Модуль 4. Алгоритмы и структуры данных

Тема* 4.16. Графы

Оглавление

4.16. Графы

4.16.1. Общие понятия графов

Что же собой представляют графы? Вообще говоря, это понятие дается в математике, но используется повсеместно. Граф представляет собой определенное количество **вершин** (или **узлов**), некоторые из которых (а может быть, и все) соединены между собой так называемыми **ребрами** (или **дугами**). Вершины чаще всего обозначают (именуют) числами или латинскими буквами. Ребра могут иметь на одном из концов стрелку, тогда говорят о том, что мы имеем дело с **ориентированным** графом (см. рисунок 4.16.1б). Если же все имеющиеся ребра определены без стрелок, то такой граф называют **неориентированным** (см. рисунок 4.16.1а). В математике существует целая область, изучающая графы. Однако мы коснемся графов только в некоторой степени, так как полноценное их изучение потребует не одной книги.

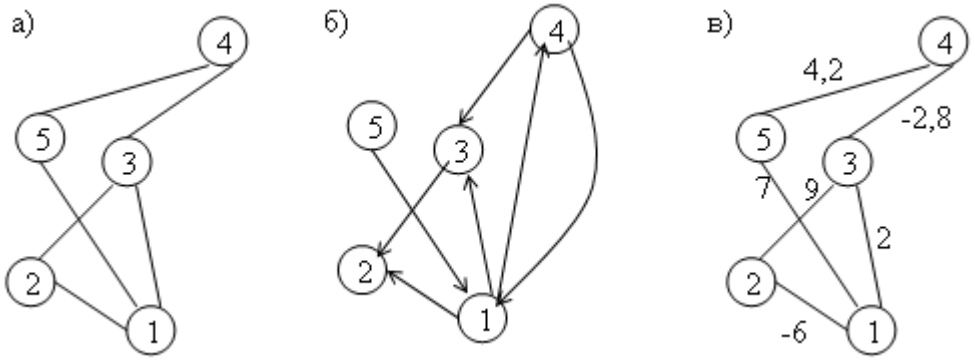


Рисунок 4.16.1 – Примеры графов: а) неориентированный граф; б) ориентированный граф; в) взвешенный граф.

Применение графов достаточно обширно. Мы, не задумываясь, оперируем терминами теории графов даже в обычной жизни. Например, популярные в настоящий момент социальные сети могут быть описаны в виде графа, в котором вершинами являются люди, зарегистрированные в них, а ребра показывают, кто с кем состоит в дружественных отношениях. Причем отношение дружбы разумно понимать, как ребро без стрелки и такой граф будет неориентированным. В то же время в некоторой социальной сети можно становиться так называемым «подписчиком» какого-то человека (или даже организации), что приводит нас к ориентированным графам, так как совсем не обязательно, чтобы организация, на новости которой мы подписались, тоже решила стать нашим подписчиком.

Многие из нас пользуются достижениями в области теории графов, когда речь заходит о поиске маршрута между двумя населенными пунктами или схемы проезда от одного места города до другого, особенно, если этот город достаточно велик. Такие сервисы предоставляются и крупными порталами, и специальными автомобильными навигаторами. Для решения подобных задач программистам было необходимо представить город или страну как множество вершин, между которыми имеется огромное количество ребер, представляющих наши транспортные возможности. Такие графы могут содержать до десятков тысяч вершин, и для решения многих

других практических задач может потребоваться серьезный объем разработанных в этой сфере алгоритмов.

Приведенный пример важен также и тем, что мы сталкиваемся со случаем, когда, во-первых, некоторые пары вершин соединены несколькими ребрами (можно добраться от одной остановки до другой на метро, на троллейбусе или автобусе или вообще пешком), а, во-вторых, такие ребра имеют так называемый вес. **Вес ребра** – это некоторое число (или даже текст), которое связывают с этим ребром. При наличии у ребер графа весов такой граф называется **взвешенным**, или **нагруженным** (см. рисунок 4.16.1в), иначе – **невзвешенным**. Это число может быть как положительным, так и отрицательным (и нулем, конечно, тоже). Чаще всего под весом ребра понимается некоторая стоимость, время или расстояние. Например, пеший переход от остановки А до остановки Б вам обойдется в 15 минут, а проезд на троллейбусе всего – 3 минуты. С другой стороны, можно определить вес ребра как расстояние между двумя пунктами или как цену билета. Когда вес ребра – расстояние, то ребро, характеризующее движение пешком, и ребро, характеризующее движение на транспорте, часто различаются по весу, ведь пешком мы вполне можем «срезать дорогу». Когда вес ребра представляет собой стоимость билета, то мы получаем интересный эффект: ребра, проведенные между вершинами, соответствующими остановкам конкретного маршрута автобуса, имеют одинаковый вес (то есть, цену). Проезд от остановки А до остановки Б стоит, например, 40 рублей, от Б до В – тоже 40 рублей, и от А до В – тоже 40 рублей.

4.16.2. Представление графа в памяти компьютера

Существует большое количество уже ставших классическими алгоритмов на графах, но прежде, чем мы коснемся некоторых из них, нужно поговорить о том, каким образом информацию о графе можно хранить в памяти компьютера. Основных способов два: представление графа в виде **матрицы смежности** и в виде **списков смежности**.

Две вершины называются **смежными**, если имеется соединяющее их ребро. В этом случае говорят, что эти вершины **инцидентны** этому ребру. Количество вершин, смежных с данной, определяет так называемую **степень** вершины для случая неориентированных графов.

Матрица смежности представляет собой в случае хранения в памяти компьютера двумерный массив с одинаковым количеством строк и столбцов. Первая строка соответствует описанию ребер, выходящих из первой вершины (в этом случае может учитываться наличие у ребра направления), вторая строка – для второй вершины и т.д. Что же хранится в элементе нашего двумерного массива? Если имеющийся граф – невзвешенный, то достаточно хранить информацию о том, есть ли ребро или нет. Очевидно, элементу матрицы в этом случае достаточно хранить логическое значение. Наличию ребра соответствует истинное значение, отсутствию – ложное. Если же ребра графа имеют вес, то необходимо это число и сохранить, а, следовательно, тип элемента матрицы соответствует типу значения веса. Если вес ребра суть целое число, то и тип элемента – целый, если вещественное – тогда вещественный. Как уже было сказано, значение веса ребра может быть как положительным, так и отрицательным. Для обозначения отсутствия ребра в зависимости от задачи бывает удобно либо использовать нулевое значение, либо – достаточно большое число, обозначающее «бесконечный» вес ребра.

В качестве иллюстрации представим в виде таблицы 4.16.1 матрицу смежности для графа, изображенного на рисунке 4.16.1а:

Таблица 4.16.1 – Матрица смежности графа на рисунке 4.16.1а

0	1	1	0	1
1	0	1	0	0
1	1	0	1	0
0	0	1	0	1
1	0	0	1	0

*Для неориентированного графа матрица смежности имеет **симметричный** вид. Это следует из того, что наличие ребра между какой-либо парой вершин в таком графе позволяет «перемещаться» по этому ребру в обоих направлениях.*

Представление графа в виде списков смежности требует от программиста чуть больше навыков. Можно использовать, конечно, и динамические линейные связные списки, а можно – динамические массивы, что в языке Java достаточно просто. Если граф – невзвешенный, то для каждой вершины нам нужно лишь хранить в виде списка те вершины, куда ведут ребра, выходящие из выбранной. Все несколько сложнее, если ребра графа имеют вес. Тогда нужно хранить пары чисел – вершину, куда ведет ребро, и сам вес этого ребра. Например, у графа на рисунке 4.16.1а список смежности для вершины 1 содержит номера {2, 3, 5}, для вершины 2 – только номер {1}, а для вершины 5 – номера {1, 4}.

Существуют и более сложные способы представления графов, но нам вполне достаточно и этих двух. Каждый из них имеет свои преимущества и недостатки, что приводит нас к аккуратному их выбору для реализуемого алгоритма на основе эффективности их применения в конкретной задаче. Эффективность каждого из представлений всегда рассматривается с двух позиций: объем необходимой памяти и скорость работы алгоритма.

Начнем описывать класс GraphMatrix, в котором реализуем основные алгоритмы работы с графами. Предполагается, что класс будет хранить информацию о графе в виде матрицы смежности. Методы создаваемого класса должны реализовывать алгоритмы работы с графами. Перечислим поля нашего класса:

```
public class GraphMatrix {
        private int vertices, // количество вершин графа
                infinity; // "бесконечный" вес ребра
        private int[][] matrix; // матрица смежности
}
```

Конструктор класса должен выделить память под матрицу смежности, проинициализировав каждое ребро значением «бесконечного» веса (в качестве этого значения можно брать либо 0, либо очень большое число). Оговоримся, что ввиду нумерации вершин графа от 1, для удобства

мы будем выделять в матрице дополнительную память, чтобы можно было обращаться по индексам от 1 до количества вершин **vertices**.

```
// Конструктор создания объекта с v вершинами и "бесконечным" весом inf
public GraphMatrix(int v, final int inf) {
        infinity = inf; vertices = v;
        matrix = new int[v + 1][v + 1];
        for (int i = 1; i <= vertices; i++) {
        for (int j = 1; j <= vertices; j++)
                        matrix[i][j] = infinity; // ребро бесконечного веса
        }
        for (int i = 1; i <= vertices; i++)
        matrix[i][i] = 0; // главная диагональ с нулями
}
```

Напишем функцию для добавления нового ребра в граф. Этот метод универсален в том смысле, что позволяет нам сформировать ориентированный граф. В дальнейшем для удобства понадобится создавать метод для добавления неориентированного ребра.

```
// Добавление ребра веса weight от вершины v1 к v2
public void addEdge(int v1, int v2, int weight) {
      matrix[v1][v2] = weight;
}
```

Чтобы можно было определить, имеется ли в графе ребро между определенной парой вершин, а также, каков вес ребра, которое их соединяет, реализуем соответствующие методы:

```
// Определяет наличие ребра между вершинами v1 и v2
public boolean hasEdge(int v1, int v2) {
        if (v1 == v2) // считается, что вершина не соединена сама с собой
        return false;
        return (matrix[v1][v2] != infinity);
}
// узнать вес ребра между вершинами v1 и v2
public int edgeWeight(int v1, int v2){
        return matrix[v1][v2];
}
```

Так как поле vertices имеет модификатор private, сформируем короткий метод, позволяющий определить, сколько в графе вершин:

```
public int getVertices(){ return vertices; }
```

Создадим теперь новый Java-проект (**File⇒New ⇒Java Project**) по имени ProgramWithGraph с одноименным классом (**File⇒ New ⇒ Class**), представляющим собой основную программу со статическим методом main. Для работы с графом нам необходимо добавить (**File⇒New ⇒Class**) описание класса GraphMatrix в проект в виде отдельного одноименного файла.

Упражнение 4.16.1

Построить граф, изображенный на рисунке 4.16.1а, создав объект класса GraphMatrix и добавив необходимые ребра. Вывести матрицу смежности построенного графа.

