Урок 7

Графы

Рассмотрим работу с одной из самых гибких и универсальных структур.

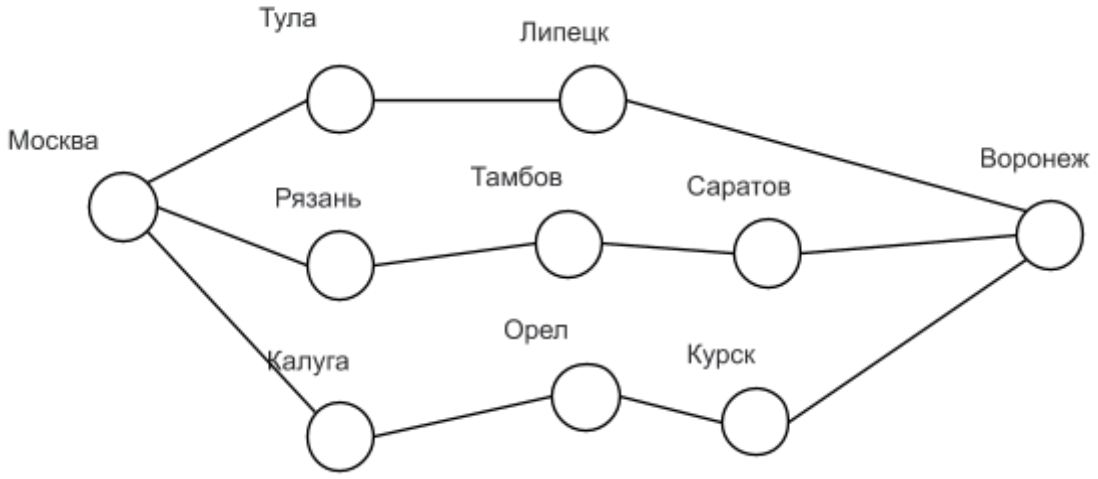
Введение

Графы относятся к абстрактным структурам данных и представлены несколькими типами: взвешенные (или связанные) и несвязанные (или направленные). В математике граф считается частным случаем дерева. Одной из основных задач, которую можно решить с помощью графов — нахождение кратчайшего пути.

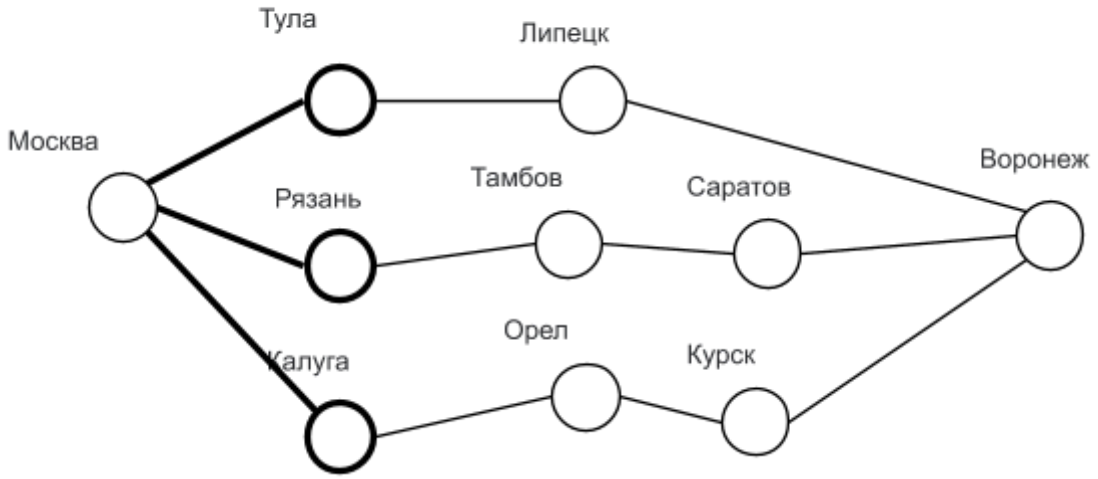
В этом уроке мы подробно рассмотрим графы. Узнаем, что такое ребра и вершины, связанный и несвязанный граф. Рассмотрим матрицу и список смежности, направленный и ненаправленный граф. Научимся обходить графы в глубину и ширину.

