Графы

Графы принадлежат к числу самых гибких и универсальных структур, используемых в программировании. Как правило, задачи, решаемые при помощи графов, существенно отличаются от задач, которые рассматривались нами ранее. В области типичного хранения данных, скорее всего, графы вам не понадобятся, но в некоторых областях — и притом очень интересных — они оказывают неоценимую помощь.

Наше знакомство с графами будет разделено на две главы. В этой главе рассматриваются алгоритмы невзвешенных графов, приводятся описания некоторых алгоритмов, для представления которых могут использоваться графы, а также двух приложений Workshop для их моделирования. В следующей главе будут рассмотрены более сложные алгоритмы, относящиеся к взвешенным графам.

Знакомство с графами

Граф как структура данных имеет много общего с деревом. Более того, в математическом смысле дерево является частным случаем графа. И все же в программировании графы используются не так, как деревья.

Архитектура структур данных, описанных ранее в книге, определялась применяемыми к ним алгоритмами. Например, двоичное дерево имеет строение, упрощающее поиск данных и вставку новых узлов. Ребра дерева представляют быстрый способ перехода от узла к узлу.

Напротив, строение графа часто определяется физической или абстрактной задачей. Например, узлы графа могут представлять города, а ребра — маршруты авиарейсов между этими городами. Или другой, более абстрактный пример: допустим, имеется некий проект, завершение которого требует выполнения ряда задач. В графе узлы могут представлять задачи, а направленные ребра определяют последовательность их выполнения. В обоих случаях строение графа определяется конкретной ситуацией.

Прежде чем двигаться дальше, необходимо упомянуть, что при описании графов узлы обычно называются *вершинами*. Возможно, это связано с тем, что терминология графов сформировалась намного раньше — несколько столетий назад в области математики. Деревья в большей степени привязаны к практическому программированию. Тем не менее оба термина являются более или менее равноправными.

Определения

На рис. 13.1, *а* изображена упрощенная карта автострад в окрестностях Сан-Хосе (штат Калифорния). На рис. 13.1, *б* представлен граф, моделирующий дорожную сеть.

На графе кружки соответствуют дорожным развязкам, а прямые линии, которыми эти кружки соединяются, — дорожным сегментам. Кружки являются *вершинами*, а линии — *ребрами* графа. Вершины обычно каким-то образом помечаются — часто для пометки используются буквы, как в этом случае. Каждое ребро ограничивается двумя вершинами, находящимися на его концах.

Граф не пытается представить географические координаты отдельных городов; он только моделирует отношения между вершинами и ребрами, то есть какие ребра соединяют те или иные вершины. Граф не имеет отношения к физическим расстояниям или направлениям. Кроме того, одно ребро может представлять несколько автострад — как, например, ребро от вершины I к H, представляющее дороги 101, 84 и 280. Важен сам факт соединения одной развязки с другой (или его отсутствия), а не конкретные дороги.

Смежность

Две вершины называются *смежными*, если они соединены одним ребром. Так, на рис. 13.1 вершины I и G являются смежными, а вершины I и F — нет. Вершины, смежные с некоторой вершиной, называются ее *соседями*. Например, соседями G являются I, H и F.

Пути

Путь представляет собой последовательность вершин. На рис. 13.1 показан путь от вершины B к вершине J, проходящий через вершины A и E; этот путь можно обозначить BAEJ. Между двумя вершинами может существовать более одного пути; например, от B к J также ведет путь BCDJ.

Связные графы

Граф называется *связным*, если от каждой вершины к любой другой вершине ведет хотя бы один путь (рис. 13.2, *а*). Но если «отсюда дотуда не добраться» (как говорят фермеры в Вермонте городским пижонам, спрашивающим дорогу), граф становится *несвязным* — пример показан на рис. 13.2, *б*.

Несвязный граф состоит из нескольких областей. На рис. 13.2, *б* одна из таких областей состоит из вершин A и B, а другая — из вершин C и D.

Для простоты алгоритмы, описанные в этой главе, рассчитаны на работу со связными графами или отдельными областями несвязных графов. Как правило, после незначительных изменений эти алгоритмы смогут работать и с несвязными графами.

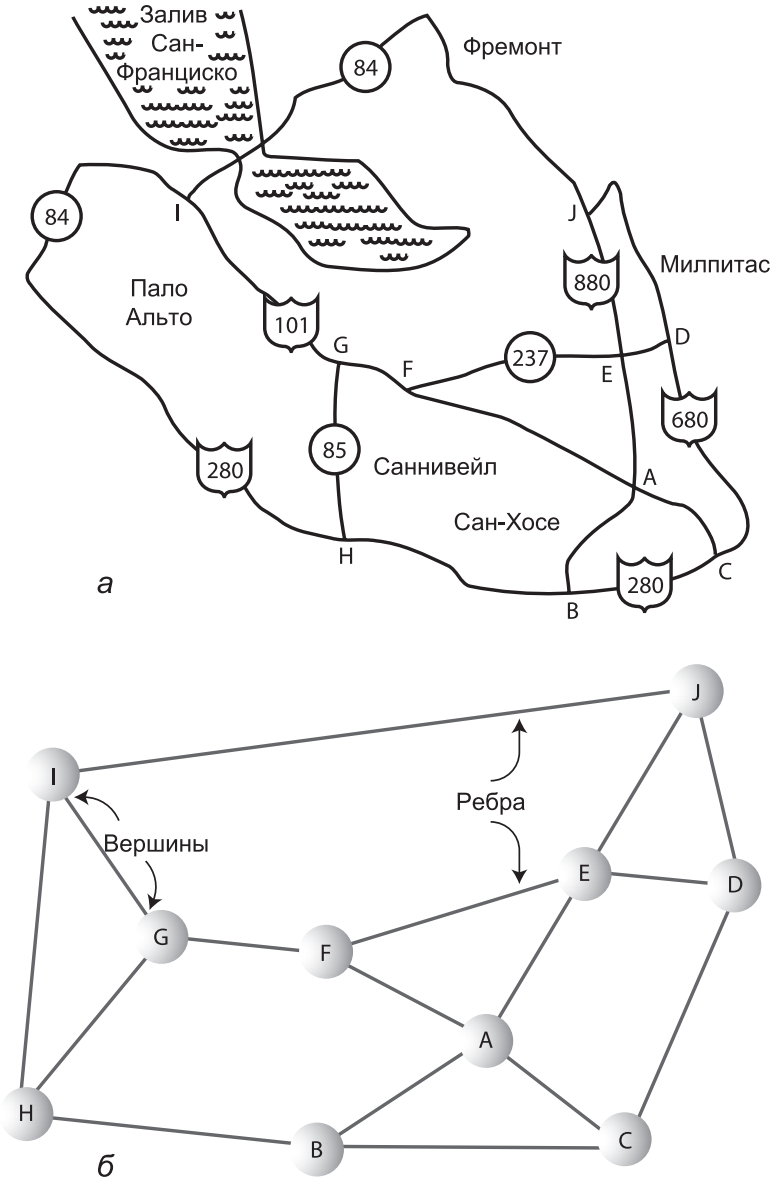


Рис. 13.1. Географическая карта (*а*) и графы (*б*)

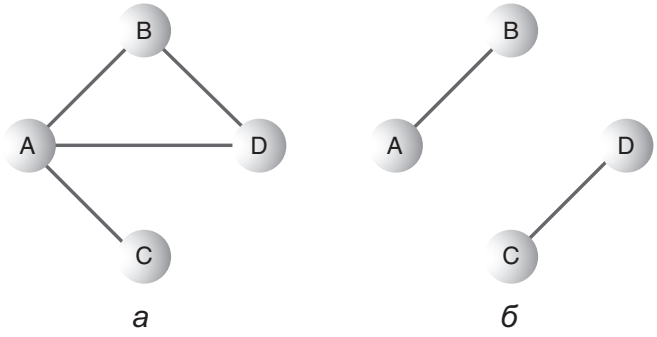


Рис. 13.2. Связные (*а*) и несвязные (*б*) графы

Направленные и взвешенные графы

На рис. 13.1 и 13.2 изображены *ненаправленные* графы. Иначе говоря, ребра таких графов не имеют направления; перемещения по ним возможны в обоих направлениях, то есть алгоритм может перейти как от вершины A к вершине B, так и от

вершины B к вершине A. Ненаправленные графы хорошо моделируют дорожную сеть, потому что по дорогам обычно можно ездить в обоих направлениях.

Однако графы также часто используются для моделирования ситуаций, в которых перемещение по ребру возможно только в одном направлении — от A к B, но не от B к A, как на улице с односторонним движением. Такие графы называются *направленными* (или *ориентированными*). Разрешенное направление обычно обозначается стрелкой на конце ребра.

На некоторых графах ребрам присваиваются *веса* — числа, представляющие физическое расстояние между двумя вершинами, время перехода от вершины к вершине или затраты на такой переход (скажем, в примере с авиарейсами). Такие графы, называемые *взвешенными*, подробно рассматриваются в следующей главе. А эта глава начинается с описания простых ненаправленных, невзвешенных графов; потом мы перейдем к направленным невзвешенным графам.

Приведенные определения ни в коей мере не исчерпывают всей терминологии, относящейся к графам. Другие термины будут вводиться по мере изложения материала.

Немного истории

Одним из первых математиков, занимавшихся теорией графов, был Леонард Эйлер (начало XVIII века). В частности, он решил знаменитую задачу с «кенигсбергскими мостами» — в городе Кенигсберг был остров с семью мостами (рис. 13.3, *а*). В популярной задаче требовалось найти путь пересечения всех семи мостов, в котором каждый мост проходился бы только один раз. Мы не будем описывать решение Эйлера; он доказал, что такого пути не существует. Для нас важно то, что его решение было основано на представлении задачи в виде графа. Вершины графа моделировали участки земли, а ребра — мосты (рис. 13.3, *б*). Возможно, это был первый случай использования графа для моделирования задачи из реального мира.

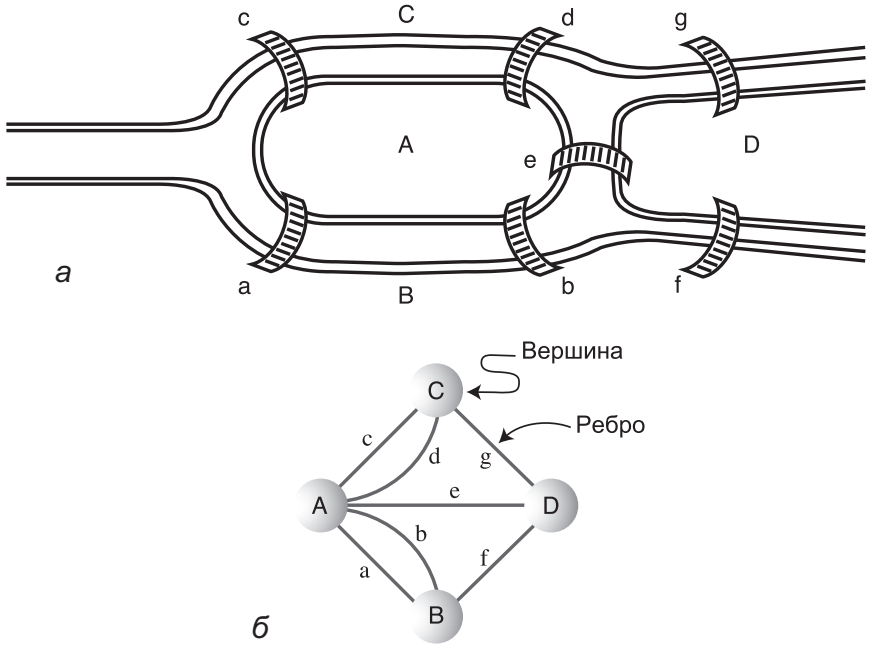


Рис. 13.3. Кенигсбергские мосты: *а* — карта; *б* — граф

Представление графа в программе

До изобретения компьютеров Эйлер и другие математики изучали графы с абстрактной точки зрения. Нас больше интересует практическое представление графов в компьютерных программах. Какая структура данных лучше подойдет для моделирования графа? Начнем с вершин, а затем перейдем к ребрам.