Для решения поставленной задачи объявим в методе main объект класса GraphMatrix, указав количество вершин, равное пяти, и «бесконечный» вес ребра 0. Добавление дуг между вершинами будем осуществлять, вызывая метод addEdge, указывая единичный вес ребра. Промежуточный вариант класса ProgramWithGraph будет иметь вид:

```
import java.io.PrintStream;
public class ProgramWithGraph {
        public static PrintStream out = System.out;
        public static void main(String[] args) {
        GraphMatrix g = new GraphMatrix(5, 0);
        g.addEdge(1, 2, 1); g.addEdge(2, 1, 1);
        g.addEdge(1, 3, 1); g.addEdge(3, 1, 1);
        g.addEdge(1, 5, 1); g.addEdge(5, 1, 1);
                g.addEdge(2, 3, 1); g.addEdge(3, 2, 1);
        g.addEdge(3, 4, 1); g.addEdge(4, 3, 1);
        g.addEdge(4, 5, 1); g.addEdge(5, 4, 1);
        }
}
```

Обратим внимание - метод addEdge вызывается дважды для добавления одного неориентированного ребра. Остается после формирования графа вставить код для вывода матрицы смежности графа:

```
out.println("Матрица смежности графа:");
int v = g.getVertices();
for (int i = 1; i <= v; i++) {
        for (int j = 1; j <= v; j++)
        out.print(g.edgeWeight(i, j) + " ");
        out.println();
}
```

Упражнение 4.16.2

Добавьте в описание класса GraphMatrix метод addEdgeNotOriented для добавления неориентированного ребра. Замените в функции main все парные вызовы метода addEdge на однократный вызов вновь созданного метода.

Решение выглядит так:

```
// добавление неориентированного ребра
public void addEdgeNotOriented(int v1, int v2, int weight) {
        addEdge(v1, v2, weight); addEdge(v2, v1, weight);
}
```

Задание 4.16.1

Перенесите реализацию вывода матрицы смежности графа в новый метод класса GraphMatrix. В качестве параметра этот метод должен принимать объект класса PrintStream.

4.16.3. Обходы графа

При работе с графами часто возникает необходимость некоторого систематического посещения вершин графа для их обработки в какой-либо последовательности. Под обработкой здесь понимается некоторое действие с вершиной, например, вывод информации о ее метке. Такой процесс называется обходом. Существует два базовых алгоритма обхода графа: **обход в ширину** и **обход в глубину**. Идеи каждого из этих алгоритмов в дальнейшем используются для построения более сложных алгоритмов обработки графов.

Оба обхода имеют один общий принцип – перемещение между смежными вершинами графа происходит по соединяющим их ребрам. Если ребро имеет направление, то двигаться можно только в одном направлении, если граф неориентированный, то в обоих. Однако оба обхода мы рассмотрим для случая неориентированных и невзвешенных графов. Для организации посещения вершин в определенном порядке при обходе в глубину используется специальная структура данных стек, а при обходе в ширину – очередь.

4.16.3.1. Обход графа в глубину

При посещении вершин во время обхода в глубину идея состоит в том, что мы движемся из текущей вершины к не посещенной смежной вершине. Если из текущей вершины нет хода ни в одну смежную не посещенную вершину, то мы возвращаемся в ту вершину, откуда пришли в текущую. Информацию об этом мы храним в специальной структуре данных – стеке.

Пусть нам нужно при посещении вершины выводить ее номер. В общем случае алгоритм посещения вершин при обходе в глубину можно записать следующим образом.

1. Пометим все вершины нашего графа как не посещенные. Пусть мы решили обходить граф из вершины с номером K. Инициализируем пустой стек. Поместим в него число K (мы в этой вершине). Отметим вершину K как посещенную. Выведем ее номер в ответ.

2. Если у текущей вершины нет смежных не посещенных вершин, перейдем к пункту “3”. Если есть – выберем любую (например, с минимальным номером). Пусть это вершина A. Перейдем в нее, отметим ее как посещенную, поместим ее номер в стек. Выведем ее номер в ответ. Повторим правило 2.
3. При отсутствии смежных непосещенных вершин извлекаем из стека текущую вершину. Если после этого стек не пуст, переходим в вершину, номер которой находится на вершине стека. При этом переходим к пункту 2 алгоритма. Если стек пуст, то обход в глубину окончен.

Рассмотрим для примера следующий граф, изображенный на рисунке 4.16.2а. Сплошными линиями показаны ребра исходного графа. Пунктирными стрелками показаны переходы между вершинами. В качестве начала обхода выбрана вершина номер 5. Следуя алгоритму, мы посетим вершины 6, 4, 3, затем возвратимся обратно к вершине 5, зайдем в вершину 7, вернемся в 5, перейдем в 8, 1, 2 (см. рисунок 4.16.2б).

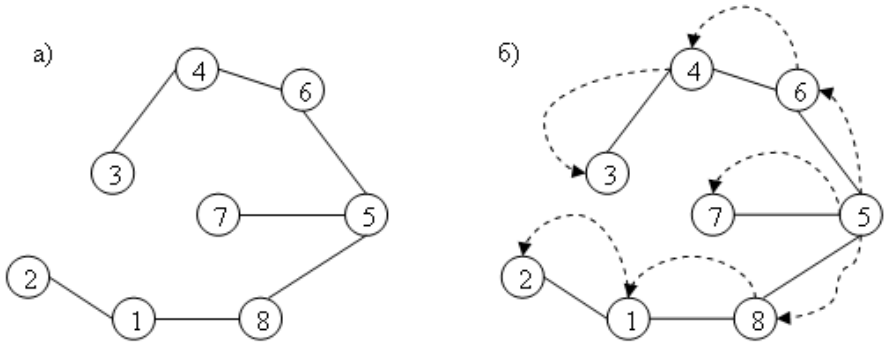


Рисунок 4.16.2 – а) невзвешенный неориентированный граф из 8 вершин; б) он же с переходами при обходе в глубину.

Представим подробно процесс обхода вершин рассматриваемого графа.

0. Помечаем все вершины графа как непосещенные, инициализируем пустой стек.
1. Переходим в стартовую вершину 5, помечаем ее как посещенную, добавляем ее в стек, выводим номер в ответ.
2. У вершины 5 непосещенные смежные вершины: 6, 7, 9. Выбираем вершину, например, 6. Помечаем ее как посещенную, помещаем в стек, переходим в нее, выводим номер 6 в ответ.
3. У вершины 6 только одна непосещенная смежная вершина 4 (вершина 5 была посещена ранее). Помечаем вершину 4 как посещенную, добавляем в стек, переходим в нее, выводим номер 4 в ответ.
4. У вершины 4 только одна непосещенная смежная вершина 3. Помечаем вершину 3 как посещенную, добавляем в стек, переходим в нее, выводим номер 3 в ответ.
5. У вершины 3 нет непосещенных смежных вершин, поэтому применяем пункт 3 из указанного выше алгоритма, а именно: извлекаем из стека текущее значение (это 3), смотрим на вершину стека (это 4), возвращаемся туда.
6. У вершины 4 нет непосещенных смежных вершин, поэтому извлекаем из стека текущее значение (это 4), смотрим на вершину стека (это 6), возвращаемся туда.

7. У вершины 6 нет непосещенных смежных вершин, поэтому извлекаем из стека текущее значение (это 6), смотрим на вершину стека (это 5), возвращаемся туда.
8. У вершины 5 непосещенными смежными вершинами остались 7 и 8. Выбираем вершину, например, 7. Помечаем ее как посещенную, помещаем ее номер в стек, переходим в нее, выводим номер 7 в ответ.
9. У вершины 7 нет непосещенных смежных вершин (из единственной соседней вершины 5 мы только что пришли), поэтому удаляем из стека текущую вершину (это 7), смотрим на значение в стеке (это 5), переходим туда.
10. Аналогично мы пойдем из вершины 5 в вершины 8, 1 и 2, выводя их значения в ответ.

Для удобства мы отобразим в таблице 4.16.2 состояние стека в процессе обхода в глубину.

Таблица 4.16.2 – Содержимое стека при обходе графа в глубину

Действие	Содержимое стека (от дна стека к его вершине)
Посещение 5	5
Посещение 6	5, 6
Посещение 4	5, 6, 4
Посещение 3	5, 6, 4, 3
Возврат к 4	5, 6, 4
Возврат к 6	5, 6
Возврат к 5	5
Посещение 7	5, 7
Возврат к 5	5
Посещение 8	5, 8
Посещение 1	5, 8, 1
Посещение 2	5, 8, 1, 2
Возврат к 1	5, 8, 1
Возврат к 8	5, 8
Возврат к 5	5
Выход	пусто

Продолжим разработку нашего класса GraphMatrix. На этот раз мы реализуем обход в глубину. Однако для универсальности, наш метод вместо вывода номера вершины на экран просто вернет в качестве результата список вершин в порядке их обхода. С учетом представленного выше алгоритма, нам придется воспользоваться готовыми структурами данных: Stack – для организации стека, ArrayList – для хранения списка вершин в порядке обхода. Тогда новый метод класса GraphMatrix примет следующий вид:

```
public ArrayList<Integer> depthFirstSearch(int startVertex) {
        // пометить все вершины непосещенными
        boolean[] visited = new boolean[vertices+1];
```

```
        for (int i = 1; i <= vertices; i++)
        visited[i] = false;
        Stack<Integer> stack = new Stack<>();
        ArrayList<Integer> outList = new ArrayList<>();
        // посетить стартовую вершину
        visited[startVertex] = true;
        stack.add(startVertex);
        outList.add(startVertex);
        // обход остальных вершин
        while (!stack.isEmpty()) {
        int currentVertex = stack.peek();
        // найдем любую смежную непосещенную вершину
        int adjVertex = 1;
        for (; adjVertex <= vertices; adjVertex++)
                        if (hasEdge(currentVertex, adjVertex) &&
!visited[adjVertex])
                                break;
        if (adjVertex > vertices) // нет непосещенных смежных вершин
                        stack.pop();
        else { // надо посетить adjVertex
                        visited[adjVertex] = true;
                        stack.add(adjVertex);
                        outList.add(adjVertex);
        }
        }
        return outList;
}
```

Если говорить о том, насколько важен алгоритм обхода в глубину, то необходимо указать некоторые из алгоритмов, которые либо используют его, как некий элемент, либо строятся исходя из общей идеи этого алгоритма. Обход в глубину позволяет находить компоненты связности графа, строить его топологическую сортировку, определять наличие цикла в графе, определять так называемые мосты и точки сочленения. В качестве идеи обход в глубину используется, например, при построении эйлерового цикла в графе.

Упражнение 4.16.3

Добавьте в описание класса представленный метод обхода в глубину. Измените полученную в упражнении 4.16.2 программу следующим образом: в главном методе main нужно сформировать граф, изображенный на рисунке 4.16.2а, запросить у пользователя стартовую вершину обхода в глубину и вывести на экран список вершин графа в порядке обхода. Введите при запуске программы в качестве стартовой вершины число 5. Убедитесь, что ваша программа выводит на экран следующий список вершин: 5, 6, 4, 3, 7, 8, 1, 2. Попробуйте посмотреть на результат работы программы при других значениях стартовой вершины.