Графы в примерах

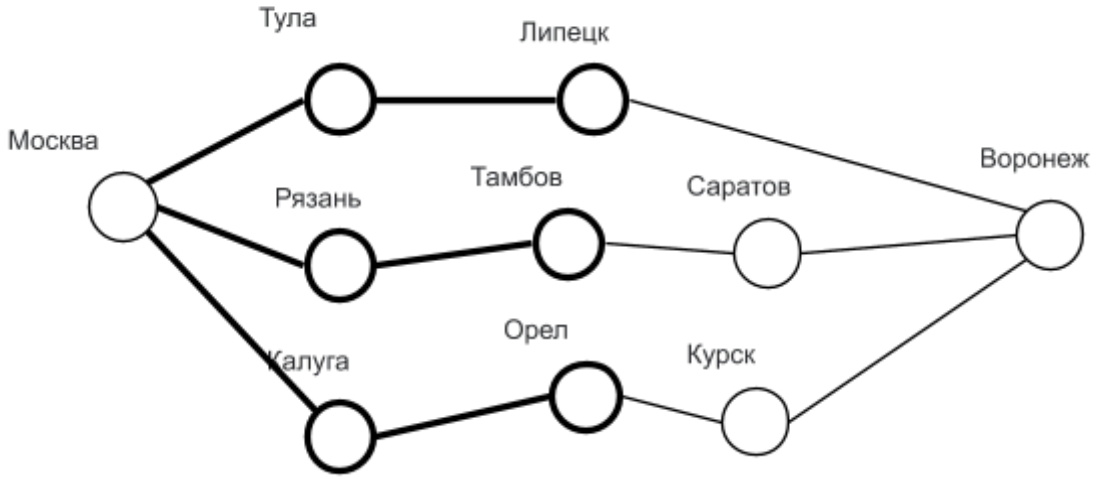
Представим (в рамках примера, а не реальной транспортной доступности), что нам необходимо добраться из Москвы в Воронеж. На карте обозначены дороги и города, через которые это можно сделать. Задача: доехать на автобусе до Воронежа с минимальным количеством пересадок.



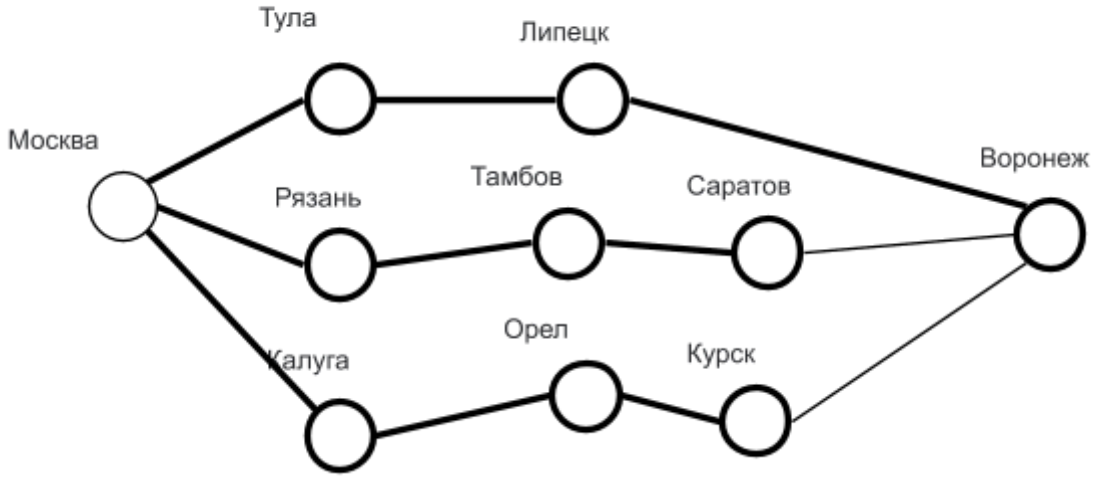
Попробуем сделать это за один шаг. На следующем рисунке выделены пути и города, до которых можно добраться из Москвы с одной пересадкой.



