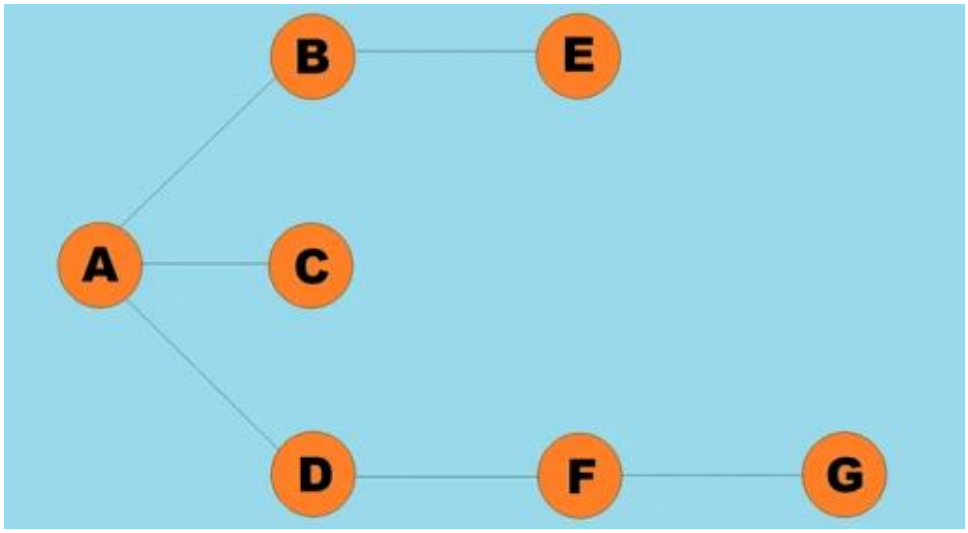
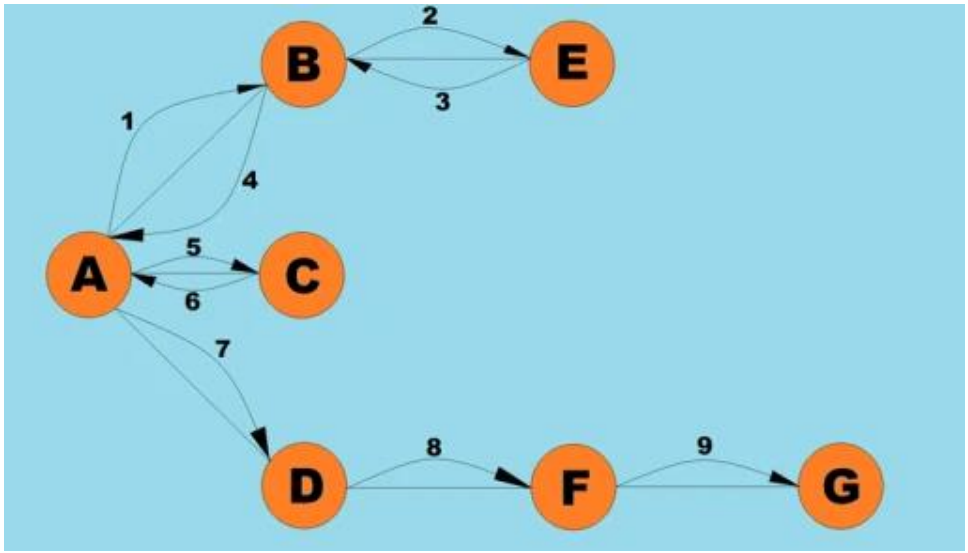
Алгоритмы поиска пути (глубина, ширина)
Одна из основных операций, которые выполняются с графами, — это определение всех вершин, достижимых от заданной вершины. Представьте, что вы пытаетесь определить, как можно добраться от одного города в другой с возможными пересадками. К одним городам можно добраться напрямую, к другим нужно ехать в обход через другие города. Существует много других ситуаций, в которых может понадобиться нахождение всех вершин, к которым можно найти путь от заданной вершины. Так вот, существует два основных способа обхода графов: обход в глубину и обход в ширину, которые мы и рассмотрим. Оба способа обеспечат перебор всех связных вершин. Для дальнейшего рассмотрения алгоритмов в глубину и ширину возьмем следующий граф:



Обход в глубину
Это один из наиболее распространенных методов обхода графа. Данная стратегия поиска в глубину состоит в том, чтобы идти «вглубь» графа насколько это возможно, а достигнув тупика, возвращаться к ближайшей вершине, у которой есть смежные ранее не посещенные вершины. Этот алгоритм хранит в стеке информацию о том, куда следует вернуться при достижении “тупика”. Правила обхода в глубину:

1. Посетить смежную, ранее не посещенную вершину, пометить её и занести в стек.
2. Перейти на данную вершину.
3. Повторить этап 1.
4. Если выполнение пункта 1 невозможно, вернуться к предыдущей вершины и попытаться повторить правило 1. Если это невозможно — вернуться к вершине до нее, и так далее, пока не найдем вершину, с которой можно продолжить обход.
5. Продолжать до тех пор, пока все вершины не окажутся в стеке.




```
public class Graph {

  private final int MAX_VERTS = 10;
  private Vertex vertexArray[]; //массив вершин
  private int adjMat[][]; // матрица смежности
  private int nVerts; // текущее количество вершин
  private Stack<integer> stack;

  public Graph() { // инициализация внутрених полей
    vertexArray = new Vertex[MAX_VERTS];
    adjMat = new int[MAX_VERTS][MAX_VERTS];
    nVerts = 0;
    for (int j = 0; j < MAX_VERTS; j++) {
      for (int k = 0; k < MAX_VERTS; k++) {
        adjMat[j][k] = 0;
      }
    }
    stack = new Stack<>();
  }

  public void addVertex(char lab) {
    vertexArray[nVerts++] = new Vertex(lab);
  }

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

  public void displayVertex(int v) {
    System.out.println(vertexArray[v].getLabel());
  }
```

```
  public void dfs() { // обход в глубину
    vertexArray[0].setWasVisited(true); // берётся первая вершина
    displayVertex(0);
    stack.push(0);

    while (!stack.empty()) {
      int v = getAdjUnvisitedVertex(stack.peek()); // вынуть индекс смежной веришины,
если есть 1, нету -1
      if (v == -1) { // если непройденных смежных вершин нету
        stack.pop(); // элемент извлекается из стека
      }
      else {
        vertexArray[v].setWasVisited(true);
        displayVertex(v);
        stack.push(v); // элемент попадает на вершину стека
      }
    }

    for (int j = 0; j < nVerts; j++) {  // сброс флагов
      vertexArray[j].wasVisited = false;
    }

  }

  private int getAdjUnvisitedVertex(int v) {
    for (int j = 0; j < nVerts; j++) {
      if (adjMat[v][j] == 1 && vertexArray[j].wasVisited == false) {
        return j; //возвращает первую найденную вершину
      }
    }
    return -1;
  }
}
</integer>
```

Вершина имеет вид:

```
public class Vertex {
  private char label;  // метка A например
  public boolean wasVisited;

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

  public char getLabel() {
    return this.label;
  }
```

```
  public boolean isWasVisited() {
    return this.wasVisited;
  }

  public void setWasVisited(final boolean wasVisited) {
    this.wasVisited = wasVisited;
  }
} Запустим данный алгоритм с конкретными вершинами и посмотрим на корректность работы: public class Solution {
  public static void main(String[] args) {
    Graph graph = new Graph();
    graph.addVertex('A'); //0
    graph.addVertex('B'); //1
    graph.addVertex('C'); //2
    graph.addVertex('D'); //3
    graph.addVertex('E'); //4
    graph.addVertex('F'); //5
    graph.addVertex('G'); //6

    graph.addEdge(0,1);
    graph.addEdge(0,2);
    graph.addEdge(0,3);
    graph.addEdge(1,4);
    graph.addEdge(3,5);
    graph.addEdge(5,6);

    System.out.println("Visits: ");
    graph.dfs();
  }
} Вывод в консоли:
```

Visits: A B E C D F G

Так как у нас есть матрица смежности и в методе прохода мы используем цикл, вложенный в цикл, то временная сложность будет $O(N^2)$.