Содержимое главного класса программы будет таким:

```
public static PrintStream out = System.out;
public static Scanner in = new Scanner(System.in);
public static void main(String[] args) {
        GraphMatrix g = new GraphMatrix(8, 0);
        g.addEdgeNotOriented(5, 6, 1); g.addEdgeNotOriented(5, 7, 1);
        g.addEdgeNotOriented(5, 8, 1); g.addEdgeNotOriented(4, 6, 1);
        g.addEdgeNotOriented(1, 8, 1); g.addEdgeNotOriented(1, 2, 1);
        g.addEdgeNotOriented(3, 4, 1);
        out.println("Введите стартовую вершину");
        int startVertex = in.nextInt();
        ArrayList<Integer> traverseDFS = g.depthFirstSearch(startVertex);
        for (Integer vertex : traverseDFS) {
        out.print(vertex + " ");
        }
}
```

4.16.3.2. Обход графа в ширину

Принцип, используемый для посещения вершин графа при обходе в ширину, состоит в том, что в первую очередь нам необходимо попасть во все непосещенные вершины, смежные с текущей. Затем наступает этап посещения вершин, смежных с указанными. То есть пока мы не посетим всех соседей исходной вершины, мы не приступим к обходу их соседей. Для организации такого обхода требуется другая структура данных – очередь.

Пусть так же, как и для случая обхода в глубину, нам нужно при посещении вершины выводить ее номер. Тогда в общем случае алгоритм посещения вершин при обходе в ширину можно записать следующим образом.

1. Пометим все вершины нашего графа как не посещенные. Пусть мы решили обходить граф из вершины с номером K. Инициализируем пустую очередь. Поместим в нее число K (мы в этой вершине). Отметим вершину K как посещенную. Выведем ее номер в ответ.
2. Если очередь не пуста, извлекаем очередную вершину из очереди, иначе переходим к пункту 3. Затем для каждой непосещенной вершины, смежной с данной, выполняем следующие действия: помечаем вершину как посещенную, выводим ее номер в ответ, помещаем эту вершину в очередь. Повторяем пункт 2.
3. Если очередь пуста, обход в ширину окончен.

Рассмотрим поведение алгоритма на примере графа, изображенного на рисунке 4.16.3а.

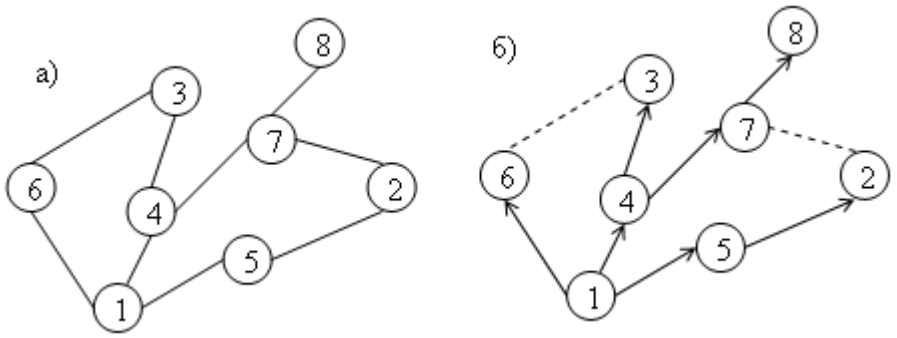


Рисунок 4.16.3 – а) невзвешенный неориентированный граф из 8 вершин; б) остовное дерево этого графа после обхода в ширину.

Представим подробно по шагам процесс обхода вершин рассматриваемого графа.

0. Помечаем все вершины графа как непосещенные, инициализируем пустую очередь.
1. Переходим в стартовую вершину 1, помечаем ее как посещенную, добавляем ее в очередь, выводим номер в ответ.
2. Очередь не пуста, извлекаем из нее вершину, это вершина 1. У этой вершины есть непосещенные смежные вершины под номерами 4, 5, 6. Для каждой такой вершины выполняем действия из правила 2 алгоритма обхода в ширину: помечаем как посещенные, выводим в ответ, помещаем в очередь.
3. Очередь не пуста, извлекаем очередную вершину – это вершина 4. Аналогично для всех непосещенных смежных с ней вершин (вершины графа 3 и 7): помечаем как посещенные, выводим в ответ, помещаем в очередь.
4. Очередь не пуста, извлекаем вершину 5. Она имеет одну непосещенную смежную вершину – это вершина 2. Помечаем ее как посещенную, выводим в ответ, помещаем в очередь.
5. Извлекаем из непустой очереди вершину 6. Все смежные с ней вершины уже посещены, поэтому ничего в очередь не добавляем.
6. Из непустой очереди извлекаем вершину 3. У нее нет непосещенных смежных вершин, на этом шаг заканчивается.
7. Очередь не пуста, поэтому извлекаем вершину 7. Единственной смежной с ней непосещенной вершиной является вершина 8. Помечаем ее как посещенную, выводим ее номер в ответ, помещаем в очередь.
8. Опять извлекаем из очереди вершину 2, но так как у нее нет смежных непосещенных вершин, выполняем следующий шаг.
9. Извлекаем вершину 8 из очереди, но смежных с ней вершин, до сих пор не посещенных нет.
10. Очередь пуста, алгоритм останавливается.

Различия в схеме построения алгоритмов обходов графа в глубину и в ширину проявляются в том, в каком порядке вершины графа обрабатываются в процессе обхода. Можно заметить, что обход в глубину пытается уйти от исходной вершины как можно дальше (с точки зрения количества пройденных ребер), а обход в ширину не отходит от стартовой вершины на большее расстояние, пока не посетит все достижимые вершины на минимальном расстоянии.

Упражнение 4.16.4

Определите кратчайшие расстояния от стартовой вершины 1 до остальных вершин графа на рисунке 4.16.3а.

Ответ: (1: 0, 2: 2, 3: 2, 4: 1, 5: 1, 6: 1, 7: 2, 8: 3).

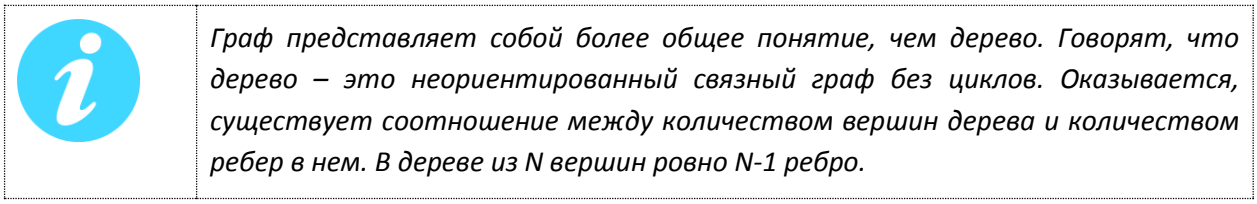
4.16.3.3. Дерево как граф без циклов

Графы представляют собой достаточно общее понятие. Эта схема (структура) для хранения данных обобщает рассмотренную ранее структуру данных, называемую деревом. Чтобы понять связь между ними, необходимо рассмотреть некоторые определения: пути, циклы, связность и другие.

Путь представляет собой такую последовательность вершин графа u_1, u_2, …, u_K, что в графе обязательно имеются дуги вида $u_1 \rightarrow u_2$, $u_2 \rightarrow u_3$, …, $u_{k-1} \rightarrow u_K$. В невзвешенном графе такой путь имеет длину, равную (K-1). Если все вершины, за исключением может быть первой и последней, различны, то такой путь называется простым. Простой путь, первая и последняя вершины в котором совпадают, называется циклом.

Например, в графе на рисунке 4.16.3а: (1→4→7→8) – простой путь, (1→6→3→4→1) и (7→2→5→1→6→3→4→7) – циклы.

Рассмотрим вопросы связности на примере неориентированных графов, что будет несколько проще, чем изучать эту тему для ориентированных. Компонентой связности графа называется максимальное множество вершин графа, в котором существуют пути между любой парой вершин u и v этого множества. Если у графа одна компонента связности, такой граф называется связным.



Граф представляет собой более общее понятие, чем дерево. Говорят, что дерево – это неориентированный связный граф без циклов. Оказывается, существует соотношение между количеством вершин дерева и количеством ребер в нем. В дереве из N вершин ровно N-1 ребро.

Зададимся вопросом, каким образом информацию о структуре дерева можно сохранить в компьютере. Конечно, можно воспользоваться тем, что дерево есть частный случай графа, а, следовательно, можно хранить дерево в виде либо матрицы смежности, либо списков смежности. Но можно воспользоваться более простой структурой для хранения информации о дереве. Если в нем ребра не имеют весов, то тогда достаточно применить следующую идею.

Деревья часто подразделяют на свободные и корневые. В свободных деревьях ни одной вершине не отдается предпочтения перед другими. В корневых же деревьях некоторая вершина выбирается так называемым корнем и дерево как бы «подвешивается» за эту вершину. Все вершины такого дерева располагаются ниже корня на разных уровнях. Если определять уровень корня как 0, то вершины, смежные с корнем, имеют уровень 1, вершины смежные с уровнем 1, имеют уровень 2 и так далее. Можно сказать, что дерево «растет» вниз.

Структуру корневого дерева достаточно легко описать, так как каждая вершина имеет ровно одного родителя – смежную с ней вершину из предыдущего уровня. Если для каждой вершины хранить такую информацию, то мы получим полное представление дерева. Конечно, определяя разный корень в свободном дереве, мы формально получаем разные корневые деревья, однако по сути это будет один и тот же граф-дерево.

В качестве примера рассмотрим граф на рисунке 4.16.2а. Это свободное дерево. Посмотрим, что получится, если определить в нем корнями сначала вершину 5, а затем вершину 8. Полученные корневые деревья изображены на рисунке 4.16.4. В дереве с корнем в вершине 5 уровень 1 имеют вершины 6, 7 и 8, уровень 2 – вершины 4 и 1, уровень 3 – вершины 2 и 3. В дереве с корнем в вершине 8 уровень 1 имеют вершины 1 и 5, уровень 2 – вершины 2, 6 и 7, уровень 3 – вершина 4,

уровень 4 – вершина 3. Исходя из предложенного способа хранения дерева в первом случае мы получим следующий массив предков в виде восьми чисел: (8, 1, 4, 6, 0, 5, 5, 5), а во втором случае – (8, 1, 4, 6, 8, 5, 5, 0). Ноль означает, что данная вершина является корнем дерева.