Вершины

В очень абстрактной программе для работы с графом можно пронумеровать вершины от 0 до $N - 1$ (где N — количество вершин). Для хранения вершин переменные вообще не нужны, потому что их полезность зависит от связей с другими вершинами.

Однако на практике вершины обычно представляют реально существующие объекты, а объекты описываются полями данных. Например, если вершина представляет город в модели сети воздушного сообщения, в ней может храниться название города, высота над уровнем моря, координаты и т. д. Таким образом, для представления вершин обычно удобно использовать объекты соответствующего класса. В наших примерах программ для каждой вершины хранится только буквенная метка (например, A) и флаг, используемый поисковыми алгоритмами (об этом ниже). Определение класса вершины графа `Vertex` выглядит так:

```
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean wasVisited;

   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
   }  // Конец класса Vertex
```

Объекты вершин можно хранить в массиве и обращаться к ним по индексу. В наших примерах они будут храниться в массиве с именем `vertexList`. Для хранения вершин также можно использовать список или другую структуру данных. Впрочем, структура для хранения вершин выбирается только для удобства. Она никак не связана со способом соединения вершин ребрами. Для моделирования ребер графа понадобится другой механизм.

Ребра

Как было показано в главе 9, «Красно-черные деревья», в программах могут использоваться разные способы представления деревьев. В основном мы изучали представление, в котором каждый узел содержит ссылки на своих потомков, но также было рассмотрено представление на базе массива, в котором позиция узла в массиве определяет его связи с другими узлами. В главе 12, «Пирамиды», была представлена возможность использования массива для представления особой разновидности деревьев — так называемых пирамид.

Однако граф обычно не имеет такой жесткой структуры, как дерево. В двоичном дереве каждый узел имеет не более двух потомков, а в графе каждая вершина может быть соединена с произвольным количеством других вершин. Например, на рис. 13.2, *а* вершина A соединена с тремя другими вершинами, а вершина C соединена только с одной.

Для моделирования подобной нежесткой структуры нужен другой способ представления ребер. При работе с графами обычно применяются две структуры: *матрица смежности* и *список смежности*. (Напомним, что вершины называются смежными, если они соединены одним ребром.)

Матрица смежности

Матрица смежности представляет собой двумерный массив, элементы которого обозначают наличие связи между двумя вершинами. Если граф содержит N вершин, то матрица смежности представляет собой массив $N \times N$. В таблице 13.1 приведена матрица смежности для графа на рис. 13.2, *а*.

Таблица 13.1. Матрица смежности

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Заголовками как строк, так и столбцов являются метки вершин. Ребро между двумя вершинами обозначается 1; при отсутствии ребра элемент матрицы равен 0. (Также можно использовать логические значения `true/false`.) Как видно из таблицы, вершина A является смежной со всеми тремя остальными вершинами, вершина B — с вершинами A и D, вершина C — только с A, а вершина D — с A и B. В этой матрице «соединение» вершины с самой собой обозначается 0, поэтому диагональ, идущая из левого верхнего в правый нижний угол (от A–A до D–D), заполнена нулями. Элементы диагонали не несут полезной информации, поэтому в них с таким же успехом можно записать 1, если это окажется удобнее в вашей программе.

Обратите внимание: треугольная часть матрицы над диагональю является зеркальным отражением части, находящейся под ней; оба треугольника содержат одни и те же данные. Подобная избыточность может показаться неэффективной, но в большинстве языков программирования не существует удобных средств для представления треугольных массивов, поэтому проще смириться с избыточностью. Соответственно при добавлении ребра в граф в матрице смежности необходимо изменить два элемента вместо одного.

Список смежности

В другом способе представления ребер графа используется список смежности (речь идет о связанных списках, рассматривавшихся в главе 5). Вообще говоря,

список смежности скорее является массивом списков (а иногда списком списков). Каждый отдельный список содержит информацию о том, какие вершины являются смежными по отношению к заданной. В таблице 13.2 приведены списки смежности для графа на рис. 13.2, *а*.

Таблица 13.2. Списки смежности

Вершина	Список смежных вершин
A	B—>C—>D
B	A—>D
C	A
D	A—>B

В этой таблице знаком —> обозначается связь в списке. Элементами списка являются вершины. В нашем примере вершины каждого списка упорядочены в алфавитном порядке, хотя на самом деле это не обязательно. Не путайте содержимое списков смежности с путями. Список смежности показывает, какие вершины являются смежными по отношению к заданной (то есть находятся от нее на расстоянии одного ребра); он не является описанием пути между вершинами.

Позднее мы разберемся, в каких ситуация предпочтительнее использовать матрицу смежности, а когда лучше выбрать список смежности. Во всех приложениях Workshop этой главы используется матрица смежности, хотя в некоторых случаях списковое решение оказывается более эффективным.

Добавление вершин и ребер в граф

Чтобы включить в граф новую вершину, создайте новый объект вершины оператором `new` и вставьте его в массив вершин `vertexList`. В реальной программе объект вершины может содержать много полей данных, но в наших примерах для простоты используется только один символ. Таким образом, создание вершины выглядит примерно так:

```
vertexList[nVerts++] = new Vertex('F');
```

Команда вставляет в граф вершину F. В переменной `nVerts` хранится текущее количество вершин в графе. Способ добавления ребра зависит от выбранного представления (матрица смежности или списки смежности). Допустим, в программе используется матрица смежности, и в граф добавляется ребро между вершинами 1 и 3. Эти числа соответствуют индексам массива `vertexList`, в котором хранятся вершины. При создании матрица смежности `adjMat` заполняется нулями. Чтобы вставить в нее ребро, выполните следующие команды:

```
adjMat[1][3] = 1;
adjMat[3][1] = 1;
```

В программе со списком смежности в список 3 добавляется вершина 1, а в список вершины 1 добавляется вершина 3.

Класс Graph

Класс Graph содержит методы для создания списка вершин и матрицы смежности, а также для добавления вершин и ребер в объект Graph:

```
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Массив вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
// -------------------------------------------------------------
   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] = 0;
      }
// -------------------------------------------------------------
   public void addVertex(char lab)    // В аргументе передается метка
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   }  // Конец класса Graph
```

В классе Graph вершины определяются своим индексом в списке vertexList. Большинство приведенных методов уже рассматривалось ранее. Вывод вершины сводится к выводу ее символьной метки.

Информация, хранящаяся в матрице смежности (или списке смежности), является локальной для конкретной вершины, а именно она сообщает, какие вершины соединены одним ребром с заданной вершиной. Чтобы получить ответы на более глобальные вопросы о взаимном расположении вершин, необходимо использовать специальные алгоритмы. Начнем с обхода.

Обход

Одной из основных операций, выполняемых с графами, является определение всех вершин, достижимых от заданной вершины. Представьте, что вы пытаетесь определить, до каких городов в США можно добраться пассажирским поездом от Канзас-Сити (с возможными пересадками). До одних городов добраться можно. Другие города недоступны, потому что в них нет железнодорожного сообщения. Третьи недоступны даже при наличии железной дороги, например из-за того, что их узкоколейное полотно не стыкуется со стандартной шириной колеи вашей железнодорожной сети.

Или другая ситуация, в которой тоже может потребоваться найти все вершины, к которым можно перейти от заданной вершины. Представьте, что вы проектируете печатную плату вроде тех, которые установлены в вашем компьютере. На плате размещаются различные компоненты — в основном микросхемы, контакты которых вставляются в отверстия на плате. Микросхемы закрепляются посредством пайки, а электрическое соединение их контактов с другими контактами осуществляется при помощи *дорожек* — тонких металлических полосок, проходящих по поверхности платы (рис. 13.4).

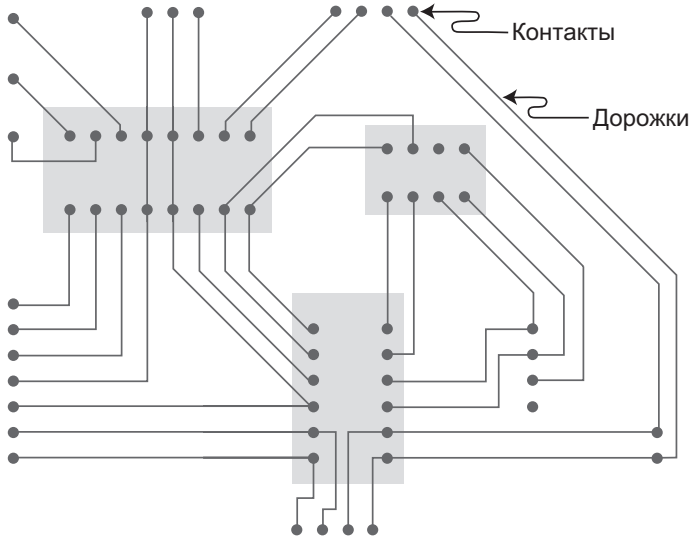


Рис. 13.4. Контакты и дорожки на печатной плате

На графе контакты могут представляться вершинами, а дорожки — ребрами. На печатной плате обычно имеется много электрических цепей, не связанных друг с другом, поэтому граф ни в коем случае не обязан быть связным. А следовательно, в процессе проектирования будет полезно построить граф и воспользоваться им для определения того, какие контакты подключены к текущей цепи.

Предположим, такой граф построен. Теперь нужен алгоритм, который начинается от заданной вершины и систематически перемещается по ребрам к другим вершинам так, что после завершения его работы будут посещены все вершины, соединенные с исходной. Как в главе 8, «Двоичные деревья», под «посещением» понимается выполнение с вершиной некоторой операции, например вывод.

Существует два основных способа обхода графов: *обход в глубину* и *обход в ширину*. Оба способа в конечном итоге обеспечивают перебор всех соединенных вершин. Обход в глубину реализуется на базе стека, а обход в ширину реализуется на базе очереди. Как вы вскоре убедитесь, это приводит к обходу вершин графа в разном порядке.

Обход в глубину

Алгоритм обхода в глубину хранит в стеке информацию о том, куда следует вернуться при достижении «тупика». Мы рассмотрим пример, немного поэкспериментируем с приложением GraphN Workshop, а потом разберем код конкретной реализации.

Пример

Для объяснения принципов обхода в глубину будет использован граф на рис. 13.5. Цифры на графе обозначают порядок посещения вершин.

Обход в глубину начинается с выбора отправной точки — в нашем примере это вершина A. Затем алгоритм выполняет три операции: посещает вершину, заносит ее в стек и помечает для предотвращения повторных посещений.

Затем алгоритм переходит к любой вершине, смежной с A, которая еще не посещалась ранее. Будем считать, что вершины выбираются в алфавитном порядке; значит, это будет вершина B. Алгоритм посещает B, помечает вершину и заносит в стек.

Что теперь? Текущей вершиной является B, поэтому алгоритм делает то же, что и прежде: он переходит к смежной вершине, которая еще не посещалась ранее. В нашем примере это вершина F. Назовем этот процесс правилом 1.

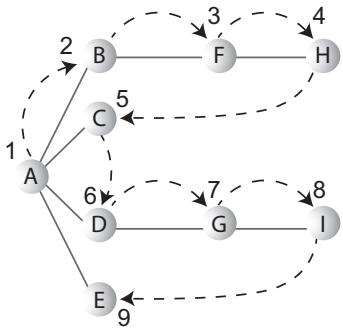


Рис. 13.5. Обход в глубину

ПРАВИЛО 1

Посетить смежную вершину, не посещавшуюся ранее, пометить ее и занести в стек.

Применение правила 1 снова приводит к вершине H. Однако на этот раз необходимо сделать что-то другое, потому что в графе не осталось непосещенных вершин, смежных с H. На помощь приходит правило 2.

ПРАВИЛО 2

Если выполнение правила 1 невозможно, извлечь вершину из стека.