Это Тула, Рязань и Калуга. Попробуем доехать до Воронежа с двумя пересадками.



И снова до Воронежа не добрались. Попробуем третий шаг.



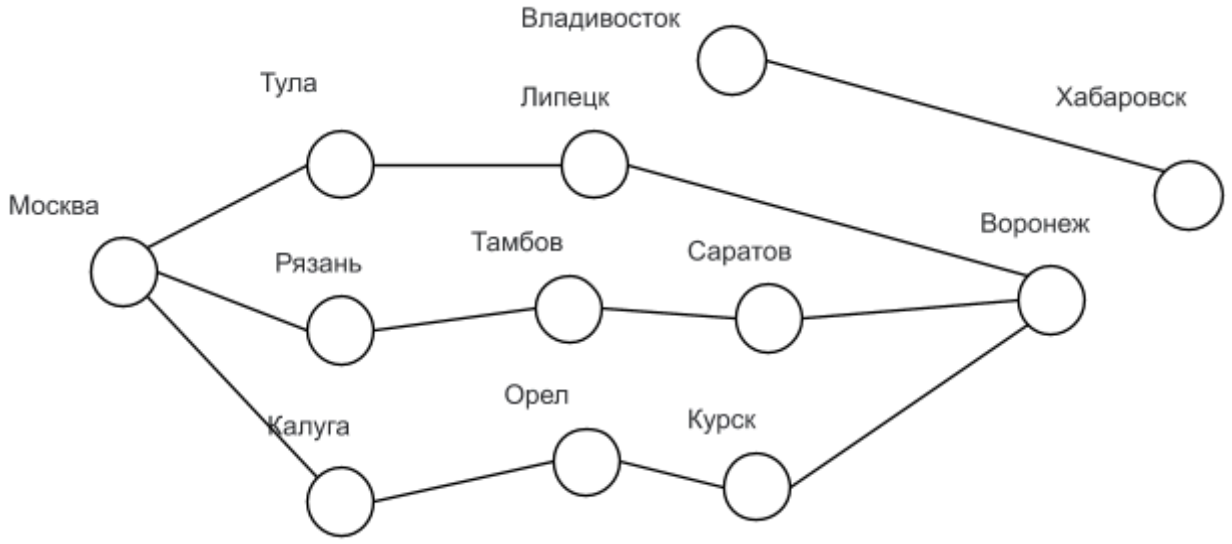
В нашем примере до Воронежа из Москвы можно добраться за три пересадки через Тулу и Липецк. Есть и другие маршруты, но они длиннее. Используя такой алгоритм — он называется поиском в ширину, — мы нашли самый короткий маршрут. Это задача нахождения кратчайшего пути.

Чтобы решить эту задачу, нам понадобилось смоделировать ее с помощью графа и выполнить поиск в ширину. Теперь немного теории о графах.

На рисунках выше, которые моделируют дорожную сеть, города, обозначенные кружками, — это вершины графа, а дороги, соединяющие их — ребра графа.

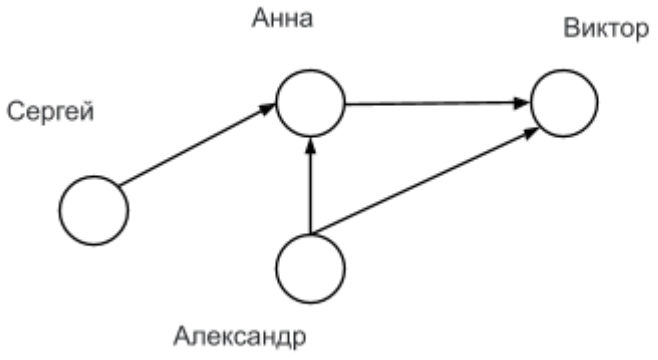
Если две вершины соединены ребром, они называются смежными. Москва и Тамбов — смежные вершины, а Москва и Воронеж — нет. Смежность — это понятие, которое обозначает отношения только между двумя вершинами. Если есть вершина «Москва» и она связана одним ребром с Тулой, другим — с Рязанью, а третьим — с Калугой, такие отношения называются соседними. Вершины — Рязань, Калуга и Тула — соседние для Москвы.