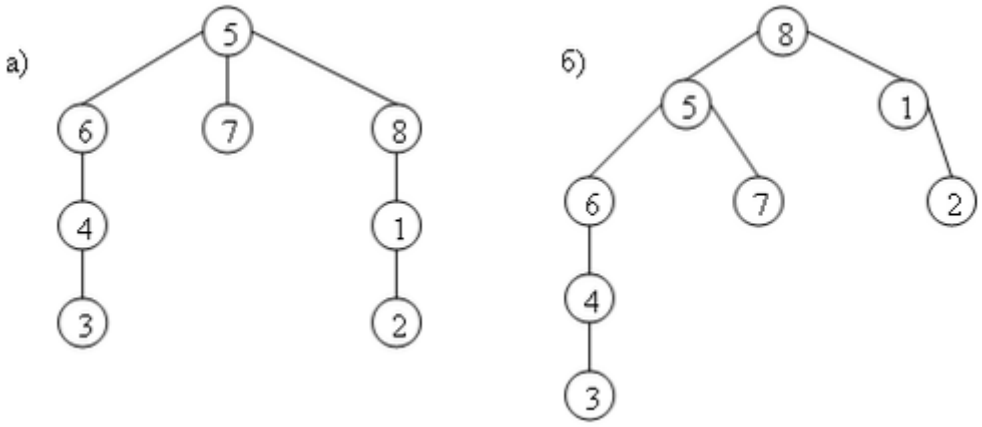


Рисунок 4.16.4 – Корневые деревья: а) с корнем в вершине 5, б) с корнем в вершине 8.

Мы показали, каким образом можно хранить дерево в виде массива предков. Также было упомянуто, что для хранения дерева можно воспользоваться и каким-либо универсальным способом представления графа в компьютере, например, матрицей смежности. В связи с этим рассмотрим следущее упражнение.

Упражнение 4.16.5

Реализовать для класса GraphMatrix новый конструктор (то есть осуществить перегрузку) на основе представления дерева в виде массива предков вершин. Параметрами конструктора является массив предков вершин. Основываясь на содержимом упражнений 4.16.1 и 4.16.2, вывести матрицу смежности графа, построенного на массиве предков вершин (8, 1, 4, 6, 0, 5, 5, 5), соответствующего дереву на рисунке 4.16.4а. Убедитесь, что и для массива предков (8, 1, 4, 6, 8, 5, 5, 0) корневого дерева на рисунке 4.16.4б будет получаться такая же матрица смежности.

Решение. В формируемом графе имеются только ребра, соединяющие вершину с ее предком. Отметим также, что в данном случае мы говорим о неориентированном графе-дереве, то есть получаемая матрица смежности должна оказаться симметричной. Тогда новый конструктор будет иметь вид:

```
// конструктор графа-дерева на основе массива предков
public GraphMatrix(int[] predecessorTree) {
        this(predecessorTree.length, 0); // вызов первого конструктора
        for (int i = 0; i < predecessorTree.length; i++)
            if (predecessorTree[i] != 0)
                // добавление ребра "вершина-родитель"
                addEdgeNotOriented(i + 1, predecessorTree[i], 1);
}
```

Осталось привести код основной программы:

```
import java.io.PrintStream;
public class ProgramWithGraph {
        public static PrintStream out = System.out;
        public static void main(String[] args) {
        int[] tree = {8, 1, 4, 6, 0, 5, 5, 5};
        GraphMatrix g = new GraphMatrix(tree);
        out.println("Матрица смежности графа:");
        int v = g.getVertices();
        for (int i = 1; i <= v; i++) {
                        for (int j = 1; j <= v; j++)
                                out.print(g.edgeWeight(i, j) + " ");
                        out.println();
        }
        }
}
```

4.16.3.4. Остовное дерево

Между обходами графа и понятием дерева есть определенная связь. Она состоит в том, что в процессе обхода графа мы пользуемся при перемещении из одной вершины в другую ребрами, их соединяющими, однако только некоторые ребра графа используются для этого. Если оставить в графе только эти ребра, потребовавшиеся для обхода, мы получим дерево. Такое дерево называется остовным. Пример остовного дерева после обхода графа в ширину приведен на рисунке 4.16.3б. Так как обход графа может развиваться в разном направлении, то из него могут получиться разные остовные деревья. То есть, если граф не является деревом, на его основе можно построить не одно остовное дерево.

Обход графа в ширину позволяет нам не только получить в качестве результата список вершин, задающий определенный порядок обхода, но и собственно механизм, в соответствии с которым мы можем сохранить дерево так, как это описано выше – в виде массива предков. По сути, полученное при обходе в ширину дерево будет являться корневым деревом, если за корень взять стартовую вершину обхода.

Дополнительной информацией, которую можно получить при обходе в ширину, являются кратчайшие расстояния от стартовой вершины до любой другой достижимой вершины графа. Однако это возможно только в том случае, если граф является невзвешенным, иначе необходимо применять более сложные алгоритмы.

Чтобы это понять, рассмотрим поведение алгоритма на примере графа, представленного на рисунке 4.16.5а. Этот граф не является деревом, в нем явно присутствуют циклы. Справа, на рисунке 4.16.5б, в графе стрелками отмечены ребра, соответствующие остовному дереву обхода в ширину, запущенному из стартовой вершины 9, а также рядом с вершинами стоят пометки в виде чисел, указывающие на расстояние до стартовой вершины. Ребра исходного графа, не попавшие в остовное дерево, представлены пунктиром.

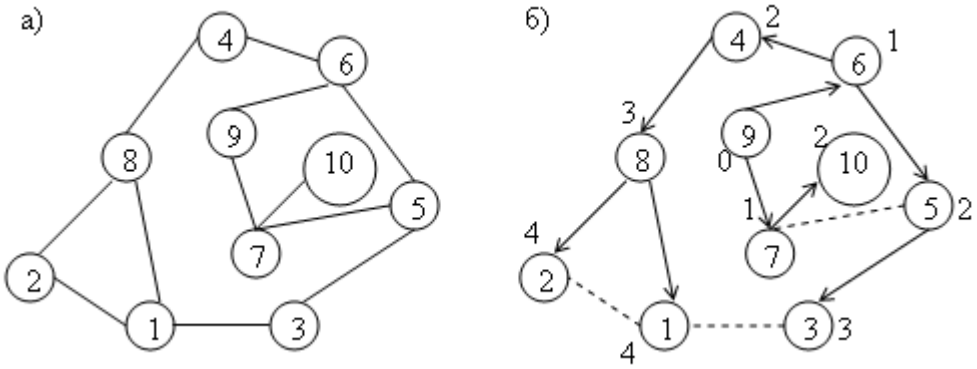


Рисунок 4.16.5 – Граф: а) в исходном виде; б) с выделенными ребрами, соответствующими остовному дереву обхода в ширину, и указанными расстояниями до стартовой вершины.

Итак, каков же алгоритм выделения дерева из имеющегося графа? Схема состоит в том, что в каждый момент времени у нас имеется множество вершин графа, уже включенных в строящееся дерево (в терминах обхода в ширину – посещенных вершин), и остальных вершин графа. На очередной итерации мы присоединяем к нашему дереву все непосещенные вершины, смежные с множество уже присоединенных вершин, а в качестве ребер дерева выбираем те ребра, которые соединяют вершины, добавленные в дерево на предыдущем этапе, и вершины, добавляемые на этой итерации. Так как нашей целью стоит построение дерева, а дерево мы храним в виде массива предков, то для каждой добавляемой вершины на текущей итерации становится известно, какая вершина из ранее включенных в дерево будет ее предком – именно ее мы и записываем в массив. Собственно, так мы и отмечаем ребро, соединяющее добавляемую вершину и ее предка. Дополнительно мы определяем расстояние от стартовой вершины до текущей как расстояние до предка текущей вершины, увеличенное на единицу.

Проиллюстрируем пошаговое поведение алгоритма на примере графа на рисунке 4.16.5. Изначально множество вершин, включаемых в окончательное дерево состоит только из стартовой вершины обхода – вершины 9.

1. Смежными с вершиной 9, но не включенными в дерево, являются вершины 6 и 7. Поэтому в дерево добавляются эти вершины и ребра (9→6) и (9→7). Теперь множество вершин дерева (6,7,9).
2. Непосещенными, смежными с данными, являются вершины 10, 4, 5, поэтому они добавляются в дерево. Обращаем ваше внимание, что вершина 5 добавляется в дерево только с ребром (6→5) без ребра (7→5). В дерево включается только одно ребро, соединяющее добавляемую вершину с множеством уже добавленных вершин. Выбрать можно любое ребро, тогда, как уже упоминалось, мы получим несколько другое дерево. Также в примере будут включены ребра (6→4) и (7→10). На этом шаге дерево содержит вершины (4, 5, 6, 7, 9, 10).
3. Смежными с данными и непосещенными являются вершины 3 и 8. Добавляя их в дерево, мы также включаем дуги (4→8) и (5→3). Множество вершин дерева примет вид (3, 4, 5, 6, 7, 8, 9, 10).
4. На последнем этапе осталось добавить вершины 1 и 2, смежные вершинами дерева. При этом дугами, соединяющими их с деревом и включаемыми в него, станут ребра (8→4) и (8→1). Так как все вершины графа добавлены в строящееся дерево, то алгоритм окончен.

Реализуем теперь представленный алгоритм в виде метода класса GraphMatrix. Предполагаем, что в качестве результата метод должен возвращать как массив предков вершин, так и кратчайшие расстояния до стартовой вершины (последнее в виде задания). Оформим возвращаемое значение в виде массива из двух элементов типа ArrayList<Integer>. Первый элемент будет отвечать за массив предков вершин, второй – за кратчайшие расстояния от стартовой вершины в невзвешенном графе. Как уже упоминалось ранее, для реализации обхода в ширину нам необходимо воспользоваться структурой данных «очередь». В качестве подходящего типа выберем динамический связанный список LinkedList<Integer>. В этом случае метод будет выглядеть так:

```
public ArrayList<ArrayList<Integer>> breadthFirstSearchToTree(int startVertex) {
        boolean[] visited = new boolean [vertices+1];
        int[] predecessor = new int [vertices+1];
        for (int i = 1; i <= vertices; i++) {
        // вершина не посещена и нет вершины-родителя
        visited[i] = false;
        predecessor[i] = 0;
        }
        LinkedList<Integer> queue = new LinkedList<Integer>();
        // посетить стартовую вершину
        visited[startVertex] = true;
        queue.addLast(startVertex);
        // цикл для посещения остальных вершин
        while (!queue.isEmpty()) {
        int currentVertex = queue.removeFirst();
        // для каждой непосещенной смежной вершины
        for (int adjVertex = 1; adjVertex <= vertices; adjVertex++) {
                        if (!visited[adjVertex] && hasEdge(currentVertex,
adjVertex)) {
                                // добавим adjVertex в дерево
                                visited[adjVertex] = true;
                                predecessor[adjVertex] = currentVertex;
                                queue.addLast(adjVertex);
                        }
        }
        }
        ArrayList<ArrayList<Integer>> result = new
ArrayList<ArrayList<Integer>>();
        result.add(new ArrayList<Integer>()); // для хранения массива предков
        for (int i = 1; i <= vertices; i++)
        result.get(0).add(predecessor[i]);
        return result;
}
```

Упражнение 4.16.6

Воспользовавшись вновь сконструированным методом формирования остовного дерева при обходе в ширину, в программе построить граф, представленный на рисунке 4.16.5а и вывести массив предков дерева, получаемого при обходе в ширину этого графа из вершины 9.