В соответствии с этим правилом вершина H извлекается из стека, в результате чего алгоритм возвращается к F. У вершины F тоже нет непосещенных смеж-

ных вершин; она извлекается из стека. То же происходит с вершиной B. Теперь в стеке остается только вершина A. Однако у A имеются непосещенные смежные вершины, поэтому алгоритм посещает следующую вершину C. Но поскольку эта вершина тоже является тупиковой, она извлекается из стека, и алгоритм возвращается к A. Далее алгоритм посещает D, G и I, а затем извлекает их при достижении тупика в I. Происходит возврат к A. Алгоритм посещает вершину E и снова возвращается к A.

Но на этот раз у вершины A не осталось непосещенных соседей, и она извлекается из стека. В стеке не остается элементов, и вступает в действие правило 3.

ПРАВИЛО 3

Если выполнение правил 1 и 2 невозможно, обход закончен.

В таблице 13.3 представлено содержимое стека на разных стадиях этого процесса для рис. 13.5.

Таблица 13.3. Содержимое стека при обходе в глубину

Событие	Стек
Посещение A	A
Посещение B	AB
Посещение F	ABF
Посещение H	ABFH
Извлечение H	ABF
Извлечение F	AB
Извлечение B	A
Посещение C	AC
Извлечение C	A
Посещение D	AD
Посещение G	ADG
Посещение I	ADGI
Извлечение I	ADG
Извлечение G	AD
Извлечение D	A
Посещение E	AE
Извлечение E	A
Извлечение A	
Обход завершен	

Содержимое стека описывает путь от исходной вершины до текущей. Удаляясь от исходной вершины, алгоритм заносит вершины в стек, а при возвращении

к ней — извлекает из стека. В нашем примере вершины будут посещаться в порядке ABFHCDGIE.

Таким образом, алгоритм обхода в глубину стремится как можно быстрее удалиться от исходной вершины и возвращается к ней только при достижении тупика. Если понимать под «глубиной» расстояние от исходной вершины, становится понятно, откуда взялось название алгоритма.

Аналогия

Обход в глубину можно сравнить с поиском пути в лабиринте. Лабиринт состоит из узких проходов (ребер) и перекрестков (вершин), на которых они встречаются друг с другом.

Допустим, некто заблудился в лабиринте. Он знает, что выход существует, и пытается систематично обойти весь лабиринт в поисках выхода. К счастью, у странника имеется клубок веревки и фломастер. Он начинает с ближайшего перекрестка и идет по случайно выбранному проходу, разматывая веревку. На следующем перекрестке он снова сворачивает на другой случайно выбранный проход и т. д., пока не окажется в тупике. Странник возвращается, сматывая веревку, пока не окажется у последнего перекрестка. Здесь он помечает тупиковый путь, чтобы не заходить в него в будущем, и опробует другой проход. Когда все пути от текущего перекрестка будут проверены, странник возвращается к предыдущему перекрестку, и все повторяется снова. Веревка в этом примере является аналогом стека: она используется для «запоминания» пути к текущей позиции.

Приложение GraphN Workshop и обход в глубину

Кнопка `DFS` в приложении GraphN Workshop позволяет поэкспериментировать с обходом в глубину.

Запустите приложение. В исходном состоянии граф не содержит ни вершин, ни ребер — рабочая область представляет собой пустой прямоугольник. Вершины создаются двойным щелчком в нужной позиции. Первой вершине автоматически присваивается метка A, второй — метка B и т. д. Цвета вершин выбираются случайным образом. Чтобы создать ребро, перетащите указатель мыши между вершинами. На рис. 13.6 изображен граф с рис. 13.5, созданный в приложении.

Возможность удаления отдельных ребер или вершин не предусмотрена, так что если вы допустите ошибку, придется начать все заново — кнопка `New` уничтожает все созданные вершины и ребра (после подтверждения пользователя). Кнопка `View` выводит матрицу смежности созданного графа (рис. 13.7). Повторное нажатие кнопки `View` возвращает режим отображения графа.

Алгоритм обхода в глубину запускается кнопкой `DFS`. Вам будет предложено щелкнуть (один раз!) на вершине, с которой должен начаться обход.

Возьмите за образец граф на рис. 13.6 или создайте более или менее сложный граф по своему усмотрению. После непродолжительных экспериментов вы сможете предсказать, что алгоритм сделает на следующем шаге (если граф не отличается особой сложностью).

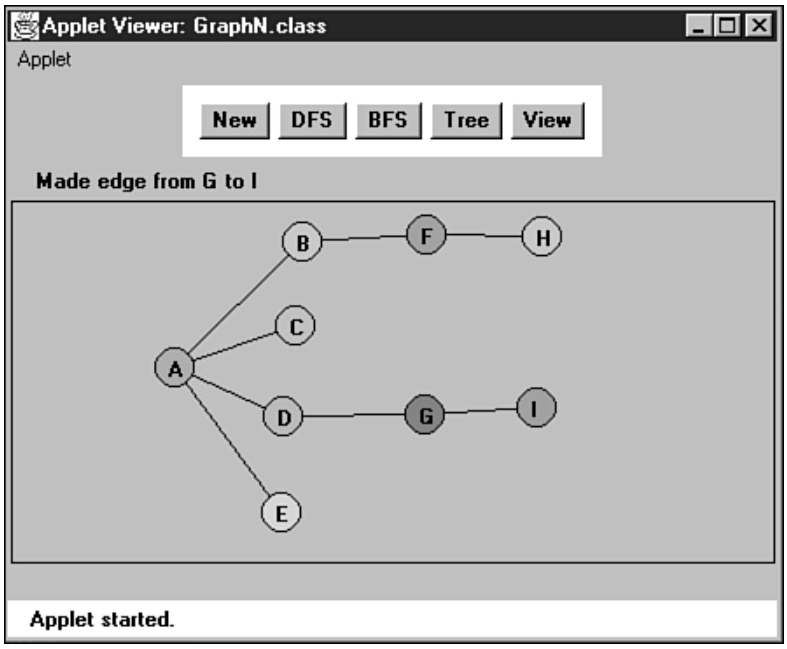
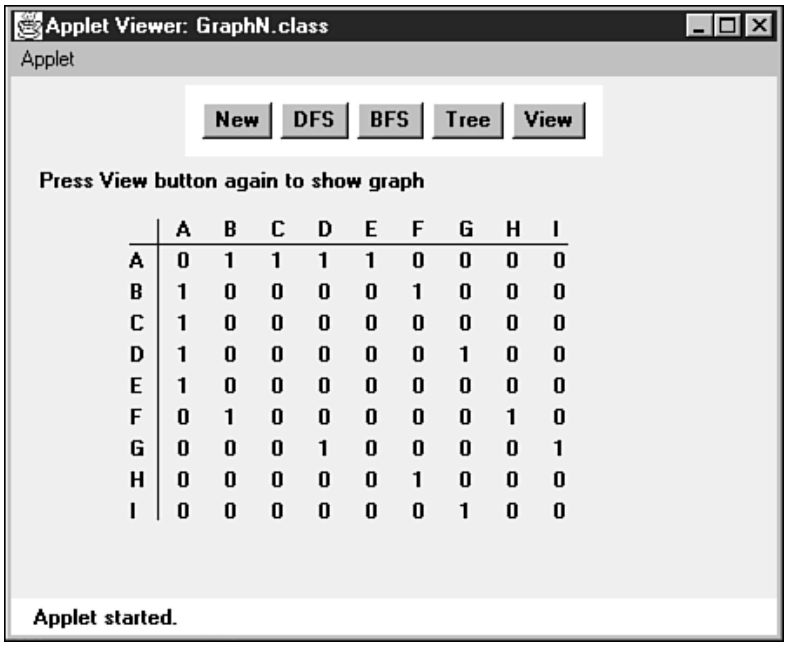


Рис. 13.6. Приложение GraphN Workshop



	A	B	C	D	E	F	G	H	I
A	0	1	1	1	1	0	0	0	0
B	1	0	0	0	0	1	0	0	0
C	1	0	0	0	0	0	0	0	0
D	1	0	0	0	0	0	1	0	0
E	1	0	0	0	0	0	0	0	0
F	0	1	0	0	0	0	0	1	0
G	0	0	0	1	0	0	0	0	1
H	0	0	0	0	0	1	0	0	0
I	0	0	0	0	0	0	1	0	0

Рис. 13.7. Вывод матрицы смежности в GraphN Workshop

При выполнении для несвязного графа алгоритм обходит только вершины, соединенные с исходной вершиной.

Реализация на языке Java

Центральное место в алгоритме обхода в глубину занимает поиск вершин, смежных по отношению к заданной и ранее не посещавшихся. Как решить эту задачу? При помощи матрицы смежности. Программа просматривает строку заданной вершины и отбирает столбцы, содержащие 1; номер столбца определяет номер смежной вершины. Далее программа проверяет, посещалась ли вершина ранее. Если вершина

не посещалась, значит, мы нашли искомое — следующую вершину для посещения. Если в строке не осталось ни одной вершины с 1 в ячейке матрицы (смежной), для которой не был бы установлен флаг посещения, значит, непосещенных вершин, смежных по отношению к данной, не осталось. Эта логика запрограммирована в методе getAdjUnvisitedVertex():

```
// Метод возвращает непосещенную вершину, смежную по отношению к v
public int getAdjUnvisitedVertex(int v)
   {
   for(int j=0; j<nVerts; j++)
      if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
         return j;               // Возвращает первую найденную вершину
   return -1;                    // Таких вершин нет
   }
```

Теперь у нас есть все необходимое для написания метода dfs() класса Graph, выполняющего обход в глубину. Код наглядно реализует три правила, перечисленных ранее. Цикл продолжает выполняться, пока в стеке остается хотя бы один элемент. На каждой итерации:

1) проверяется элемент на вершине стека методом peek();
2) делается попытка найти непосещенного соседа этой вершины;
3) если поиск окажется неудачным, элемент извлекается из стека;
4) если вершина будет найдена, алгоритм посещает ее и заносит в стек.

Код метода dfs():

```
public void dfs()  // Обход в глубину
   {                                // Алгоритм начинает с вершины 0
   vertexList[0].wasVisited = true;  // Пометка
   displayVertex(0);                // Вывод
   theStack.push(0);                // Занесение в стек

   while( !theStack.isEmpty() )     // Пока стек не опустеет
      {
      // Получение непосещенной вершины, смежной к текущей
      int v = getAdjUnvisitedVertex( theStack.peek() );
      if(v == -1)                   // Если такой вершины нет,
         theStack.pop();            // элемент извлекается из стека
      else                          // Если вершина найдена
         {
         vertexList[v].wasVisited = true;  // Пометка
         displayVertex(v);                 // Вывод
         theStack.push(v);                 // Занесение в стек
         }
      }

      // Стек пуст, работа закончена
      for(int j=0; j<nVerts; j++)    // Сброс флагов
         vertexList[j].wasVisited = false;
   }
```

В конце метода dfs() сбрасываются флаги wasVisited, чтобы dfs() можно было снова вызвать в будущем. Стек к этому моменту уже пуст и очищать его не нужно.

Теперь у нас имеются все необходимые компоненты класса Graph. Следующий фрагмент создает объект графа, включает в него несколько вершин и ребер, а затем выполняет обход в глубину:

```
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);     // AB
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(3, 4);     // DE

System.out.print("Visits: ");
theGraph.dfs();             // Обход в глубину
System.out.println();
```

Граф, построенный этим фрагментом, изображен на рис. 13.8. Результат выполнения выглядит так:

```
Visits: ABCDE
```

Попробуйте изменить этот код, чтобы он строил другой граф по вашему выбору; запустите его и проверьте, правильно ли выполняется обход в глубину.

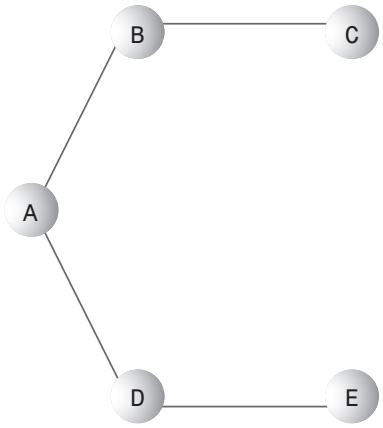


Рис. 13.8. Граф, используемый в программах dfs.java и bfs.java

Программа dfs.java

В листинге 13.1 приведен код программы dfs.java с методом dfs(). В ней используется измененная версия класса StackX из главы 4, «Стеки и очереди».