Расстояние от Москвы до Воронежа через другие города — это путь. Граф называется связным, если из каждой вершины можно добраться к любой другой, в ином случае — несвязным.



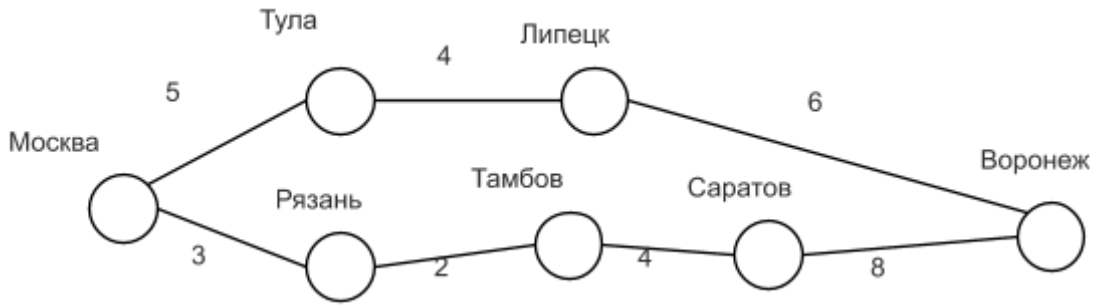
Пути из Москвы до Хабаровска на нашей модели дорожной сети нет. Поэтому граф, представленный на рисунке выше, считается несвязным.

Рассмотрим пример, в котором есть список друзей, которые должны друг другу деньги. Например, Сергей должен Анне, а Анна — Виктору, и есть еще Александр, который занимал и у Анны, и у Виктора. Такую схему можно изобразить с помощью графа.



Граф, в котором есть связи в одном направлении, называется направленным. На рисунке выше представлен направленный граф.

Вернемся к нашим дорогам и добавим на ребра графа обозначения расстояний между городами в условных единицах.



Такие условные единицы называются весами, а графы — взвешенными.

Реализация графа на Java

Вершины

На практике вершинами описываются реально существующие объекты. Если это города, которые объединены сетью дорог, то поля таких объектов могут содержать название города, плотность населения, температуру и другие характеристики. Для простоты мы будем обозначать вершины объектом **Vertex** с полем, в котором будет храниться символьная метка и флаг, необходимый для поисковых алгоритмов.

Определение класса вершины графа представлено в листинге ниже:

```
class Vertex{
    public char label;

    public boolean wasVisited;

    public Vertex(char label) {
        this.label = label;
        this.wasVisited = false;
    }

}
```

В качестве структуры данных, в которой будут храниться вершины, выберем массив. В таком случае обращаться к вершинам будем по индексу. Хранить вершины можно в списке или другой удобной структуре данных.

Ребра

Ребра — это связи между вершинами. Какую структуру данных выбрать для хранения этих взаимоотношений? В двоичных деревьях мы хранили ссылки на правого и левого потомка в объекте. В отличие от дерева, в графе каждая вершина может быть связана с любой другой, и количество взаимоотношений может быть **N** (по количеству вершин). Для моделирования такой структуры взаимосвязей обычно используют матрицу смежности или список смежности.

Матрица смежности

Матрица смежности представляет собой двумерный массив, в котором значения обозначают связи между вершинами. Если граф содержит **N** вершин, матрица смежности будет размером **NxN**.