Для решения необходимо следующим образом видоизменить главный метод программы:

```
public static void main(String[] args) {
        GraphMatrix g = new GraphMatrix(10, 0);
        g.addEdgeNotOriented(1, 2, 1); g.addEdgeNotOriented(1, 3, 1);
        g.addEdgeNotOriented(1, 8, 1); g.addEdgeNotOriented(2, 8, 1);
        g.addEdgeNotOriented(3, 5, 1); g.addEdgeNotOriented(4, 6, 1);
        g.addEdgeNotOriented(4, 8, 1); g.addEdgeNotOriented(5, 6, 1);
        g.addEdgeNotOriented(5, 7, 1); g.addEdgeNotOriented(6, 9, 1);
        g.addEdgeNotOriented(7, 9, 1); g.addEdgeNotOriented(7, 10, 1);
        ArrayList<ArrayList<Integer>> dfs = g.breadthFirstSearchToTree(9);
        for (Integer predecessor : dfs.get(0)) {
        out.print(predecessor + " ");
        }
}
```

Задание 4.16.2

Усовершенствовать метод формирования остовного дерева, заполнив второй элемент массива типа ArrayList<Integer>. В процессе построения дерева внести в этот динамический массив кратчайшие расстояния до стартовой вершины. В программе из упражнения 4.16.6 дополнительно вывести вычисленные кратчайшие расстояния до стартовой вершины 9. Убедитесь, что эти значения совпадают с данными, приведенными на рисунке 4.16.5б.

4.16.4. Алгоритм Дейкстры - кратчайшее расстояние в графе

Когда речь идет о поиске кратчайшего расстояния в графе, ребра которого имеют веса, алгоритм обхода в ширину уже не дает правильного ответа. Однако переработка идеи присоединения к построенному на каком-то этапе дереву очередной вершины приводит нас к популярному алгоритму Дейкстры (Dijkstra). Этот алгоритм находит кратчайшие расстояния от заданной вершины (источника) до всех остальных.

Суть алгоритма Дейкстры заключается в следующем. На основе исходного графа фактически строится дерево. В это дерево войдут вершины графа, достижимые из источника, и ребра, по которым проходят кратчайшие пути. На каждом этапе алгоритма множество вершин графа поделено на два: одно множество U, содержащее уже добавленные в окончательное дерево вершины, и множество еще не присоединенных вершин W. Множество вершин U характерно еще и тем, что оно содержит вершины, кратчайшее расстояние до которых от источника уже

окончательно найдено. За один этап к дереву присоединяется одна вершина и одно ребро. Выбор вершины и ребра происходит следующим образом: среди всех неприсоединенных вершин множества W мы ищем такую вершину, что расстояние от нее до источника, известное на текущем этапе, меньше, чем расстояние от других вершин множества W до источника.

Предположим, такая вершина найдена. При присоединении этой вершины ко множеству U необходимо пересчитать расстояния от источника до неприсоединенных вершин. Указанные расстояния на самом деле являются кратчайшими от источника до некоторой вершины при условии, что промежуточными вершинами в пути от источника до некоторой вершины являются только представители сформированного на предыдущем этапе множества U. Поэтому при добавлении новой вершины в это множество, нужно учесть возможность, что кратчайший путь от источника проходит именно через добавляемую вершину. Пусть D[v] – кратчайшее расстояние от источника до вершины v, а P[v] – вершина, из которой мы попадаем в v при движении по кратчайшему пути от источника. Тогда псевдокод алгоритма Дейкстры может быть представлен так:

```
U = {1}; // источник – вершина №1
для каждой вершины v от 2 до N {
        D[v] = стоимость ребра, соединяющего источник и вершину i
        P[v] = (есть ребро от источника к v)? 1 : -1
}
выполним N-1 раз {
        выберем среди вершин w из множества W такие, что D[w]-минимально
        добавим w ко множеству U, исключим из W
        для каждой вершины v из множества W
                если D[v] > D[w] + “стоимость ребра от w к v” {
                        D[v] = min(D[v], D[w] + “стоимость ребра от w к v”)
                        P[v] = w
                }
}
```

В каких случаях алгоритм Дейкстры может быть применен? Во-первых, веса в графе должны быть неотрицательны. Иначе нужно использовать более общий алгоритм Беллмана-Форда. Во-вторых, описанный вариант алгоритма применим для связного графа. В случае несвязного графа при замене отсутствующего ребра в матрице смежности ребром очень большого веса мы получим нужный результат. Алгоритм работает одинаково правильно для случая ориентированных и неориентированных графов.

Рассмотрим пошагово действие алгоритма поиска кратчайших путей от вершины 1 до остальных вершин на примере графа, представленного на рисунке 4.16.6.

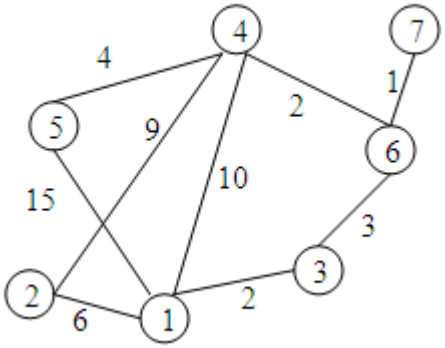


Рисунок 4.16.6 – Неориентированный граф из 7 вершин.

Возьмем в качестве бесконечного веса ребра число 100, так как оно заведомо больше суммы всех ребер.

0. Определяем вершину 1 как добавленную в дерево. Множества U = {1}, W = {2, 3, 4, 5, 6, 7}. Массивы D = (0, 6, 2, 10, 15, 100, 100), P = (1, 1, 1, 1, 1, -1, -1).
1. Находим среди вершин множества W такую, что в массиве D ее значение минимально. Такой вершиной является вершина 3, D[3] = 2. Добавляем ее во множество U:{1, 3}, удаляем из W:{2, 4, 5, 6, 7}. Теперь кратчайший путь может проходить через вершину 3. Среди вершин из множества W информация изменится только для вершины 6: D[6] = D[3]+C[3,6] = 2+3=5 < 100, P[6] = 3. Массивы: D = (0, 6, 2, 10, 15, 5, 100), P = (1, 1, 1, 1, 1, 3, -1).
2. Снова ищем среди вершин множества W такую, что в массиве D ее значение минимально: это вершина 6, D[6] = 5. Тогда множество U = {1, 3, 6}, а множество W = {2, 4, 5, 7}. Теперь кратчайший путь может проходить и через вершину 6. Среди вершин из множества W информация изменится для вершины 4: D[4] = D[6]+C[6,4] = 5+2=7 < 10, P[4] = 6; для вершины 7: D[7] = D[6]+C[6,7] = 5+1=6 < 100, P[7] = 6. Массивы: D = (0, 6, 2, 7, 15, 5, 6), P = (1, 1, 1, 6, 1, 3, 6).
3. С минимальным значением в массиве D нам подходят из множества W вершины 2 и 7. Выбрать можно любую, выберем 2. Тогда U = {1, 2, 3, 6}, а множество W = {4, 5, 7}. Ни до какой вершины кратчайшее расстояние улучшено не будет.
4. Как и в предыдущем пункте, нам подходит вершина 7. Эта вершина тоже не даст изменений в расстояниях, а множества примут вид: U = {1, 2, 3, 6, 7}, W = {4, 5}.
5. Под критерии отбора подходит теперь только вершина 4, D[4] = 7. Тогда множества будут: U = {1, 2, 3, 4, 6, 7}, W = {5}. Изменения коснутся вершины 5: D[5] = D[4]+C[4,5] = 7+4=11 < 15, P[5] = 4. Массивы: D = (0, 6, 2, 7, 11, 5, 6), P = (1, 1, 1, 6, 4, 3, 6).
6. Осталась только вершина 5. Она будет добавлена в дерево, но никаких изменений в массивах D и P не произойдет.

Итог: D = (0, 6, 2, 7, 11, 5, 6), P = (1, 1, 1, 6, 4, 3, 6).

Рассмотрим вопрос о кратчайшем пути от вершины 1 до вершины 5. Кратчайшее расстояние D[5]=11. Для построения кратчайшего пути нужно формировать его «с конца». Двигаясь по кратчайшему пути от вершины 1 к 5, мы попали в 5 из вершины P[5]=4, в вершину 4 – из P[4]=6, в вершину 6 – из P[6]=3, в вершину 3 – из P[3]=1. Таким образом, путь выглядит так: (1→3→6→4→5).

Перейдем к реализации алгоритма Дейкстры для нашего класса GraphMatrix. Опишем следующие поля:

```
public static final int NO_PATH = -1;
private int sourceVertex; // вершина-источник
private int[] pathTree;
private int[] pathWeight;
```

Массив pathTree будет соответствовать массиву P, pathWeight – массиву D из объяснений и псевдокода.

Разобъем логику алгоритма на две составляющие: один метод будет заключаться в запуске алгоритма Дейкстры, другие методы будут предоставлять доступ к стоимости кратчайшего пути и формированию самого пути. Это связано с тем, что достаточно один раз выполнить алгоритм Дейкстры, чтобы затем, не пересчитывая, пользоваться этой информацией для определения кратчайшего расстояния от одного источника до всех остальных вершин. Кроме того, формат хранения кратчайшего пути в самом классе отличается от привычного представления пути от источника к какой-то вершине. Поэтому необходимо явно воспользоваться идеей инкапсуляции, скрыв формат представления и предоставив удобный интерфейс получения кратчайшего пути.

Начнем с самого алгоритма Дейкстры:

```
public void startDijkstra(int startVertex) {
        sourceVertex = startVertex;
        pathTree = new int [vertices+1];
        pathWeight = new int [vertices+1];
        boolean[] vertexAdded = new boolean[vertices+1];
        for (int i = 1; i <= vertices; i++) {
        vertexAdded[i] = false; // вершина не добавлена в дерево
        if (hasEdge(sourceVertex, i)) {
                pathTree[i] = sourceVertex;
                        pathWeight[i] = edgeWeight(sourceVertex, i);
        }
        else {
                        pathTree[i] = NO_PATH;
                        pathWeight[i] = infinity;
        }
        }
        // добавляем стартовую вершину
        vertexAdded[sourceVertex] = true;
        pathTree[sourceVertex] = sourceVertex;
        pathWeight[sourceVertex] = 0;
        // цикл по добавлению очередной вершины в дерево
        for (int i = 1; i <= vertices - 1; i++) {
        // найти недобавленную вершину
        int minVertex = 1;
        while (minVertex <= vertices && vertexAdded[minVertex])
                        minVertex++;
        // найти недобавленную вершину с минимальной стоимостью пути
```

```
        for (int j = minVertex + 1; j <= vertices; j++)
                        if (!vertexAdded[j] && pathWeight[j] <
pathWeight[minVertex])
                                minVertex = j;
        // добавить minVertex в дерево
        vertexAdded[minVertex] = true;
        // пересчитать кратчайшие расстояния
        for (int j = 1; j <= vertices; j++)
                if (!vertexAdded[j] && hasEdge(minVertex, j)) {
                int d = pathWeight[minVertex] + edgeWeight(minVertex, j);
                    if (d < pathWeight[j]) {
                         pathWeight[j] = d;
                         pathTree[j] = minVertex;
                    }
                }
        }
}
```

Упражнение 4.16.7

Реализуйте для класса GraphMatrix следующие методы:

а) метод getShortestDistance должен вернуть величину кратчайшего расстояния до переданной в параметре метода вершины, или -1, если пути нет

б) метод getShortestPath должен вернуть в виде массива ArrayList путь от стартовой вершины до вершины, переданной в качестве параметра. Учтите случай отсутствия пути, случай совпадения стартовой и конечной вершины.