Листинг 13.1. Программа dfs.java

```
// dfs.java
// Обход в глубину
// Запуск программы: C>java DFSApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
// ----------------------------------------------------------
   public StackX()           // Конструктор
      {
      st = new int[SIZE];    // Создание массива
      top = -1;
      }
```

```
// --------------------------------------------------------------
   public void push(int j)   // Размещение элемента в стеке
      { st[++top] = j; }
// --------------------------------------------------------------
   public int pop()          // Извлечение элемента из стека
      { return st[top--]; }
// --------------------------------------------------------------
   public int peek()         // Чтение с вершины стека
      { return st[top]; }
// --------------------------------------------------------------
   public boolean isEmpty()  // true, если стек пуст
      { return (top == -1); }
// --------------------------------------------------------------
}  // Конец класса StackX
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // метка (например, 'A')
   public boolean wasVisited;
// --------------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
// --------------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private StackX theStack;
// --------------------------------------------------------------
   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] = 0;
      theStack = new StackX();
      }
// --------------------------------------------------------------
```

продолжение ⇩

Листинг 13.1 (*продолжение*)

```
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void dfs()  // Обход в глубину
      {                                 // Алгоритм начинает с вершины 0
      vertexList[0].wasVisited = true;  // Пометка
      displayVertex(0);                 // Вывод
      theStack.push(0);                 // Занесение в стек

      while( !theStack.isEmpty() )      // Пока стек не опустеет
         {
         // Получение непосещенной вершины, смежной к текущей
         int v = getAdjUnvisitedVertex( theStack.peek() );
         if(v == -1)                    // Если такой вершины нет,
            theStack.pop();             // элемент извлекается из стека
         else                           // Если вершина найдена
            {
            vertexList[v].wasVisited = true;  // Пометка
            displayVertex(v);                 // Вывод
            theStack.push(v);                 // Занесение в стек
            }
         }

      // Стек пуст, работа закончена
      for(int j=0; j<nVerts; j++)    // Сброс флагов
         vertexList[j].wasVisited = false;
      }
// -------------------------------------------------------------
   // Метод возвращает непосещенную вершину, смежную по отношению к v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;               // Возвращает первую найденную вершину
      return -1;                    // Таких вершин нет
      }
```

```
// --------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class DFSApp
   {
   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);     // AB
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Visits: ");
      theGraph.dfs();             // Обход в глубину
      System.out.println();
      }
   }  // Конец класса DFSApp
////////////////////////////////////////////////////////////////
```

Обход в глубину в программировании игр

Обход в глубину часто используется в программировании игр (и сходных ситуаций в реальном мире). В типичной игре существует выбор между несколькими действиями. Каждый вариант приводит к дополнительному набору вариантов, те ведут к своим вариантам и т. д. — постоянно расширяющийся граф вариантов. Точка выбора соответствует вершине, а выбранный вариант — ребру графа, которое ведет к следующей точке выбора.

Вспомните игру «крестики-нолики». Первый игрок может сделать один из 9 возможных ходов. Его противник отвечает одним из 8 возможных ходов и т. д. Каждый ход ведет к следующей группе вариантов противника, каждый из которых ведет к группе вариантов для вас и т. д., пока не будет заполнена последняя клетка.

Как выбрать ход в текущей позиции? Можно мысленно представить себе ход, рассмотреть возможные ответы противника, потом ваши ответы на них и т. д. Ход выбирается в зависимости от того, какой из путей ведет к оптимальному результату. В простых играх вроде «крестики-нолики» количество возможных ходов ограничено, что позволяет проанализировать все пути до конца игры. Полный анализ путей определяет начальный ход. Ситуация может быть представлена в виде графа с одним узлом, представляющим первый ход. Узел соединен с восемью узлами, представляющими возможные ответы противника; каждый из этих узлов соединяется с семью узлами, представляющими ваши ответы и т. д. Каждый путь

от начального узла к конечному состоит из девяти узлов. Чтобы выполнить полный анализ, необходимо построить девять графов, по одному для каждого начального хода.

Даже в этой простой игре количество путей оказывается неожиданно большим. Если забыть об упрощениях, связанных с симметрией, девять графов образуют 9*8*7*6*5*4*3*2*1 путей. Результат равен 9! (факториал 9), или 362 880. В такой игре, как шахматы, где количество возможных ходов намного больше, даже самые мощные компьютеры (такие, как «Deep Blue» фирмы IBM) не способны «видеть» позицию до конца партии. Они только проходят путь до определенной глубины, а затем оценивают позицию на доске и определяют, выглядит ли она более перспективной по сравнению с другими вариантами.

Естественный способ анализа таких ситуаций в программе основан на использовании обхода в глубину. В каждом узле алгоритм принимает решение, что делать дальше, как в методе `getAdjUnvisitedVertex()` программы `dfs.java` (см. листинг 13.1). Если в графе еще остались непосещенные узлы (точки выбора), алгоритм заносит текущий узел в стек и переходит к следующему. Если в некоторой позиции следующий ход невозможен (`getAdjUnvisitedVertex()` возвращает −1), алгоритм «отступает назад», извлекая узел из стека, и проверяет, не осталось ли в этой позиции неисследованных вариантов.

Последовательность ходов хорошо представляется в виде дерева, узлы которого соответствуют отдельным ходам. Первый ход представлен корневым узлом. В «крестиках-ноликах» после первого хода возможно восемь вторых ходов, каждый из которых представлен узлом, соединенным с корнем. После восьми вторых ходов возможно семь третьих ходов, представленных узлами, соединенных с узлами второго кода. Построенное таким образом дерево состоит из 9! возможных путей от корня к листовым узлам. Оно называется *деревом игры*.

На самом деле количество ветвей в дереве игры несколько меньше максимального, потому что выигрыш часто достигается до заполнения всех клеток. Но несмотря на это, полный анализ игры «крестики-нолики» остается относительно сложным, а ведь эта игра очень проста по сравнению с многими другими (скажем, шахматами).

Лишь часть путей дерева игры ведет к выигрышу. Другие пути ведут к победе противника. При достижении такого результата алгоритм должен отступить к предыдущему узлу и проверить другой путь. Анализ дерева продолжается до тех пор, пока не будет найден путь к успешному результату. Далее остается сделать первый ход на этом пути.

Обход в ширину

Как было показано ранее, алгоритм обхода в глубину старается как можно быстрее удалиться от исходной вершины на максимальное расстояние. При обходе в ширину, напротив, алгоритм стремится держаться как можно ближе к исходной вершине. Он посещает все вершины, смежные с исходной, и только после этого отходит дальше. Такая разновидность обхода реализуется на базе очереди (вместо стека).

Пример

На рис. 13.9 изображен тот же граф, что и на рис. 13.5, но на этот раз применяется обход в ширину. Как и прежде, цифрами обозначается порядок посещения вершин.

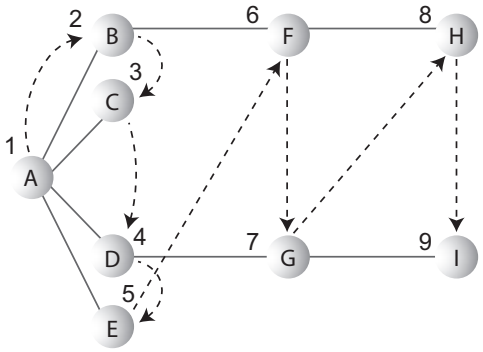


Рис. 13.9. Обход в ширину

Обход начинается с вершины A; алгоритм посещает ее и делает текущей. Далее применяются следующие правила.

ПРАВИЛО 1

Посетить следующую вершину, не посещавшуюся ранее, смежную с текущей вершиной, пометить ее и занести в очередь.

ПРАВИЛО 2

Если выполнение правила 1 невозможно, извлечь вершину из очереди и сделать ее текущей вершиной.

ПРАВИЛО 3

Если выполнение правил 1 и 2 невозможно, обход закончен.

Таким образом, сначала посещаются все вершины, смежные с A. Алгоритм вставляет каждую вершину в очередь при посещении. Так посещаются вершины A, B, C, D и E. На этой стадии очередь (от начала к концу) содержит вершины BCDE.

Других непосещенных и смежных с A вершин не осталось, поэтому алгоритм извлекает B из очереди и ищет вершины, смежные с B. Алгоритм находит вершину F и вставляет ее в очередь. Других непосещенных вершин, смежных с B, не осталось, поэтому вершина C извлекается из очереди. И снова непосещенных смежных вершин нет, алгоритм извлекает D из очереди и посещает G. У D не осталось других смежных непосещенных вершин, из очереди извлекается E. Теперь в очереди остались вершины FG. Алгоритм извлекает F и посещает H, затем извлекает G и посещает I.

В результате очередь содержит вершины HI. Но когда алгоритм извлекает каждую из них и не находит смежных непосещенных вершин, очередь остается пустой, и работа алгоритма завершается. В таблице 13.4 приведено содержимое очереди на разных стадиях процесса обхода.

Таблица 13.4. Содержимое очереди при обходе в ширину

Событие	Очередь
Посещение A	
Посещение B	B
Посещение C	BC
Посещение D	BCD
Посещение E	BCDE
Извлечение B	CDE
Посещение F	CDEF
Извлечение C	DEF
Извлечение D	EF
Посещение G	EFG
Извлечение E	FG
Извлечение F	G
Посещение H	GH
Извлечение G	H
Посещение I	HI
Извлечение H	I
Извлечение I	
Обход завершен	

В каждый промежуточный момент в очереди хранятся вершины, которые были посещены в процессе обхода, но соседи которых не были полностью исследованы. (Сравните с обходом в ширину, при котором в стеке хранится маршрут от исходной вершины к текущей.) Узлы посещаются в порядке ABCDEFGHI.

Приложение GraphN Workshop и обход в ширину

Поэкспериментируйте с обходом в ширину в приложении GraphN Workshop (кнопка `BFS`). Как и в предыдущем случае, вы можете взять за образец рис. 13.9 или построить собственный граф.

Обратите внимание на сходство и различие между обходом в ширину и обходом в глубину.

Последовательность обхода в ширину можно сравнить с кругами, расходящимися по воде от брошенного камня, или для читателей, сведущих в эпидемиологии, с распространением вируса гриппа от города к городу. Сначала посещаются все вершины, соединенные с исходной, затем все вершины, удаленные от исходной на два ребра и т. д.

Реализация на языке Java

Метод `bfs()` класса `Graph` похож на метод `dfs()`, но стек в нем заменен очередью, а один цикл — вложенными циклами. Внешний цикл проверяет наличие элементов в очереди, а внутренний последовательно перебирает каждого непосещенного соседа текущей вершины. Код метода:

```
public void bfs()                  // Обход в ширину
   {                               // Алгоритм начинает с вершины 0
   vertexList[0].wasVisited = true; // Пометка
   displayVertex(0);               // Вывод
   theQueue.insert(0);             // Вставка в конец очереди
   int v2;

   while( !theQueue.isEmpty() )    // Пока очередь не опустеет
      {
      int v1 = theQueue.remove();  // Извлечение вершины в начале очереди
                                   // Пока остаются непосещенные соседи
      while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
         {                                  // Получение вершины
         vertexList[v2].wasVisited = true;  // Пометка
         displayVertex(v2);                 // Вывод
         theQueue.insert(v2);               // Вставка
         }
      }

   // Очередь пуста, обход закончен
   for(int j=0; j<nVerts; j++)             // Сброс флагов
      vertexList[j].wasVisited = false;
   }
```

Для графа, использованного ранее в программе `dfs.java` (см. рис. 13.8), метод `bfs.java` выводит следующий результат:

```
Visits: ABDCE
```

Программа bfs.java

Программа `bfs.java`, приведенная в листинге 13.2, имеет много общего с `dfs.java`; отличия невелики: вместо класса `StackX` в ней используется класс `Queue` (слегка измененный по сравнению с версией из главы 4), а метод `dfs()` заменен методом `bfs()`.