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

В представленной матрице смежности единицы указывают на связи между вершинами. При отсутствии ребра значение элемента равно 0. Если посмотреть на диагональ, которая пересекает сверху вниз всю матрицу (слева направо), то значения в элементах, расположенных в ней, равны 0. Это означает, что не существует связи вершины самой с собой. Такие матрицы называются треугольными, так как верхняя часть до диагонали является зеркальным отражением нижней. Если в граф будет добавлено ребро, связывающее две вершины, то в матрице необходимо будет изменить два значения.

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

Список смежности

Отношения между вершинами можно хранить в списке смежности. Он может быть представлен в виде массива, элементы которого являются связными списками. Вместо массива можно использовать список. Каждый отдельный список содержит информацию о связях с одной вершиной.

Вершина	**Список смежности**
A	B->C
B	A->C->D
C	A->B->D
D	B->C

Создадим массив с названием **vertexList**, в который с помощью оператора **new** будем добавлять новую вершину в графе:

```
Vertex [] vertexList = new Vertex[32];
vertexList[0] = new Vertex('A');

vertexList[1] = new Vertex('B');

vertexList[2] = new Vertex('C');
```

Код, представленный выше, создает массив размером 32 и добавляет три вершины: «A», «B» и «C». Сейчас граф не имеет связей между вершинами. Чтобы добавить связь, необходимо сделать матрицу смежности **adjMat**. При создании значения матрицы заполняются нулями. Для добавления ребра между вершинами «A» и «C» запишем в матрицу следующие значения:

```
adjMat[0][2] = 1;

adjMat[2][0] = 1;
```

Создадим класс **Graph**, в котором реализуем методы для создания списка вершин и матрицы смежности:

```
class Vertex{
    public char label;

    public boolean wasVisited;

    public Vertex(char label) {
        this.label = label;
        this.wasVisited = false;
    }
}

class Graph{
    private final int MAX_VERTS = 32;
    private Vertex[] vertexList;
    private int[][] adjMat;
    private int size;

    public Graph(){
        vertexList = new Vertex[MAX_VERTS];
        adjMat = new int[MAX_VERTS][MAX_VERTS];
        size = 0;
        for (int i = 0; i < MAX_VERTS; i++) {
            for (int j = 0; j < MAX_VERTS; j++) {
                adjMat[i][j] = 0;
            }

        }
    }

    public void addVertex(char label){
        vertexList[size++] = new Vertex(label);
    }

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

    public void displayVertex(int vertex){
        System.out.println(vertexList[vertex]);
    }

}
```

Поле **MAX_VERTS** определяет максимальное количество вершин, которое может быть представлено в графе. **VertexList** — это массив, который хранит вершины, а **adjMat** — матрица смежности. В поле **size** будет храниться текущее количество вершин. При вызове конструктора инициализируется массив размером **MAX_VERTS**, матрица смежности размером **MAX_VERTS x MAX_VERTS**, значение текущего количества вершин. Также конструктор заполняет матрицу смежности нулями.

В классе **Graph** реализовано три метода:

- **addVertex** — добавление вершины в граф;
- **addEdge** — добавление ребра между вершинами;

- **displayVertex** — вывод в консоль наименования вершины.

Обход

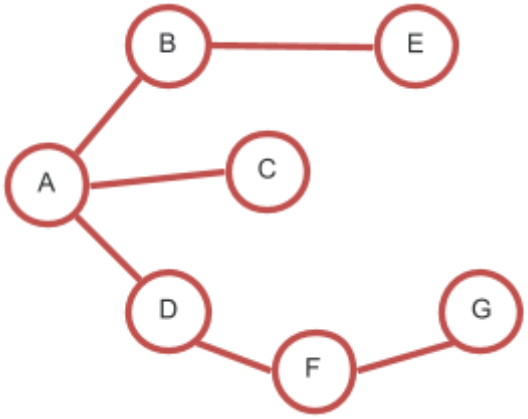
Обход — одна из основных операций, которая проводится над графами. Она состоит в посещении из заданной вершины всех других, которые связаны ребрами. Если представить граф железнодорожного сообщения, можно поставить задачу найти маршрут со всеми остановками между Москвой и Воронежем.