Составьте программу, в которой, воспользовавшись написанным классом, нужно вывести на экран кратчайшие расстояния в графе на рисунке 4.16.6 от вершины 1 до остальных вершин. Также вывести кратчайший путь от вершины 1 до вершины 5.

Перечисленные в упражнении 4.16.7 методы имеют вид:

```
public int getShortestDistance(int targetVertex) {
        if (pathTree[targetVertex] == NO_PATH)
        return -1;
        return pathWeight[targetVertex];
}
public ArrayList<Integer> getShortestPath(int targetVertex) {
        ArrayList<Integer> path = new ArrayList<>();
        if (targetVertex == sourceVertex) {
        path.add(sourceVertex);
        return path;
        }
        if (pathTree[targetVertex] == NO_PATH)
```

```
        return null;
        Stack<Integer> stack = new Stack<>();
        int t = targetVertex;
        do {
        stack.push(t);
        t = pathTree[t];
        } while (t != sourceVertex);
        stack.push(sourceVertex);
        while (!stack.isEmpty())
        path.add(stack.pop());
        return path;
}
```

Метод main основной программы тогда можно составить так:

```
GraphMatrix g = new GraphMatrix(7, 1000);
g.addEdgeNotOriented(1, 2, 6); g.addEdgeNotOriented(1, 3, 2);
g.addEdgeNotOriented(1, 4, 10); g.addEdgeNotOriented(1, 5, 15);
g.addEdgeNotOriented(2, 4, 9); g.addEdgeNotOriented(3, 6, 3);
g.addEdgeNotOriented(4, 5, 4); g.addEdgeNotOriented(4, 6, 2);
g.addEdgeNotOriented(6, 7, 1);
g.startDijkstra(1);
for (int i = 1; i <= 7; i++)
        out.print(g.getShortestDistance(i) + " ");
out.println();
ArrayList<Integer> path = g.getShortestPath(5);
for (Integer vertex : path) {
        out.print(vertex + " ");
}
```

Задание 4.16.3

Временно опишите как private (или просто удалите) метод getShortestDistance класса GraphMatrix из упражнения 4.16.7. В главной программе этого же упражнения уберите вывод кратчайших расстояний до всех вершин, оставьте вывод кратчайшего пути до вершины 5. Рассчитайте кратчайшее расстояние до вершины 5, воспользовавшись методами getShortestPath и edgeWeight.

4.16.5. Эйлеров цикл.

Возникновение теории графов часто связывают с так называемой задачей о кенигсбергских мостах, которую в свое время решал Леонард Эйлер. Через город Кенигсберг (ныне город

Калининград в Калининградской области) протекает река Преголя, на которой имеется два острова (см. рисунок 4.16.7а). Имеющиеся семь мостов соединяют не только острова с обоими берегами, но и между собой. Эйлер сформулировал в терминах теории графов и решил следующую задачу: а можно ли пройти по всем мостам ровно один раз, вернувшись при этом обратно на исходный берег? Не приводя решение ученого, скажем, что им было показано отсутствие существования такого пути.

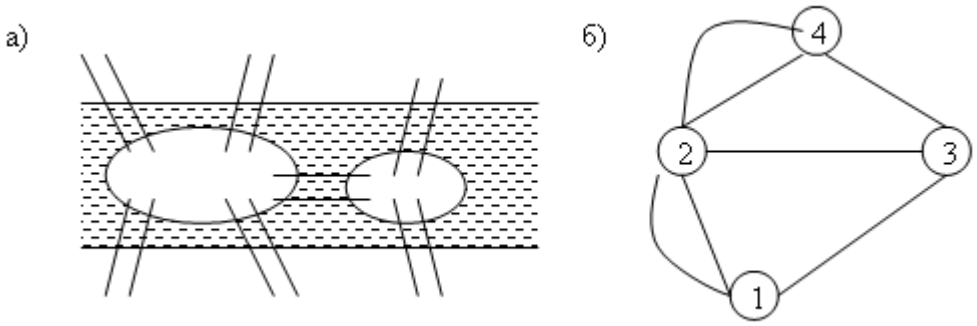


Рисунок 4.16.7 – а) кенигсбергские мосты; б) граф для кенигсбергских мостов

Данная задача имеет общую постановку, суть которой заключается в том, чтобы определить, существует ли в связном графе такой цикл, который проходит по всем дугам графа, но при этом по каждой дуге мы проходим ровно один раз. Такой путь называется эйлеровым циклом, а граф – эйлеровым. Чтобы выяснить, возможно ли построить такой путь, нам нужно рассмотреть следующие понятия.

Степенью исхода вершины ориентированного графа называется количество ребер, выходящих из нее, степенью входа – количество ребер, входящих в нее. Для неориентированного графа рассматривается понятие **степени** вершины – количество ребер, соединяющих эту вершину с другими.

Так как при построении эйлерова цикла мы должны пройти по всем дугам ориентированного графа, то это означает, что в каждую вершину мы войдем столько раз, какова ее степень входа, и выйдем столько раз, какова ее степень исхода. Чтобы можно было выйти из вершины, в которую мы вошли, для посещения остальных дуг графа и организации цикла, необходимо требовать равенства степеней исхода и входа у каждой вершины. Оказывается, что это требование является также и достаточным условием существования эйлерова цикла для ориентированного графа.

Для того чтобы в ориентированном графе существовал эйлеров цикл, необходимо и достаточно, чтобы для каждой вершины было выполнено: степень входа вершины и ее степень выхода должны совпадать.

Иногда возникает задача с меньшими требованиями, чем задача построения эйлерова цикла. В ней требуется найти путь, который проходит по всем дугам только один раз, но он не обязан быть циклическим. То есть все требования те же, за исключением того, что теперь начаться и закончиться путь может в разных вершинах. Такой путь, или цепь, тоже называется эйлеровым.

Граф, в котором существует эйлеров путь, называется полуэйлеровым. Существует и критерий существования эйлеровой цепи: у всех вершины, кроме двух, должны совпадать степени входа и выхода. У одной вершины степень входа должна быть на единицу больше степени выхода, у другой – наоборот: степень входа должна быть на единицу меньше степени выхода.

В неориентированном графе несколько другие критерии существования эйлерова цикла. Эйлеров цикл существует, если степень каждой вершины четна.

Алгоритм построения эйлерова цикла опирается на базовый алгоритм обхода графа – обход в глубину. Для построения эйлерова цикла нам, правда, понадобятся два стека (а не один): А и Б. Один стек будет промежуточным, а второй будет в итоге содержать вершины графа в порядке обхода эйлерова цикла. Предположим, что мы убедились в существовании эйлерова цикла, проверив условия теоремы. В этом случае не важно, из какой вершины будет начат обход. Пусть это вершина номер 1. В начале оба стека пусты. Назначим ее текущей вершиной. Будем двигаться по непосещенным ребрам от вершины к вершине, помещая их в стек А, пока не окажемся в ситуации, когда все ребра, выходящие из вершины, уже пройдены. Это означает, что мы вернулись в стартовую вершину, замкнув цикл. Однако, не все ребра графа (а, может быть, даже и не все вершины) могли быть пройдены. Будем двигаться обратно по сформированному циклу. Порядок обратного пути мы получаем, извлекая вершины из стека А, помещая их при этом в стек Б. Двигаясь обратно, мы обязательно наткнемся на вершину с выходящими из нее непосещенными ребрами. Из такой вершины мы снова начнем обход в глубину, снова заполняя стек А. Когда стек А опустеет окончательно, в стеке Б будут содержаться вершины в порядке обхода по эйлерову циклу.

На псевдокоде алгоритм построения эйлерова цикла имеет вид:

```
поместить 1 в стек А
пока стек А не пуст {
        t – вершина стека А
        если есть непройденное ребро (t→u) {
                пометить ребро (t→u) как пройденное
        поместить u в стек А
        } иначе {
                извлечь t из стека А
                поместить t в стек Б
        }
}
```

Содержимое стека Б и будет ответом.

Займемся реализацией алгоритма построения эйлерова цикла. Результатом нашего нового метода в классе GraphMatrix будет список вершин в порядке обхода по циклу. Предположим также, что метод hasEulerCycleAsDirectedGraph, проверяющий факт существования эйлерова цикла в ориентированном графе, уже реализован. Тогда искомый метод может быть написан так:

```
public ArrayList<Integer> getEulerCycleAsDirectedGraph() {
        if (!hasEulerCycleAsDirectedGraph())
        return null;
        // создадим для каждой вершины список смежных вершин,
        // что соответствует выходящему ребру
        LinkedList<Integer>[] adjacencyList = new LinkedList[vertices+1];
        for (int i = 1; i <= vertices; i++)
        adjacencyList[i] = new LinkedList<>();
        for (int i = 1; i <= vertices; i++)
        for (int j = 1; j <= vertices; j++)
                if (hasEdge(i, j))
                        adjacencyList[i].add(j);
        Stack<Integer> sTemp = new Stack<>(), sResult = new Stack<>();
        sTemp.push(1);
        while (!sTemp.isEmpty()) {
        int t = sTemp.peek();
        if (!adjacencyList[t].isEmpty())
                        sTemp.push(adjacencyList[t].pollFirst());
        else
                        sResult.push(sTemp.pop());
        }
        ArrayList<Integer> cycle = new ArrayList<>(sResult.size());
        while (!sResult.isEmpty())
        cycle.add(sResult.pop());
        return cycle;
}
```

Отметим, что для реализации алгоритма мы фактически перешли от матрицы смежности к спискам смежности. Это позволило нам перечислять ребра, выходящие из вершины, рассматривая каждое ребро только один раз.

Упражнение 4.16.8

Дополните описание класса GraphMatrix реализацией метода hasEulerCycleAsDirectedGraph, проверяющего выполнение критерия существования эйлерова цикла в ориентированном графе. В основной программе создайте граф с 7 вершинами и добавьте в него следующие дуги:

(1,2), (1,4), (2,1), (2,3), (2,5), (3,6), (4,1), (4,2), (4,7), (5,2), (5,4), (5,7), (6,5), (7,4), (7,5). Выведите вершины графа в порядке обхода по эйлерову циклу, воспользовавшись реализованным методом.