Листинг 13.2. Программа bfs.java

```
// bfs.java
// Обход в ширину
// Запуск программы: C>java BFSApp
////////////////////////////////////////////////////////////////
class Queue
   {
   private final int SIZE = 20;
```

продолжение ⇩

Листинг 13.2 (*продолжение*)

```
   private int[] queArray;
   private int front;
   private int rear;
// ---------------------------------------------------------
   public Queue()            // Конструктор
      {
      queArray = new int[SIZE];
      front = 0;
      rear = -1;
      }
// ---------------------------------------------------------
   public void insert(int j) // Вставка элемента в конец очереди
      {
      if(rear == SIZE-1)
         rear = -1;
      queArray[++rear] = j;
      }
// ---------------------------------------------------------
public int remove()       // Извлечение элемента в начале очереди
   {
   int temp = queArray[front++];
   if(front == SIZE)
      front = 0;
   return temp;
}
// ---------------------------------------------------------
   public boolean isEmpty()  // true, если очередь пуста
      {
      return ( rear+1==front || (front+SIZE-1==rear) );
      }
// ---------------------------------------------------------
   }  // Конец класса Queue
////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean wasVisited;
// ---------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
// ---------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
```

```
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private Queue theQueue;
// -----------------
   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] = 0;
      theQueue = new Queue();
      }
// -------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void bfs()                   // Обход в ширину
      {                                // Алгоритм начинает с вершины 0
      vertexList[0].wasVisited = true; // Пометка
      displayVertex(0);                // Вывод
      theQueue.insert(0);              // Вставка в конец очереди
      int v2;

      while( !theQueue.isEmpty() )     // Пока очередь не опустеет
         {
         int v1 = theQueue.remove();  // Извлечение вершины в начале очереди
                                      // Пока остаются непосещенные соседи

         while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
            {                                 // Получение вершины
            vertexList[v2].wasVisited = true;  // Пометка
            displayVertex(v2);                // Вывод
```

продолжение ⇩

Листинг 13.2 (*продолжение*)

```
            theQueue.insert(v2);                // Вставка
            }
         }

   // Очередь пуста, обход закончен
      for(int j=0; j<nVerts; j++)             // Сброс флагов
         vertexList[j].wasVisited = false;
      }
// ---------------------------------------------------------
   // Метод возвращает непосещенную вершину, смежную по отношению к v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;               // Возвращает первую найденную вершину
      return -1;                    // Таких вершин нет
      }
// ---------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////
class BFSApp
   {
   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);     // AB
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Visits: ");
      theGraph.bfs();             // Обход в ширину
      System.out.println();
      }
   }
////////////////////////////////////////////////////////////
```

Обход в ширину обладает одним интересным свойством: он сначала находит все вершины, находящиеся на расстоянии одного ребра от начальной вершины, затем все вершины на расстоянии двух ребер и т. д. Это свойство может пригодиться при поиске кратчайшего пути от начальной вершины к заданной. Запустите обход в ширину, и при обнаружении заданной вершины вы точно знаете, что построенный путь является кратчайшим путем к вершине. Если бы более короткий путь существовал, то алгоритм обхода в ширину нашел бы его ранее.

Минимальные остовные деревья

Представьте, что вы проектируете печатную плату (вроде изображенной на рис. 13.4), и хотите свести количество дорожек к минимуму. Иначе говоря, между контактами не должно быть лишних соединений, которые только занимают место и усложняют проводку других цепей.

Для решения этой задачи понадобится алгоритм, который бы для любого связного набора контактов и дорожек (вершин и ребер в терминологии графов) удалял все лишние дорожки. Результатом его работы является граф с минимальным количеством ребер, необходимых для соединения вершин. Например, граф на рис. 13.10, *а* состоит из пяти вершин с избыточными ребрами, а на рис. 13.10, *б* изображены те же пять вершин с минимальным количеством ребер, необходимым для их соединения. Эти ребра образуют *минимальное остовное дерево* (MST, Minimum Spanning Tree).

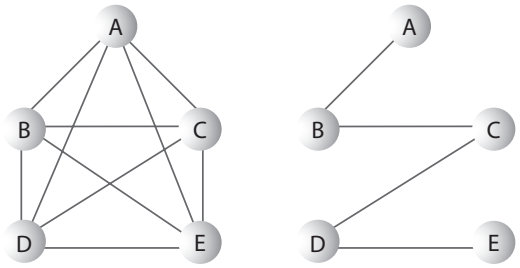


Рис. 13.10. Минимальное остовное дерево: *а* — избыточные ребра; *б* — минимальное количество ребер

Для заданного набора вершин существует много возможных минимальных остовных деревьев. На рис. 13.10, *б* обозначены ребра AB, BC, CD и DE, но с таким же успехом можно было составить минимальное остовное дерево из ребер AC, CE, ED и DB. Математически одаренные читатели наверняка заметят, что количество ребер E в минимальном остовном дереве всегда на единицу меньше количества вершин V:

$$E = V - 1.$$

Стоит напомнить, что длина ребер нас не интересует. Мы хотим определить не минимальную физическую длину, а минимальное количество ребер. (Ситуация изменится в следующей главе, когда речь пойдет о взвешенных графах.)

Алгоритм построения минимального остовного дерева почти идентичен алгоритму обхода. Он может базироваться как на обходе в глубину, так и на обходе в ширину. В нашем примере будет использоваться обход в глубину.

Это может показаться неожиданным, но обход в глубину с сохранением ребер, посещенных при поиске, приводит к автоматическому построению минимального остовного дерева. Единственное различие между методом построения минимального остовного дерева `mst()`, который будет приведен ниже, и методом обхода в глубину `dfs()`, приведенным ранее, заключается в том, что метод `mst()` должен каким-то образом сохранять посещенные ребра.

Приложение GraphN Workshop

Кнопка Tree в приложении GraphN Workshop строит минимальное остовное дерево для созданного вами графа. Опробуйте ее с разными графами. Вы увидите, что алгоритм выполняет те же действия, что и при обходе в глубину (кнопка DFS). Однако при использовании кнопки Tree ребра, включаемые в минимальное остовное дерево, темнеют. После завершения алгоритма приложение убирает все остальные ребра, а на графе остается только минимальное остовное дерево. Последнее нажатие кнопки Tree восстанавливает исходный граф на тот случай, если вы захотите снова воспользоваться им.

Реализация построения минимального остовного дерева на языке Java

Код метода mst() выглядит так:

```
while( !theStack.isEmpty() )       // Пока стек не опустеет
   {                               // Извлечение элемента из стека
   int currentVertex = theStack.peek();
   // get next unvisited neighbor
   int v = getAdjUnvisitedVertex(currentVertex);
   if(v == -1)                     // Если соседей больше нет,
      theStack.pop();              // извлечь элемент из стека
   else                            // Сосед существует
      {
      vertexList[v].wasVisited = true;  // Пометка
      theStack.push(v);                 // Занесение в стек
                                        // Вывод ребра
      displayVertex(currentVertex);     // От currentVertex
      displayVertex(v);                 // к v
      System.out.print(" ");
      }
   }

   // Стек пуст, работа закончена
   for(int j=0; j<nVerts; j++)          // Сброс флагов
      vertexList[j].wasVisited = false;
}
```

Как видите, код метода очень похож на код dfs(). В секции else выводится текущая вершина и следующий непосещенный сосед. Эти две вершины определяют ребро, по которому алгоритм перемещается для получения новой вершины. Такие ребра составляют минимальное остовное дерево.

В методе main() программы mst.java фрагмент построения графа выглядит так:

```
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);     // AB
theGraph.addEdge(0, 2);     // AC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(0, 4);     // AE
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(1, 3);     // BD
theGraph.addEdge(1, 4);     // BE
theGraph.addEdge(2, 3);     // CD
theGraph.addEdge(2, 4);     // CE
theGraph.addEdge(3, 4);     // DE
```

Построенный граф показан на рис. 13.10, *а*. Когда метод `mst()` завершит свою работу, в графе остаются только четыре ребра, показанных на рис. 13.10, *б*. Результат выполнения программы `mst.java` выглядит так:

```
Minimum spanning tree: AB BC CD DE
```

Как уже говорилось выше, это лишь одно из многих возможных минимальных остовных деревьев, которые могут быть построены для приведенного графа. Например, если начать с другой вершины, алгоритм построит другое дерево. К таким же последствиям приведут небольшие вариации в коде — скажем, если обход в методе `getAdjUnvisitedVertex()` будет начинаться не с начала, а с конца списка `vertexList[]`.

Минимальное остовное дерево естественным образом строится на базе обхода в глубину, потому что этот алгоритм посещает все узлы, но только по одному разу. Он никогда не переходит к узлу, который уже был посещен ранее, а все его перемещения по ребрам ограничены строгой необходимостью. Таким образом, путь алгоритма по графу должен образовывать минимальное остовное дерево.

Программа mst.java

В листинге 13.3 приведена программа `mst.java`. Она отличается от программы `dfs.java` только методом `mst()` и построением графа в `main()`.

Листинг 13.3. Программа mst.java

```
// mst.java
// Построение минимального остовного дерева
// Запуск программы: C>java MSTApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
// ----------------------------------------------------------
   public StackX()            // Конструктор
      {
```

продолжение ⇩

Листинг 13.3 (*продолжение*)

```
      st = new int[SIZE];    // Создание массива
      top = -1;
      }
// ---------------------------------------------------------
   public void push(int j)   // Размещение элемента в стеке
      { st[++top] = j; }
// ---------------------------------------------------------
   public int pop()          // Извлечение элемента из стека
      { return st[top--]; }
// ---------------------------------------------------------
   public int peek()         // Чтение с вершины стека
      { return st[top]; }
// ---------------------------------------------------------
   public boolean isEmpty()  // true, если стек пуст
      { return (top == -1); }
// ---------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean wasVisited;
// ---------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
// ---------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private StackX theStack;
// ---------------------------------------------------------
   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] = 0;
```

```
      theStack = new StackX();
      }
// ------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// ------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// ------------------------------------------------------------
   public void mst()  // Построение минимального остовного дерева
      {                                 //
      vertexList[0].wasVisited = true;  // Пометка
      theStack.push(0);                 // Занесение в стек
      while( !theStack.isEmpty() )      // Пока стек не опустеет
         {                              // Извлечение элемента из стека
         int currentVertex = theStack.peek();
                                        // Получение следующего соседа
         int v = getAdjUnvisitedVertex(currentVertex);
         if(v == -1)                    // Если соседей больше нет,
            theStack.pop();             // извлечь элемент из стека
         else                           // Сосед существует
            {
            vertexList[v].wasVisited = true;  // Пометка
            theStack.push(v);                 // Занесение в стек
                                              // Вывод ребра
            displayVertex(currentVertex);     // От currentVertex
            displayVertex(v);                 // к v
            System.out.print(" ");
            }
         }

         // Стек пуст, работа закончена
         for(int j=0; j<nVerts; j++)          // Сброс флагов
            vertexList[j].wasVisited = false;
      }
// ------------------------------------------------------------
   // Метод возвращает непосещенную вершину, смежную по отношению к v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
      return -1;
   }
```

продолжение ⇩

Листинг 13.3 (*продолжение*)

```
// --------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class MSTApp
   {
   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);     // AB
      theGraph.addEdge(0, 2);     // AC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(0, 4);     // AE
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(1, 3);     // BD
      theGraph.addEdge(1, 4);     // BE
      theGraph.addEdge(2, 3);     // CD
      theGraph.addEdge(2, 4);     // CE
      theGraph.addEdge(3, 4);     // DE

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

Граф, построенный в `main()`, представляет собой пятиконечную звезду, в которой каждый узел соединен со всеми остальными узлами. Программа выводит следующий результат:

```
Minimum spanning tree: AB BC CD DE
```

Топологическая сортировка с направленными графами

Топологическая сортировка также принадлежит к числу операций, которые могут моделироваться при помощи графов. Она удобна в ситуациях, в которых требуется расставить некоторые элементы или события в определенном порядке. Рассмотрим пример.

Пример

В институтах и колледжах студент не может выбрать любой учебный курс по своему усмотрению. У некоторых курсов имеются необходимые условия — обязательное прохождение других учебных курсов. С другой стороны, некоторые курсы могут быть необходимым условием для получения ученой степени в определенной области. На рис. 13.11 изображена (условная) структура курсов, необходимых для получения ученой степени по математике.

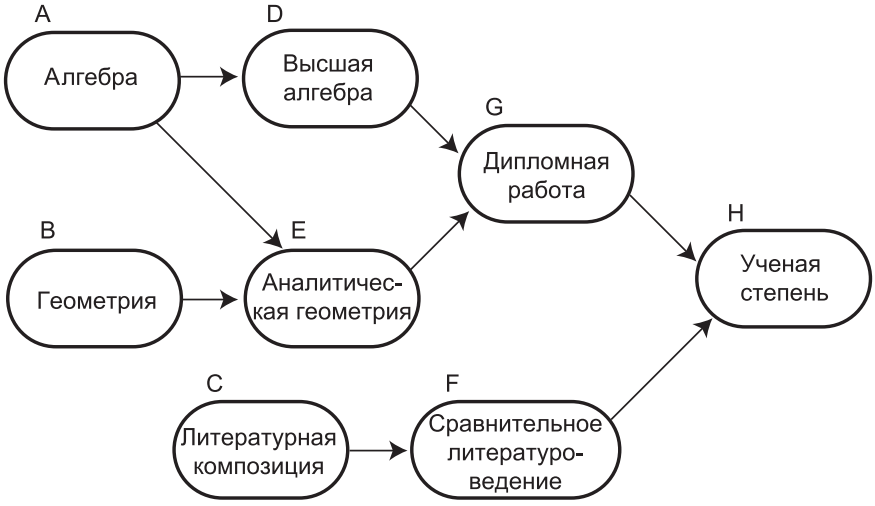


Рис. 13.11. Обязательные курсы

Чтобы получить ученую степень, необходимо сдать дипломную работу и (из-за давления со стороны факультета английского языка) сравнительное литературоведение. Однако сдача дипломной работы возможна лишь при условии сдачи высшей алгебры и аналитической геометрии, а курс сравнительного литературоведения доступен лишь после прохождения курса литературной композиции. Кроме того, геометрия является обязательным условием для курса аналитической геометрии, а алгебра — для высшей алгебры и аналитической геометрии.

Направленные графы

Как видно из рис. 13.11, для представления подобной структуры курсов можно воспользоваться графом. Однако граф этот должен обладать свойством, с которым мы еще не сталкивались: каждое его ребро должно обладать направлением. Такие графы называются *направленными*. В направленном графе переходы по ребрам возможны только в одном из двух направлений. На схеме графа такие направления обозначаются стрелками.

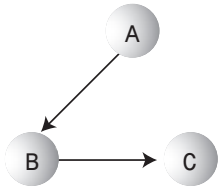


Рис. 13.12. Направленный граф

В программе основное различие между ненаправленным и направленным графом заключается в том, что ребро направленного графа обозначается только одним элементом матрицы смежности. На рис. 13.12

изображен небольшой направленный граф, а в табл. 13.5 приведена его матрица смежности.

Таблица 13.5. Матрица смежности направленного графа

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

Каждое ребро представлено одним значением 1. Заголовки строк определяют начало направленного ребра, а заголовки столбцов — его конец. Таким образом, ребро от вершины A в B представлено значением 1 на пересечении строки A и столбца B. Если бы направление ребра изменилось так, что оно будет вести от B в A, то «единица» появится на пересечении строки B и столбца A.

Для ненаправленных графов, как упоминалось ранее, половина матрицы смежности является симметричным отражением другой половины, а половина ячеек оказывается лишней. Но для взвешенного графа каждая ячейка матрицы смежности несет полезную информацию. Две треугольные части матрицы не симметричны.

В направленном графе метод для добавления ребра состоит всего из одной команды:

```
public void addEdge(int start, int end)  // Направленный граф
   {
   adjMat[start][end] = 1;
   }
```

вместо двух команд, необходимых для ненаправленного графа.

Если же для представления графа используется матрица смежности, то в списке A присутствует вершина B, но (в отличие от ненаправленного графа) в списке B вершины A нет.

Топологическая сортировка

Допустим, вы составляете список всех курсов, необходимых для получения ученой степени, на основе рис. 13.11. Курсы необходимо выстроить в порядке их прохождения. Ученая степень находится на последней позиции этого списка, который может выглядеть так:

```
BAEDGCFH
```

Граф, упорядоченный подобным образом, называется *топологически отсортированным*. Каждый учебный курс, который должен быть пройден ранее некоторого другого курса, предшествует ему в списке.

Вообще говоря, существует много разных конфигураций, соответствующих требованиям порядка прохождения курсов. Например, можно начать с литературных курсов C и F:

```
CFBAEDGH
```

Эта последовательность тоже выполняет все обязательные условия. Также существует много других возможных решений. Результаты, генерируемые алгоритмом построения топологических сортировок, зависят от выбранного метода и технических подробностей кода.

Топологическая сортировка также применяется для моделирования других ситуаций, например при планировании операций. Скажем, при сборке машины тормоза должны устанавливаться раньше колес, а двигатель должен быть собран до его закрепления на раме. Фирмы-производители автомобилей используют графы для моделирования тысяч операций производственного процесса, чтобы обеспечить их выполнение в правильном порядке.

Моделирование планирования заданий с использованием графов называется *анализом критических путей*. Применение взвешенного графа (см. следующую главу) позволяет включить в граф информацию о времени, необходимом для выполнения различных задач проекта. Например, при помощи такого графа можно вычислить минимальное время, необходимое для выполнения всего проекта.

Приложение GraphD Workshop

Приложение GraphD Workshop моделирует направленные графы. Оно работает почти так же, как GraphN, но рядом с каждым ребром выводится точка, обозначающая его направление. Будьте внимательны: направление ребра определяется направлением, в котором перетаскивается указатель мыши при его создании. На рис. 13.13 изображено приложение GraphD Workshop, использованное для моделирования графа учебных курсов на рис. 13.11.

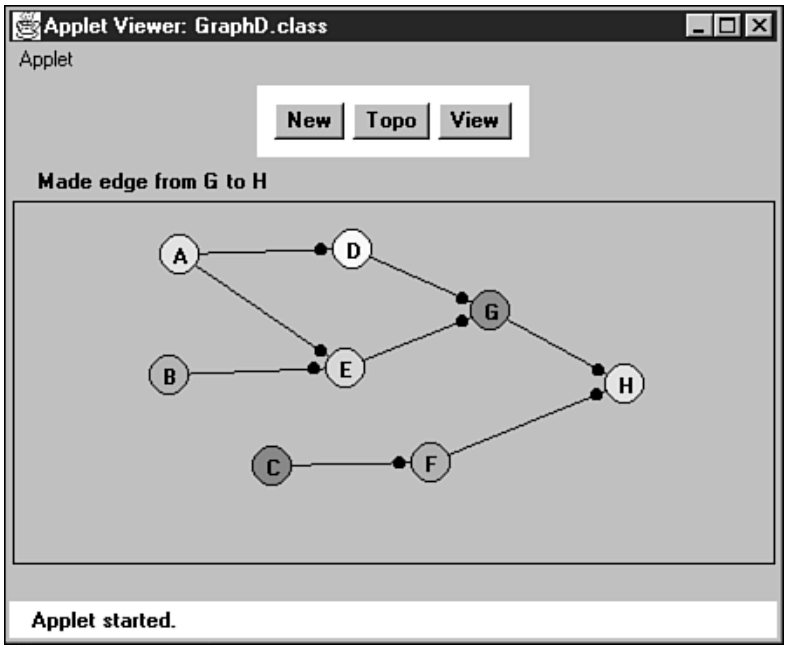


Рис. 13.13. Приложение GraphD Workshop

Алгоритм топологической сортировки основан на нестандартной, но простой идее. Он состоит из двух шагов.

ШАГ 1

Найти вершину, не имеющую преемников.

Преемниками вершины называются вершины, расположенные непосредственно «за ней», то есть соединенные с ней ребром, указывающим в их направлении. Если в графе существует ребро, ведущее от A к B, то B является преемником A. На рис. 13.11 преемники существуют у всех вершин, кроме H.

ШАГ 2

Удалить эту вершину из графа и вставить ее метку в начало списка.

Шаги 1 и 2 повторяются до тех пор, пока из графа не будут удалены все вершины. На этой стадии список состоит из всех вершин, упорядоченных в топологическом порядке.

За тем, как работает этот процесс, можно понаблюдать в приложении GraphD. Постройте граф на рис. 13.11 (или любой другой граф на ваше усмотрение); вершины создаются двойным щелчком, а ребра — перетаскиванием. Далее многократно нажимайте на кнопку `Topo`. С удалением очередной вершины из графа ее метка размещается в списке под графом.

Удаление вершины может показаться слишком радикальным действием, но оно занимает центральное место в алгоритме. Алгоритм не может найти вторую удаляемую вершину, пока первая вершина не будет удалена. При необходимости данные графа (список вершин и матрицу смежности) можно сохранить во временных переменных и восстановить после завершения сортировки, как это делается в приложении GraphD.

Работа алгоритма основана на том факте, что вершина, не имеющая преемников, должна быть последней в топологическом порядке. Сразу же после ее удаления появляется другая вершина, не имеющая преемников; она должна стать предпоследней в результатах сортировки и т. д.

Алгоритм топологической сортировки работает как со связными, так и с несвязными графами. Несвязные графы моделируют ситуацию с наличием несвязанных целей, например получением ученой степени по математике с одновременным получением медицинского сертификата по оказанию экстренной помощи.

Циклы и деревья

Одну из разновидностей графов, с которыми не может справиться алгоритм топологической сортировки, составляют графы с циклами. Что такое цикл? Это путь, который завершается в той точке, с которой он начинается. На рис. 13.14 путь B-C-D-B образует цикл. (Обратите внимание: путь A-B-C-A циклом не является, потому что переход от C к A невозможен.)

Цикл моделирует «порочный круг», при котором прохождение курса B является обязательным условием для курса C, прохождение курса C — для курса D, а курс D обязателен для B.

Граф, не содержащий циклов, называется *деревом*. Двоичные и многопутевые деревья, представленные ранее в этой книге, являются деревьями в этом смысле. Однако понятие дерева в теории графов более универсально по сравнению с двоичными и многопутевыми деревьями, имеющими фиксированное количество потомков. В графе вершина дерева может соединяться с произвольным количеством других вершин (при условии, что при этом не образуются циклы).

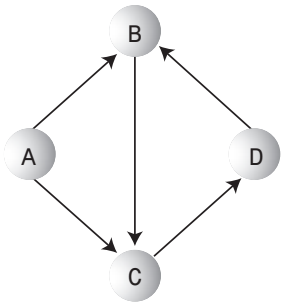


Рис. 13.14. Граф с циклом

Наличие циклов в ненаправленном графе проверяется достаточно легко. Если граф с N узлами содержит более $N - 1$ ребер, в нем заведомо существуют циклы. Чтобы убедиться в этом, попробуйте нарисовать граф из N узлов и N ребер, не имеющий циклов.

Топологическая сортировка должна применяться к направленным графам, не содержащим циклов. Такие графы называются *направленными ациклическими графами*.

Реализация на языке Java

Реализация метода `topo()`, выполняющего топологическую сортировку, на языке Java выглядит так:

```
public void topo()            // Топологическая сортировка
   {
   int orig_nVerts = nVerts;  // Сохранение количества вершин

   while(nVerts > 0)          // Пока в графе остаются вершины
   {
      // Получение вершины без преемников или -1
      int currentVertex = noSuccessors();
      if(currentVertex == -1)       // В графе есть цикл
         {
         System.out.println("ERROR: Graph has cycles");
         return;
         }
      // Вставка метки вершины в массив (начиная с конца)
      sortedArray[nVerts-1] = vertexList[currentVertex].label;

      deleteVertex(currentVertex);  // Удаление вершины
      }
   // Все вершины удалены, вывод содержимого sortedArray
   System.out.print("Topologically sorted order: ");
   for(int j=0; j<orig_nVerts; j++)
      System.out.print( sortedArray[j] );
   System.out.println("");
   }
```

Вся работа выполняется в цикле while, который продолжает выполняться до тех пор, пока количество вершин не сократится до 0. Метод работает по следующему алгоритму:

1. Вызов noSuccessors() ищет любую вершину, не имеющую преемников.
2. Если вершина найдена, ее метка помещается в конец массива sortedArray[], а сама вершина удаляется из графа.
3. Если вершина не найдена, значит, в графе существует цикл.

Последняя удаляемая вершина должна находиться на первом месте в списке, поэтому метки вершин помещаются в sortedArray, начиная с конца, и продвигаются к началу с уменьшением значения nVerts (количества вершин в графе).

Если в графе остаются вершины, но у всех вершин имеются преемники, значит, граф содержит цикл, а алгоритм выводит сообщение и завершает работу. При отсутствии циклов происходит выход из цикла while, а список из sortedArray выводится в порядке топологической сортировки вершин.

Метод noSuccessors() использует матрицу смежности для поиска вершин, не имеющих преемников. Внешний цикл for перебирает строки, последовательно просматривая все вершины. Для каждой вершины во внутреннем цикле for перебираются столбцы в поисках значения 1. Если такой столбец будет найден, значит, у вершины имеется преемник (существует ребро, ведущее от текущей вершины к другой). При обнаружении 1 происходит выход из внутреннего цикла для проверки следующей вершины.

Только если будет найдена строка, в которой нет ни одной 1, алгоритм знает, что он обнаружил вершину без преемников; в таком случае возвращается номер этой строки. Если вершина не обнаружена, возвращается –1. Код метода noSuccessors():

```
public int noSuccessors()  // Метод возвращает вершину, не имеющую
   {                       // преемников (или -1, если ее нет)
   boolean isEdge;  // Ребро в матрице adjMat, ведущее от row в column

   for(int row=0; row<nVerts; row++)  // Для каждой вершины
      {
      isEdge = false;                 // Проверка ребер
      for(int col=0; col<nVerts; col++)
         {
         if( adjMat[row][col] > 0 )   // Если существует ребро
            {                         // от текущей вершины
            isEdge = true;
            break;                    // У вершины имеется преемник
            }                         // Переход к проверке
         }                            // другой вершины
      if( !isEdge )                   // Если ребер нет,
         return row;                  // то нет и преемников
      }
   return -1;                         // Вершина не найдена
   }
```

Удаление вершины выполняется просто, не считая некоторых подробностей. Вершина удаляется из массива vertexList[], а находящиеся выше нее вершины

сдвигаются вниз, заполняя свободное место. Аналогичным образом строка и столбец вершины удаляются из матрицы смежности, а строки и столбцы выше и справа от них сдвигаются вниз и влево для заполнения свободных мест. Эти операции выполняются методами `deleteVertex()`, `moveRowUp()` и `moveColLeft()`, с которыми можно ознакомиться в полном коде `topo.java` (листинг 13.4). Вообще говоря, для этого алгоритма более эффективным было бы представление графа в формате списков смежности, но его реализация отвлечет нас от основной темы.

Метод `main()` строит граф на рис. 13.10 вызовами методов, аналогичными тем, которые приводились ранее. Метод `addEdge()`, как уже было сказано, вставляет в матрицу смежности только одно число, потому что граф является направленным.

Код метода `main()`:

```
public static void main(String[] args)
   {
   Graph theGraph = new Graph();
   theGraph.addVertex('A');    // 0
   theGraph.addVertex('B');    // 1
   theGraph.addVertex('C');    // 2
   theGraph.addVertex('D');    // 3
   theGraph.addVertex('E');    // 4
   theGraph.addVertex('F');    // 5
   theGraph.addVertex('G');    // 6
   theGraph.addVertex('H');    // 7
   theGraph.addEdge(0, 3);     // AD
   theGraph.addEdge(0, 4);     // AE
   theGraph.addEdge(1, 4);     // BE
   theGraph.addEdge(2, 5);     // CF
   theGraph.addEdge(3, 6);     // DG
   theGraph.addEdge(4, 6);     // EG
   theGraph.addEdge(5, 7);     // FH
   theGraph.addEdge(6, 7);     // GH
   theGraph.topo();            // Сортировка
   }
```

После того как граф будет построен, `main()` сортирует его вызовом `topo()` и выводит результат. Выходные данные выглядят так:

```
Topologically sorted order: BAEDGCFH
```

Конечно, метод `main()` можно изменить, чтобы он строил другие графы.

Полный код программы topo.java

Мы уже рассмотрели большинство методов `topo.java`. В листинге 13.4 приведен полный код.

Листинг 13.4. Программа topo.java

```
// topo.java
// Топологическая сортировка
// Запуск программы: C>java TopoApp
```

продолжение ⇩

Листинг 13.4 (*продолжение*)

```
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
// ----------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      { label = lab; }
   } Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private char sortedArray[];
// ----------------------------------------------------------
   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] = 0;
      sortedArray = new char[MAX_VERTS];  // Метки отсортированных вершин
   }
// ----------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// ----------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      }
// ----------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// ----------------------------------------------------------
   public void topo()  // Топологическая сортировка
      {
      int orig_nVerts = nVerts;  // Сохранение количества вершин

      while(nVerts > 0)  // Пока в графе остаются вершины
```

```
         {
         // Получение вершины без преемников или -1
         int currentVertex = noSuccessors();
         if(currentVertex == -1)       // В графе есть цикл
            {
            System.out.println("ERROR: Graph has cycles");
            return;
            }
         // Вставка метки вершины в массив (начиная с конца)
         sortedArray[nVerts-1] = vertexList[currentVertex].label;

         deleteVertex(currentVertex);  // Удаление вершины
         }

      // Все вершины удалены, вывод содержимого sortedArray
      System.out.print("Topologically sorted order: ");
      for(int j=0; j<orig_nVerts; j++)
         System.out.print( sortedArray[j] );
      System.out.println("");
      }
// ------------------------------------------------------------
   public int noSuccessors()  // Метод возвращает вершину, не имеющую
      {                       // преемников (или -1, если ее нет)
      boolean isEdge;  // Ребро в матрице adjMat, ведущее от row в column

      for(int row=0; row<nVerts; row++)  // Для каждой вершины
         {
         isEdge = false;                 // Проверка ребер
         for(int col=0; col<nVerts; col++)
            {
            if( adjMat[row][col] > 0 )   // Если существует ребро
               {                         // от текущей вершины
               isEdge = true;
               break;                    // У вершины имеется преемник
               }                         // Переход к проверке
            }                            // другой вершины
         if( !isEdge )                   // Если ребер нет,
            return row;                  // то нет и преемников
         }
      return -1;                         // Вершина не найдена
      }
// ------------------------------------------------------------
   public void deleteVertex(int delVert)
      {
      if(delVert != nVerts-1)      // Если вершина не последняя
         {                         // Удаление из vertexList
         for(int j=delVert; j<nVerts-1; j++)
            vertexList[j] = vertexList[j+1];
                                   // удаление строки из adjMat
         for(int row=delVert; row<nVerts-1; row++)
```

продолжение ⇩

Листинг 13.4 (*продолжение*)

```
            moveRowUp(row, nVerts);
                                    // Удаление столбца из adjMat
         for(int col=delVert; col<nVerts-1; col++)
            moveColLeft(col, nVerts-1);
         }
      nVerts--;                     // Количество вершин уменьшается на 1
      }
// -------------------------------------------------------------
   private void moveRowUp(int row, int length)
      {
      for(int col=0; col<length; col++)
         adjMat[row][col] = adjMat[row+1][col];
      }
// -------------------------------------------------------------
   private void moveColLeft(int col, int length)
      {
      for(int row=0; row<length; row++)
         adjMat[row][col] = adjMat[row][col+1];
      }
// -------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class TopoApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addVertex('F');    // 5
      theGraph.addVertex('G');    // 6
      theGraph.addVertex('H');    // 7
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(0, 4);     // AE
      theGraph.addEdge(1, 4);     // BE
      theGraph.addEdge(2, 5);     // CF
      theGraph.addEdge(3, 6);     // DG
      theGraph.addEdge(4, 6);     // EG
      theGraph.addEdge(5, 7);     // FH
      theGraph.addEdge(6, 7);     // GH
      theGraph.topo();            // Сортировка
      }
   }  // Конец класса TopoApp
////////////////////////////////////////////////////////////////
```

В следующей главе будут рассмотрены графы, в которых ребрам присваиваются не только направления, но и веса.

Связность в направленных графах

Вы уже знаете, как в ненаправленном графе выполняется поиск всех соединенных узлов (обход в глубину или в ширину). При поиске всех соединенных вершин в направленном графе ситуация усложняется. Алгоритм уже не может начать со случайно выбранной вершины и рассчитывать на то, что ему удастся достичь всех остальных соединенных с ней вершин.

Для примера рассмотрим граф на рис. 13.15. Если начать с вершины A, то алгоритм сможет добраться до вершины C, но не до других вершин. Начиная с B, он не сможет добраться до вершины D, а от вершины C он вообще никуда не сможет попасть. В том, что касается связности, необходимо найти ответ на один важный вопрос: каких вершин можно достичь, если начать обход с конкретной вершины?

Таблица связности

Программу `dfs.java` (см. листинг 13.1) легко изменить таким образом, чтобы обход последовательно начинался с каждой вершины. Вот как выглядит результат ее выполнения для графа на рис. 13.15:

```
AC
BACE
C
DEC
EC
```

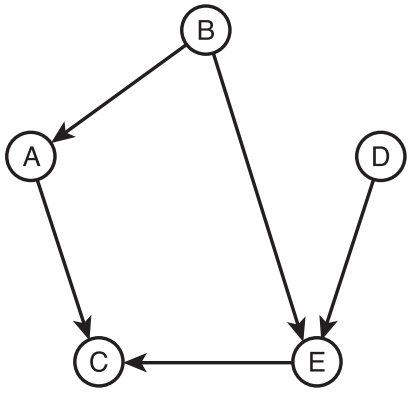


Рис. 13.15. Направленный граф

Мы получили *таблицу связности* для направленного графа. Первая буква в строке определяет начальную вершину, а остальные буквы — вершины, к которым можно перейти (напрямую или через другие вершины) от начальной вершины.

Алгоритм Уоршелла

В некоторых приложениях необходимо быстро определить, возможен ли переход от одной вершины к другой. Допустим, вы хотите перелететь из Афин в Мурманск, причем количество промежуточных пересадок вас не беспокоит. Возможен ли такой перелет?

Можно проанализировать таблицу связности, но это потребует просмотра всех элементов отдельной строки за время $O(N)$ (где N — среднее количество вершин, доступных от заданной вершины). У вас нет лишнего времени; существует ли более быстрый способ?

Оказывается, для графа можно построить таблицу, которая мгновенно (то есть за время $O(1)$) сообщит вам, возможен ли переход от одной вершины к другой. Такая таблица строится систематическим изменением матрицы смежности графа. Граф, представленный такой видоизмененной матрицей смежности, называется *транзитивным замыканием* исходного графа.

Напомним, что в обычной матрице смежности номер строки определяет начало ребра, а номер столбца — его конец. (Таблица связности имеет такую же структуру.) Единица на пересечении строки C и столбца D означает, что в графе существует ребро, ведущее от вершины C к вершине D. От одной вершины к другой можно перейти за один шаг. (Конечно, в направленном графе из этого не следует существование обратного перехода, из D в C.) В таблице 13.6 приведена матрица смежности для графа на рис. 13.15.

Таблица 13.6. Матрица смежности

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

Преобразование матрицы смежности в транзитивное замыкание графа осуществляется по *алгоритму Уоршелла*. Этот алгоритм, выполняющий довольно значительную работу в нескольких строках кода, основан на простой идее:

> Если от вершины L можно перейти к вершине M, а от вершины M — к вершине N, то можно перейти и от вершины L к вершине N.

Двухшаговый путь создается из двух одношаговых путей. Матрица смежности содержит все допустимые одношаговые пути, поэтому она станет хорошей отправной точкой для применения этого правила.

Найдет ли алгоритм пути длиной более двух ребер? В конце концов, в правиле речь идет об объединении двух одношаговых путей в один двухшаговый. Оказывается, алгоритм способен строить пути произвольной длины на основании найденных ранее многошаговых путей. Приведенная ниже реализация гарантирует результат, а его теоретическое обоснование выходит за рамки книги.

Итак, мы рассмотрим работу алгоритма на примере табл. 13.6. Мы последовательно, строку за строкой проанализируем каждую ячейку матрицы смежности.

Строка A

Начнем со строки A. В столбцах A и B стоят нули, но столбец C содержит 1; остановимся на нем.

Единица в этой ячейке означает, что в графе существует путь от A к C. Если бы нам было известно, что также существует путь от другой вершины X к A, то можно было бы утверждать и о существовании пути от X к C.

Какие ребра графа заканчиваются в вершине A (и есть ли такие ребра?)? Их следует искать в столбце A. Из табл. 13.6 видно, что столбец A содержит всего одну единицу в строке B. Она означает, что в графе существует ребро от вершины B к A.

Таким образом, существует ребро от B к A и (исходное) ребро от A к C. Из этого следует, что от B к C можно перейти за два шага. В этом можно убедиться по рис. 13.15. Чтобы сохранить этот результат, мы ставим 1 на пересечении строки B и столбца C (рис. 13.16, *а*).

Остальные ячейки строки A пока остаются пустыми.

Строки B, C и D

Переходим к строке B. Первая ячейка (столбец A) содержит 1, указывая на существование ребра от B к A. Существуют ли другие ребра, заканчивающиеся в B? Столбец B пуст; следовательно, ни одна из единиц в строке B не приведет к обнаружению более длинного пути, потому что ни одно ребро не заканчивается в B.

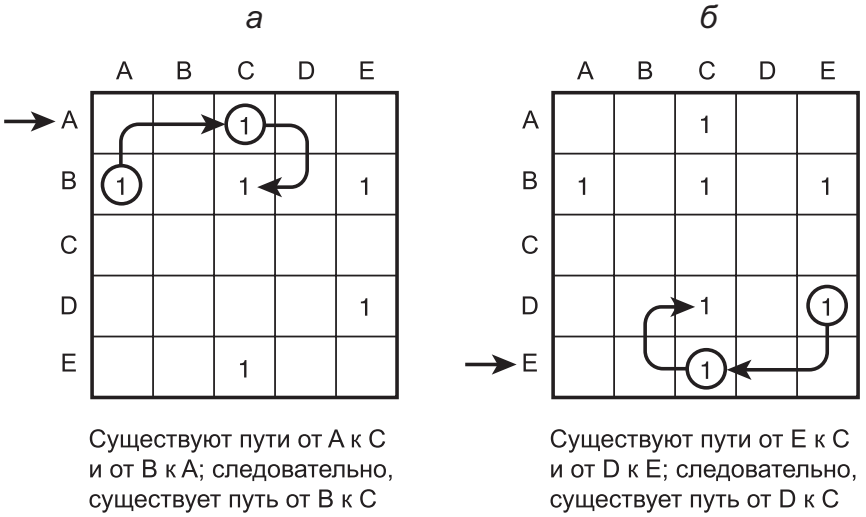


Рис. 13.16. Последовательность действий в алгоритме Уоршелла: *а*) *y* = 0; *б*) *y* = 4

Строка C не содержит ни одной единицы, переходим к строке D. Здесь находится ребро от D к E. Однако столбец D пуст; соответственно ребер, заканчивающихся на вершине D, не существует.

Строка E

В строке E отыскивается ребро от E к C. Просматривая столбец E, мы находим в нем ребро от B к E; из существования путей от B к E и от E к C делается вывод о существовании пути от B к C. Однако этот путь уже был обнаружен ранее, на что указывает 1 в ячейке таблицы. Пути от D к E и от E к C образуют путь от D к C, в эту ячейку ставится 1. Результат показан на рис. 13.16, *б*.

На этом работа алгоритма Уоршелла завершается. В матрицу смежности были добавлены две единицы; измененная матрица показывает, к каким узлам можно перейти от другого узла за любое количество шагов. Если нарисовать граф на основе новой матрицы, получится транзитивное замыкание исходного графа на рис. 13.15.

Реализация алгоритма Уоршелла

Один из способов реализации алгоритма Уоршелла основан на использовании трех вложенных циклов (предложен Седжвиком — см. приложение Б). Внешний цикл перебирает строки (переменная y). Средний цикл перебирает все ячейки строки (переменная x). Если в ячейке (x,y) обнаруживается 1, значит, в графе существует ребро от y к x; активизируется третий (внутренний) цикл с переменной z.

Третий цикл просматривает ячейки в столбце y и ищет ребро, завершающееся в y. (Обратите внимание: в первом цикле переменная y используется для перебора строк, а в третьем цикле она индексирует столбцы.) Если элемент на пересечении столбца y со строкой z содержит 1, значит, существует ребро от z к y. Из факта существования двух ребер — от z к y и от y к x — делается вывод о существовании пути от z к x, вследствие чего 1 размещается в ячейке (x, z). Подробности реализации остаются читателю для самостоятельной работы.

Итоги

- Графы состоят из вершин, соединенных ребрами.
- Графы могут представлять многие реально существующие сущности: пути авиаперелетов, электрические цепи, планируемые задачи и т. д.
- Алгоритмы обхода решают задачу систематического посещения каждой вершины графа. Обход лежит в основе ряда других операций.
- Два основных алгоритма обхода — обход в глубину и обход в ширину.
- Алгоритм обхода в глубину обычно реализуется на базе стека, а алгоритм обхода в ширину — на базе очереди.
- Минимальное остовное дерево состоит из минимального количества ребер, необходимых для соединения всех вершин графа.
- Для построения минимального остовного дерева используется незначительная модификация алгоритма обхода в глубину для невзвешенного графа.
- В направленном графе ребра обладают направлением (часто обозначаемым стрелкой).
- Алгоритм топологической сортировки создает список вершин, упорядоченных таким образом, что при наличии пути от A к B вершина A предшествует B в списке.
- Топологическая сортировка может выполняться только с направленными ациклическими графами.
- Топологическая сортировка обычно используется при планировании сложных проектов, состоящих из задач, выполнение которых возможно только после завершения других задач.
- Алгоритм Уоршелла определяет возможность перехода (прямого или многошагового) от произвольной вершины к любой другой вершине.

Вопросы

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

1. В графе _____ соединяет две _____ .
2. Как по матрице смежности определить количество ребер в ненаправленном графе?
3. Какая структурная единица графа соответствует выбору хода в игровом моделировании?
4. Направленным называется граф, в котором:
   a) ребра образуют минимальное остовное дерево;
   b) из вершины A возможен переход только в вершину B, затем в вершину C и т. д.;
   c) от одной вершины к другой можно перейти только в одном направлении;
   d) по любому заданному пути можно перемещаться только в одном направлении.
5. Матрица смежности состоит из строк {0,1,0,0}, {1,0,1,1}, {0,1,0,0} и {0,1,0,0}. Как выглядит соответствующий список смежности?
6. Минимальным остовным деревом называется граф, в котором:
   a) количество ребер, соединяющих все вершины, минимально;
   b) количество ребер равно количеству вершин;
   c) все избыточные вершины были удалены;
   d) каждая пара вершин соединяется минимальным количеством ребер.
7. Сколько разных минимальных основных деревьев существует в ненаправленном графе из трех вершин и трех ребер?
8. Ненаправленный граф обязательно содержит цикл, если:
   a) к любой вершине можно перейти от другой вершины;
   b) количество путей больше количества вершин;
   c) количество ребер равно количеству вершин;
   d) количество путей меньше количества ребер.
9. Граф, не содержащий циклов, называется _____.
10. Может ли минимальное остовное дерево ненаправленного графа содержать циклы?
11. Для заданного графа может существовать несколько правильных вариантов топологической сортировки (Да/Нет).
12. Результатом топологической сортировки является:
    a) упорядочение вершин, при котором все направленные ребра идут в одном направлении;
    b) упорядочение вершин в порядке возрастания количества ребер;

c) упорядочение вершин, при котором A предшествует B, B предшествует C и т. д.;

d) упорядочение вершин, при котором вершины, от которых идут ребра к другим вершинам, предшествуют им в списке.

13. Может ли дерево содержать циклы?

14. По какому критерию программа `topo.java` (см. листинг 13.4) делает вывод о наличии в графе цикла?

Упражнения

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

1. Создайте в приложении GraphN Workshop граф с пятью вершинами и семью ребрами. Запишите матрицу смежности графа, не используя кнопку `View`. Потом нажмите кнопку `View` и проверьте правильность своей матрицы.

2. Классическая игра «крестики-нолики» играется на доске 3 × 3; для простоты мы рассмотрим игру на доске 2 × 2, в которой игроку для победы достаточно выстроить в ряд два крестика или нолика. Используйте приложение GraphN для построения графа, моделирующего игру на доске 2 × 2. Необходима ли этому графу глубина 4?

3. Создайте матрицу смежности из пяти вершин, заполните ее случайными 0 и 1. Не обращайте внимания на симметрию. Теперь без использования кнопки `View` создайте соответствующий направленный граф в приложении GraphD Workshop. Затем нажмите кнопку `View` и проверьте, соответствует ли граф матрице смежности.

4. Попробуйте создать в приложении GraphD Workshop граф с циклом, который не будет распознан методом `topo()`.

Программные проекты

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

13.1. Измените программу `bfs.java` (см. листинг 13.2) так, чтобы для поиска минимального остовного дерева применялся алгоритм обхода в ширину вместо обхода в глубину из программы `mst.java` (см. листинг 13.3). Создайте в методе `main()` граф с 9 вершинами и 12 ребрами, постройте его минимальное остовное дерево.

13.2. Измените программу `dfs.java` (см. листинг 13.1), чтобы вместо матрицы смежности в ней использовались списки смежности. Для работы со списками

можно воспользоваться классами `Link` и `LinkList` из программы `linkList2.java` (см. листинг 5.2) главы 5. Измените метод `find()` класса `LinkList`, чтобы он искал непосещенную вершину вместо значения ключа.

13.3. Измените программу `dfs.java` (см. листинг 13.1), чтобы она выводила таблицу связности для направленного графа (см. раздел «Связность в направленных графах»).

13.4. Реализуйте алгоритм Уоршелла для построения транзитивного замыкания графа. Начните с кода программного проекта 13.3. Полезно предусмотреть возможность вывода матрицы смежности на разных стадиях работы алгоритма.

13.5. «Ход коня» — древняя и хорошо известная шахматная головоломка. Шахматный конь делает ходы на пустой шахматной доске, пока каждое поле не будет посещено ровно один раз. Напишите программу, которая решает эту головоломку методом обхода в глубину. Размер доски желательно сделать переменным, чтобы решение можно было опробовать на доске меньшего размера. Для стандартной доски 8×8 расчет решения на настольном компьютере может занять несколько лет, но для доски 5×5 потребуется около минуты. За дополнительной информацией обращайтесь к разделу «Обход в глубину в программировании игр». Возможно, при совершении хода проще создавать нового коня, остающегося на новой клетке. При таком подходе конь будет соответствовать вершине, а последовательность создания коней может храниться в стеке. Игрок побеждает, когда вся доска будет заполнена конями (то есть при заполнении стека). Традиционно в этой задаче поля доски нумеруются последовательно, от 1 в левом верхнем углу до 64 в правом нижнем углу (или от 1 до 25 на доске 5×5). При поиске следующего хода конь должен не только сделать ход по правилам, но и не выйти за пределы доски или попасть на уже занятое (посещенное ранее) поле. Если после каждого хода программа будет выводить состояние доски и ожидать нажатия клавиши, вы сможете понаблюдать за работой алгоритма: как на доске появляется все больше коней, а в тупиковой ситуации программа снимает часть коней и опробует другую последовательность ходов. Сложность этой задачи более подробно анализируется в следующей главе.