Существует два основных способа обхода деревьев — в глубину и в ширину. Оба подхода обеспечивают перебор всех связанных вершин, но обход в глубину использует в качестве структуры данных стек, а обход в ширину — очередь.

Обход в глубину

Для обхода в глубину каждую пройденную вершину будем помещать в стек, чтобы алгоритм мог вернуться назад в случае тупика.

Рассмотрим обход в глубину, используя граф, изображенный на рисунке:



Для обхода в глубину необходимо выбрать отправную точку. В нашем случае это вершина **A**. Алгоритм будет выглядеть следующим образом:

В начале в стек будет передана и помечена как посещенная вершина **A**. Потом смежная вершина **B**, с которой будут проделаны те же операции, далее будет переход к вершине **E**. У нее нет непосещенных смежных вершин, поэтому вершина **E** будет извлечена из стека. Далее возвращаемся к **B**. У нее тоже нет непосещенных смежных вершин, так что она тоже извлекается из стека. Далее переходим к вершине **A**, у которой еще есть непосещенные смежные вершины. Следующий переход — к вершине **C**. Она помещается в стек. Так как вершина **C** — тупиковая и не имеет смежных непосещенных соседей, она извлекается из стека. У вершины **A** осталась одна непосещенная смежная вершина — **D**. Переходим к ней и помещаем ее в стек. Далее с вершинами **F** и **G** повторяем то же действие, что и с **D**. Когда непосещенных смежных вершин не останется, из стека будут извлечены вершины **G**, **F**, **D** и **A**. Когда стек опустеет, обход будет завершен.

Для реализации обхода в глубину необходимо найти непосещенную смежную вершину. При помощи матрицы смежности алгоритм просматривает строку заданной вершины и отбирает столбцы со значением 1. По номеру столбца определяется номер смежной вершины. После производится проверка, посещалась вершина или нет.

Если вершина не проверялась, значит искомая вершина найдена, в ином случае смежных непосещенных вершин не осталось.

```
private int getAdjUnvisitedVertex(int ver) {
      for (int i = 0; i < size; i++) {
            if(adjMat[ver][i] == 1 && !vertexList[i].wasVisited) {
                  return i;
            }
      }

      return -1;

}
```

Теперь можно перейти к написанию метода **dfs**, который выполнит обход в глубину. В коде метода работает вышеописанный алгоритм: проверяется элемент на вершине стека, находятся непосещенные соседи; если их нет, элемент извлекается из стека, а если вершина найдена — алгоритм помещает ее в стек.

```
public void dfs(){
      vertexList[0].wasVisited = true;
      displayVertex(0);
      stack.push(0);
      while (!stack.isEmpty()) {
            int v = getAdjUnvisitedVertex(stack.peek());
            if (v == -1){
                  stack.pop();
            } else {
                  vertexList[v].wasVisited = true;
                  displayVertex(v);
                  stack.push(v);
            }
      }

      for (int i = 0; i < size; i++) {
            vertexList[i].wasVisited = false;
      }

}
```

Полный код программы обхода в глубину:

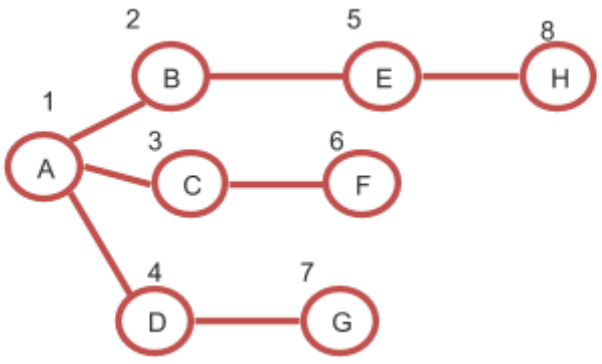
```
class Stack{
    private int maxSize;
    private int[] stackArr;
    private int top;

    public Stack(int size){
        this.maxSize = size;
        this.stackArr = new int[size];
        this.top = -1;
    }

    public void push(int i){
        stackArr[++top] = i;
    }

    public int pop(){
        return stackArr[top--];
    }

    public boolean isEmpty(){
        return (top == -1);
    }

    public int peek(){
        return stackArr[top];
    }

}
```

```
class Vertex{
    public char label;

    public boolean wasVisited;

    public Vertex(char label) {
        this.label = label;
        this.wasVisited = false;
    }
}

class Graph{
    private final int MAX_VERTS = 32;
    private Vertex[] vertexList;
    private int[][] adjMat;
    private int size;
    private Stack stack;

    public Graph(){
        stack = new Stack(MAX_VERTS);
        vertexList = new Vertex[MAX_VERTS];
        adjMat = new int[MAX_VERTS][MAX_VERTS];
        size = 0;
        for (int i = 0; i < MAX_VERTS; i++) {
            for (int j = 0; j < MAX_VERTS; j++) {
                adjMat[i][j] = 0;
            }

        }
    }

    private int getAdjUnvisitedVertex(int ver){
        for (int i = 0; i < size; i++) {
            if(adjMat[ver][i] == 1 && vertexList[i].wasVisited == false){
                return i;
            }
        }

        return -1;
    }

    public void dfs(){
        vertexList[0].wasVisited = true;
        displayVertex(0);
        stack.push(0);
        while (!stack.isEmpty()) {
            int v = getAdjUnvisitedVertex(stack.peek());
            if (v == -1){
                stack.pop();
            } else {
                vertexList[v].wasVisited = true;
                displayVertex(v);
                stack.push(v);
            }
        }

        for (int i = 0; i < size; i++) {
            vertexList[i].wasVisited = false;
        }
```

```
    }



    public void addVertex(char label){
        vertexList[size++] = new Vertex(label);
    }

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

    public void displayVertex(int vertex){
        System.out.println(vertexList[vertex].label);
    }
}

public class GraphApp{
    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addVertex('A');
        graph.addVertex('B');
        graph.addVertex('C');
        graph.addVertex('D');
        graph.addVertex('E');

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

        graph.dfs();
    }

}
```

Обход в ширину

В отличие от обхода в глубину, где алгоритм стремится как можно быстрее удалиться от вершины, обход в ширину, напротив, стремится быть как можно ближе к исходной вершине. Сначала он обходит все смежные вершины и только после этого отходит дальше.



Обход начинается с вершины **A**, алгоритм делает ее текущей. Далее посещается смежная непосещенная вершина и помещается в очередь. Когда смежные вершины заканчиваются, алгоритм извлекает следующую вершину из очереди, делает ее текущей и повторяется, пока очередь не станет пустой.

Для графа, изображенного на рисунке выше, первой вершиной становится **A**, далее в очередь помещаются вершины **B**, **C**, **D**. Когда смежных вершин для **A** не остается, из очереди извлекается **B**, и алгоритм ищет непосещенные вершины, смежные с ней. Алгоритм находит вершину **E** и помещает ее в очередь. Так как вершин, смежных с **B**, не осталось, алгоритм извлекает из очереди следующую вершину — **C**. И так далее, пока очередь не опустеет.

```
public void bfs(){
      vertexList[0].wasVisited = true;
      displayVertex(0);
      queue.insert(0); // Вставка в конец очереди
      int v2;
      while(!queue.isEmpty()){
            int v1 = queue.remove();
            while((v2=getAdjUnvisitedVertex(v1)) != -1){
                  vertexList[v2].wasVisited = true; // Пометка
                  displayVertex(v2); // Вывод
                  queue.insert(v2);
            }
      }
      for(int i=0; i<size; i++) // Сброс флагов
            vertexList[i].wasVisited = false;

}
```

Обход в ширину сначала находит все вершины на расстоянии одного ребра от начальной, затем все на расстоянии двух ребер и так далее. Такой алгоритм может пригодиться для задач нахождения кратчайшего пути.

Практическое задание

1. Реализовать программу, в которой задается граф из 10 вершин. Задать ребра и найти кратчайший путь с помощью поиска в ширину.

Дополнительные материалы

1. Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 622–664 сс.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафоре Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 574–620 сс.