Функция, проверяющая наличие эйлерова цикла, будет иметь вид:

```
public boolean hasEulerCycleAsDirectedGraph() {
        int[] vertexDegree = new int [vertices+1];
        for (int i = 0; i <= vertices; i++)
```

```
        vertexDegree[i] = 0;
        for (int i = 1; i <= vertices; i++)
        for (int j = 1; j <= vertices; j++)
                if (matrix[i][j] != 0) {
                        vertexDegree[i]++; // выходящее ребро
                        vertexDegree[j]--; // входящее ребро
                }
        for (int i = 1; i <= vertices; i++)
        if (vertexDegree[i] != 0)
                        return false;
        return true;
}
```

Главный метод main основной программы может быть реализован так:

```
public static void main(String[] args) {
        GraphMatrix g = new GraphMatrix(7, 0);
        g.addEdge(1, 2, 1); g.addEdge(1, 4, 1);
        g.addEdge(2, 1, 1); g.addEdge(2, 3, 1);
        g.addEdge(2, 5, 1); g.addEdge(3, 6, 1);
        g.addEdge(4, 1, 1); g.addEdge(4, 2, 1);
        g.addEdge(4, 7, 1); g.addEdge(5, 2, 1);
        g.addEdge(5, 4, 1); g.addEdge(5, 7, 1);
        g.addEdge(6, 5, 1); g.addEdge(7, 4, 1);
        g.addEdge(7, 5, 1);
        if (g.hasEulerCycleAsDirectedGraph()) {
        ArrayList<Integer> cycle = g.getEulerCycleAsDirectedGraph();
        for (Integer vertex : cycle)
                out.print(vertex + " ");
        }
      else
        out.println("Нет эйлерова цикла!");
}
```

Задание 4.16.4

1) Реализуйте метод проверки существования эйлерова цикла для случая неориентированного графа.

2) Реализуйте метод построения эйлерова цикла для неориентированного графа.

3) В основной программе постройте граф из 7 вершин, добавив в него ребра, соответствующие графу на рисунке 4.16.8. Проверив существование эйлерова цикла, выведите его в ответ.

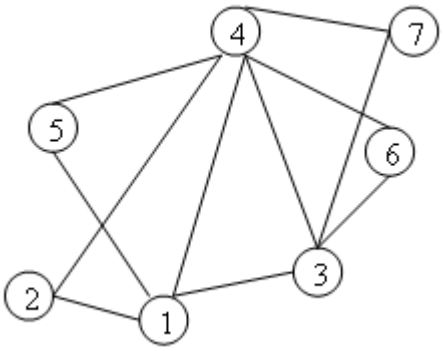


Рисунок 4.16.8 – Неориентированный граф из 7 вершин.

4.16.6. Другие задачи теории графов

Как уже было сказано, способ представления графа в памяти компьютера определенным образом влияет на эффективность работы используемых алгоритмов с точки зрения времени работы и требуемой памяти. Что касается памяти, достаточно очевидно, что списки смежности при должной реализации выигрывают по этой позиции у матрицы смежности. Что касается времени работы, все рассмотренные до этого момента алгоритмы на графах имеют не очень большую временную сложность.

Обходы в ширину и в глубину имеют порядок времени работы $O(V^2)$ для случая матрицы смежности и $O(E)$ для случая списков смежности (здесь V – количество вершин графа, E – количество ребер). Время работы алгоритма Дейкстры сильно зависит от схемы реализации и используемых структур данных, в нашем простом случае это $O(V^2)$.

Когда встает выбор представления графа в виде матрицы смежности или списков смежности, мы должны понимать, что количество ребер E в неориентированном графе с V вершинами может достигать порядка $V^2/2$. Такой граф иногда называют **насыщенным**. Граф, в котором количество ребер мало по сравнению с возможным их количеством и приблизительно равно количеству вершин, умноженному на небольшое (до десятков) число, иногда называют **разреженным**. Формально, не существует четкого разделения на разреженные и насыщенные графы. Практика показывает, что насыщенный граф в большинстве задач удобнее хранить в виде матрицы смежности, разреженный – в виде списков смежности.

Приведенные выше алгоритмы теории графов, а так же многие не названные, даже при количестве вершин порядка нескольких тысяч выполняются за разумное время (до нескольких секунд) при нераспараллеленной реализации. Существуют же задачи, которые уже при размере входных данных около сотни не могут быть выполнены на современных компьютерах даже за время жизни одного человека. К таким задачам относятся так называемые «NP-полные» задачи. Многие из представленных ниже задач относятся к этому классу, на счет некоторых окончательное мнение научного сообщества еще не установилось, однако вычислительная сложность приведенных ниже задач однозначно высока.

1. Задача об изоморфизме графов.

Граф представляет собой некоторое абстрактное понятие, даже если описывает какую-либо реальную физическую или информационную систему. При этом, изображая граф в виде рисунка, мы уделяем внимание в первую очередь вопросу о том, сколько вершин у изображаемого графа и какие у него имеются ребра. Ребра мы можем изобразить в виде прямых линий, можем в виде

дуг. Они могут пересекаться на рисунке, а могут и не пересекаться, но разные с виду рисунки могут соответствовать одному и тому же графу. Задача об изоморфизме графов ставит вопрос о том, являются ли два имеющихся графа по сути одинаковыми. Проверку изоморфизма двух графов можно выполнить, убедившись в существовании способа так перенумеровать вершины одного из графов, что получаемая в этом случае матрица смежности будет являться копией матрицы смежности второго графа. Однако перебор всех возможных перестановок номеров вершин дает оценку сложности алгоритма O(V!), где V – количество вершин в графе.

2. Задача об укладке графа на плоскости (проверка планарности графа).

Граф называется планарным, если его можно нарисовать на плоскости так, что все вершины расположены на плоскости в разных точках, а все имеющиеся ребра графа проведены в виде линий (не обязательно прямых) так, чтобы они не пересекались друг с другом. Интересно, что решение этой задачи тесно связано с решением задачи об изоморфизме графов. Теорема Понтрягина-Куратовского утверждает, что определение планарности графа может быть сведено к проверке на отсутствие в нашем графе подграфа, изоморфного какому-либо из двух конкретных специальных графов: двудольному графу $K_{3,3}$ с тремя вершинами в каждой из долей и полному графу K_5 с пятью вершинами, которые представлены на рисунке 4.16.9.

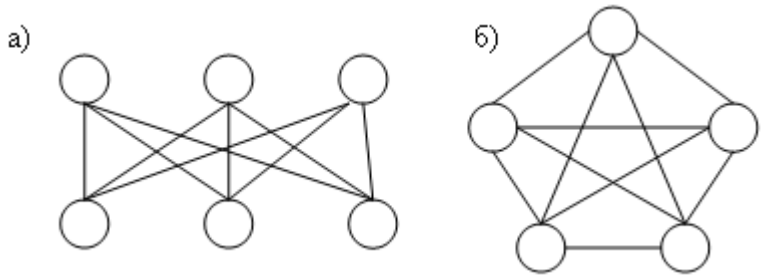


Рисунок 4.16.9 – Непланарные графы: а) $K_{3,3}$ и б) K_5.

3. Гипотеза четырех красок.

Эта гипотеза имеет непосредственное отношение к одной практической задаче. Предположим, что мы решили нарисовать политическую карту мира. Как известно, стран в мире достаточно много – несколько сотен, а хотя человеческий глаз и способен отличить большое количество оттенков цветов, удобно было бы использовать небольшой контрастный набор для рисования территорий государств. Возникает вопрос: а какого минимального количества цветов достаточно, чтобы раскрасить территории государств так, чтобы ни одно государство не граничило с другим государством, территория которого отмечена таким же цветом? Современная теория утверждает, что всегда достаточно четырех красок. Однако алгоритм поиска такой раскраски графа имеет высокую сложность.

Необходимо отметить, что изложенный вариант задачи с точки зрения теории графов соответствует вопросу о том, какого минимального количества красок достаточно для раскраски планарного графа. Гипотеза о том, что для этого достаточно четырех красок, была высказана Францисом Гутри в середине 19 века. Доказана она была в конце 20 века при помощи компьютерной программы.

4. Задача о нахождении максимальной клики (полносвязного подграфа).

В этой задаче ставится вопрос о том, чтобы определить максимальное множество вершин в имеющемся графе таких, что они образуют полный подграф, то есть в рассматриваемом графе

каждая из вершин выбранного множества соединена со всеми другими вершинами множества. Эта задача также имеет вычислительную сложность класса «NP-полных» задач. Сформулирована она была Ричардом Карпом.

5. Проверка существования гамильтонова цикла.

Говорят, что в графе существует гамильтонов цикл (а сам граф в этом случае является гамильтоновым графом), если представленный цикл проходит через все вершины графа, посещая каждую из них ровно один раз и возвращаясь в исходную.

4.16.7. Гамильтонов цикл

Проблема поиска гамильтонова цикла в графе естественным образом возникает в определенных практических задачах. Представим себе музей, в котором имеется некоторое количество выставочных залов. Необходимо так составить маршрут по экспозиции, чтобы каждый из имеющихся залов был посещен ровно один раз, при этом нужно вернуться в исходную точку.

Аналогичная проблема возникает при решении другой задачи. Пусть есть несколько человек, и их надо посадить вокруг круглого стола. У каждого из них есть в группе знакомые. Можно ли рассадить их так, чтобы слева и справа от каждого человека сидели его знакомые.

Другая задача состоит в обходе конем всех клеток шахматной доски. Представим себе шахматную доску, где в левом верхнем углу стоит шахматная фигура «конь». Задача состоит в том, чтобы переставляя коня по правилам шахмат и посещая каждую клетку доски строго один раз, обойти всю доску и вернуться в исходную клетку. Если представлять себе клетки доски как вершины графа, а ребрами соединены те клетки, между которыми может перемещаться конь за один ход, то приведенная задача трансформируется в классическую задачу о поиске гамильтонова цикла.

И, наконец, еще одной классической задачей является задача коммивояжера. Ее суть заключается в поиске такого маршрута по городам (вершинам графа), чтобы посетив их все единожды с торговыми целями, вернуться в исходную точку. Общая постановка задачи определяет стоимости перехода между вершинами. В такой форме задача имеет еще более высокие требования, чем мы будем изучать, так как требуется среди всех возможных замкнутых маршрутов выбрать цикл с минимальной стоимостью пути.

Поиск гамильтонова цикла для графа возможен в первую очередь с помощью рассмотрения всех возможных путей и проверкой каждого из них на выполнение необходимых условий. В общем случае количество таких путей составляет порядка N!, где N – количество вершин графа. Уже для N = 20 количество вариантов составляет около $2*10^{18}$. Перебор такого количества путей методом «грубой силы» (“brute force”) потребует необозримого количества времени. Однако существует возможность сокращения времени работы алгоритма. Во-первых, надо обратить внимание, что мы ищем любое (!) решение, а не все возможные. Таким образом, как только алгоритм найдет какое-то решение, он остановится. Во-вторых, в таких задачах при построении пути выполняют отсечение вариантов. Например, если построенный частично путь уже «привел в тупик», то есть не все вершины посещены, но дальнейшего хода нет, то такой путь не дает ответа. Все пути, начинающиеся так же, тоже выбрасываются из рассмотрения. В-третьих, в задачах с полным перебором стараются использовать некоторые эвристические идеи, которые позволяют найти решение несколько быстрее за счет направления поиска решения «в определенную сторону».

