае вам не

придется прерывать обработку последовательности с несуществующими ребрами; любой суммарный вес, превышающий бесконечность, обозначает невозможный маршрут. Также не беспокойтесь об исключении симметричных маршрутов (например, включайте в выходные данные и ABCDEA, и AEDCBA).

14.5. Напишите программу, которая находит и выводит все гамильтоновы циклы взвешенного ненаправленного графа.