Что касается эвристики при построении гамильтонова цикла, то можно воспользоваться аналогом эвристики Варнсдорфа, предложенного им для случая обхода конем шахматной доски. Она состоит в том, чтобы при построении гамильтонова цикла двигаться в сторону тех вершин, количество выходов из которых мало. Причем надо понимать, что это количество выходов динамически изменяется в процессе формирования пути. Идея основана на том, что посещая как можно раньше такие вершины, мы надеемся не попасть в дальнейшем «в тупик», с одной стороны, и не «отрезать» себе путь от непосещенных вершин, с другой.

Рассмотрим поиск гамильтонова цикла методом перебора всевозможных путей на примере следующего графа из 5 вершин, представленного на рисунке 4:

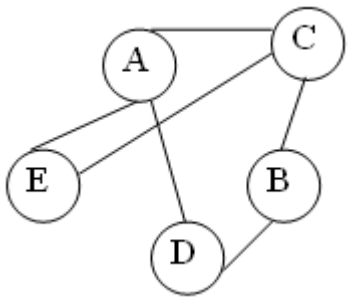


Рисунок 4.16.10 – Граф для поиска гамильтонова цикла

Нас интересуют всевозможные пути, начинающиеся, например, в вершине A. Это не умаляет общности решения, так как если гамильтонов цикл существует, то он обязательно проходит через вершину A, следовательно, обход можно начинать из нее. Для перебора путей мы просто составим все перестановки вершин от B до E и будем проверять для каждой из них, существует ли в имеющемся графе такой путь, начинающийся в вершине A и в ней же и заканчивающийся. Таких перестановок 4! = 24 варианта, начиная с BCDE и заканчивая EDCB.

Первые из перебираемых путей, начинающиеся на AB (ABCDEA, ABCEDA, …, ABEDCA), не подходят просто потому, что в графе нет ребра AB. Циклы ACBDEA, ACBEDA, ACDBEA и ACDEBA не существуют из-за отсутствия ребер DE, BE и CD в последних двух случаях, соответственно. Следующие в переборе циклы ACEBDA и ACEDBA не существуют ввиду отсутствия ребер EB и ED. И только 13 по счету вариант ADBCEA дает нам ответ в виде существующего гамильтонова цикла. В этом графе другого гамильтонова цикла нет, с точностью до направления обхода.

Стоит же в указанном графе применить эвристику Варнсдорфа, то ответ будет найден сразу же, так как из вершины A ведут ребра только в две вершины с минимальным в данном случае количеством выходов, равным двум, - в вершину D (вершина C имеет три выхода) и E. Из вершины D путь ведет только в B, оттуда в C, далее в E и путь замыкается стартовой вершиной. А из вершины E циклический путь тоже строится без каких-либо ответвлений.

Если не применять указанную эвристику, а строить гамильтонов цикл по возможным ребрам, то сначала алгоритм попытается пройти из вершины A в вершину C с дальнейшим переходом в B и D. Попав в вершину D, мы обнаружим тупик, так как вершина A есть конечная цель только в том случае, если мы обошли все другие вершины, а это не так – вершина E отсутствует в построенном пути. И только тогда мы пойдем по правильному пути из стартовой вершины.

Современная теория графов не дает достаточного и необходимого условия существования гамильтонова цикла в графе. Однако определенные теоремы позволяют в некоторых случаях

установить ответ на этот вопрос. Скажем, что рассматривать вопрос наличия гамильтонова цикла естественно только в связном графе, так как иначе не все вершины достижимы. Кроме того, очевидно утверждение, что если граф полный (граф, в котором каждая пара вершин соединена ребром), то в нем существует гамильтонов цикл. Для этого достаточно составить путь, перечислив вершины в порядке возрастания. Существует также необходимое условие существования гамильтонова цикла.

Если в неориентированном графе имеется гамильтонов цикл, то в этом графе нет вершины, степень которой меньше 2.

В качестве достаточных условий можно сослаться на условие Поша и условие Дирака существования гамильтонова пути. Мы не будем их приводить, однако интересующиеся легко могут найти формулировки этих условий в соответствующей литературе.

Перейдем к вопросу о реализации построения гамильтонова цикла. Одним из самых удобных способов запрограммировать этот алгоритм является рекурсия. Задача поиска гамильтонова цикла перекликается с огромным количеством похожих по сути задач, когда решение, которое мы ищем, состоит из нескольких этапов, и на каждом этапе имеется выбор перед определенным набором действий. Обсуждаемый способ представляет собой метод «проб и ошибок». Такие алгоритмы в программировании относятся к категории «алгоритмов с возвратом», когда в случае обнаружения «тупика» в построении решения мы вынуждены возвращаться назад, чтобы выбрать из доступных другое действие. Рекурсивное решение можно представить в виде псевдокода следующим образом:

```
Попытка_следующего_хода {
        подготовить выбор действий
        цикл {
                выбрать очередное действие из возможных
                записать действие
                Если решение неполное {
                        Попытка_следующего_хода
                        Если неудача стереть запись
                }
        } пока (решение не нашлось И еще есть возможное действие)
}
```

Удобно разделить алгоритм построения гамильтонова цикла на 2 метода для класса GraphMatrix. Один метод, закрытый, будет рекурсивно обеспечивать перебор с возвратом, а второй – открытый, предоставит удобный интерфейс для получения построенного цикла. Иногда в таких случаях говорят, что второй метод является «оберткой» для первого. Рекурсивный метод перебора вариантов с возвратом должен в качестве параметров обязательно получать длину уже построенного пути, чтобы в нужный момент проверить возможность перехода из последней в построенном пути вершины в стартовую вершину цикла.

```
private boolean tryNextVertexInHamiltonCycle(ArrayList<Integer> path,
        int moveNumber, boolean[] visited, int startVertex) {
        if (moveNumber == vertices)
        return (hasEdge(path.get(vertices-1), startVertex));
        for (int i = 1; i <= vertices; i++)
        if (!visited[i] && hasEdge(path.get(moveNumber-1), i)) {
                        visited[i] = true;
                        path.add(i); // moveNumber
                        if (tryNextVertexInHamiltonCycle(path, moveNumber+1,
visited, startVertex))
                                return true;
                        path.remove(moveNumber);
                        visited[i] = false;
        }
        return false;
}
```

Интерфейсный открытый метод выглядит несколько короче:

```
public ArrayList<Integer> getHamiltonCycle() {
        boolean[] visited = new boolean[vertices+1];
        ArrayList<Integer> hamiltonCycle = new ArrayList<>(vertices + 1);
        // поместим первую вершину в начало пути
        visited[1] = true; hamiltonCycle.add(1);
        if (tryNextVertexInHamiltonCycle(hamiltonCycle, 1, visited, 1))
        return hamiltonCycle;
        else
        return null;
}
```

Упражнение 4.16.9

1) Попробуйте вручную найти гамильтонов цикл в графе, изображенном на рисунке 4.16.11а. Этот граф соответствует развертке додекаэдра.

2) Напишите программу, воспользовавшись вновь написанными методами класса GraphMatrix, в которой нужно создать граф из 20 вершин на рисунке 4.16.11б, а затем вывести гамильтонов цикл.

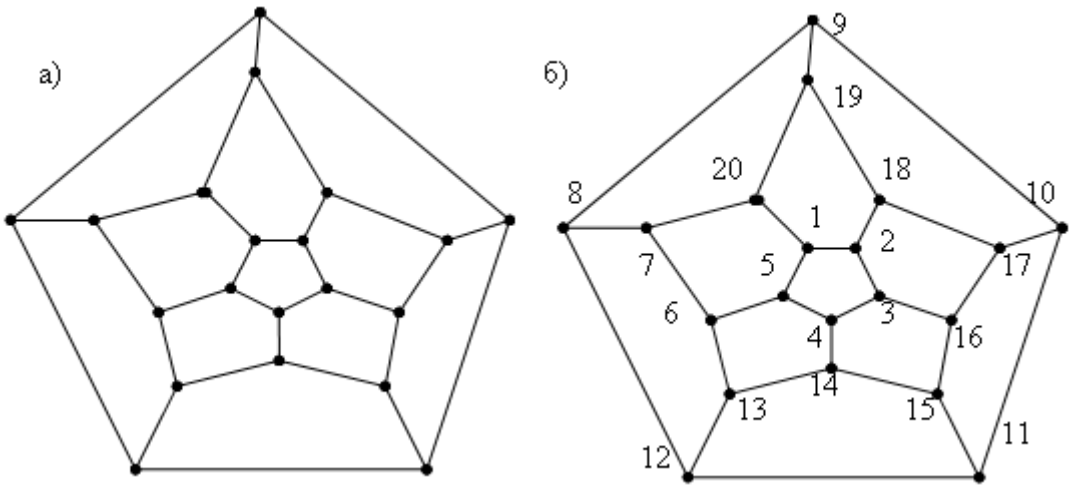


Рисунок 4.16.11 – Граф для поиска гамильтонова цикла: а) без нумерации вершин, б) с нумерацией

Решение. Главный метод main основной программы будет иметь следующий вид:

```
GraphMatrix g2 = new GraphMatrix(20, 0);
g2.addEdgeNotOriented(1, 2, 1);
g2.addEdgeNotOriented(1, 5, 1);
//......
g2.addEdgeNotOriented(18, 19, 1);
g2.addEdgeNotOriented(19, 20, 1);
ArrayList<Integer> hamiltonCycle = g2.getHamiltonCycle();
out.println("Граф из 20 вершин от додекаэдра...");
if (hamiltonCycle == null)
        out.println("Нет гамильтонова цикла");
else {
        out.println("Гамильтонов цикл:");
        for (int i = 0; i < hamiltonCycle.size(); i++)
        out.print(hamiltonCycle.get(i) + " ");
}
```

Задание 4.16.5

Попробуйте самостоятельно построить гамильтонов цикл обхода конем шахматной доски размера 6x6, затем 8x8. Напишите программу для построения гамильтонова цикла обхода конем для доски произвольного размера NxN. Убедитесь в длительном времени поиска решения уже для N = 8. Для ускорения работы алгоритма реализуйте упомянутую выше эвристику Варнсдорфа.

Литература

1. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 1296 с.

2. Ахо А., Хопкрофт Д., Ульман Д.Д. Структуры данных и алгоритмы.: Пер. с англ.: Уч. пос. – М.: Издательский дом «Вильямс», 2000. – 384 с.
3. Вирт Н. Алгоритмы и структур данных: Пер. с англ. – М.: Мир, 1989. – 360 с.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Колганову Константину Михайловичу.