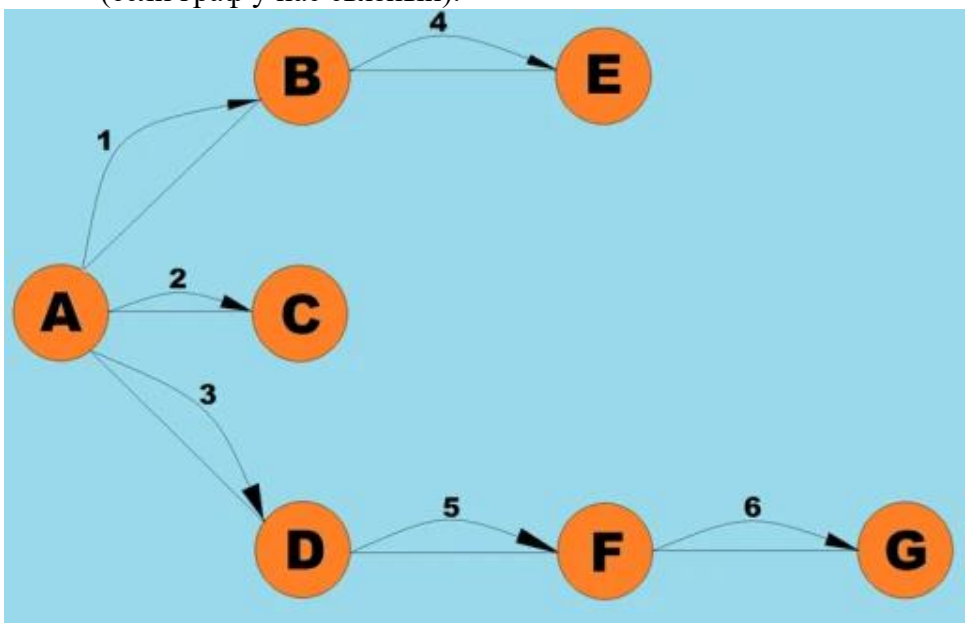
Обход в ширину

Данный алгоритм, как и обход в глубину, является одним наиболее простых и базовых методов обхода графа. Его суть в том, что у нас есть некоторая текущая вершина, с которой мы все смежные, непройденные вершины, заносим в очередь и выбираем следующий элемент (который хранится первым в очереди), чтобы его сделать текущим

Если разбить данный алгоритм на этапы, можно выделить следующие правила:

1. Посетить следующую, ранее не посещенную вершину, смежную с текущей вершиной, пометить её заранее и занести в очередь.

2. Если выполнение правила #1 невозможно — извлечь вершину из очереди и сделать её текущей вершиной.
3. Если правило #1 и #2 невозможно, обход закончен, а все вершины пройдены (если граф у нас связный).



Класс графа практически идентичен аналогичному классу из алгоритма поиска в глубину, за исключением метода, обрабатывающего алгоритм и замены внутреннего стека на очередь: **public class** Graph {

```
private final int MAX_VERTS = 10;
private Vertex vertexList[]; //массив вершин
private int adjMat[][]; // матрица смежности
private int nVerts; // текущее количество вершин
private Queue<integer> queue;

public Graph() {
  vertexList = new Vertex[MAX_VERTS];
  adjMat = new int[MAX_VERTS][MAX_VERTS];
  nVerts = 0;
  for (int j = 0; j < MAX_VERTS; j++) {
    for (int k = 0; k < MAX_VERTS; k++) {  // заполнение матрицы смежности нулями
      adjMat[j][k] = 0;
    }
  }
  queue = new PriorityQueue<>();
}

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

public void addEdge(int start, int end) {
```

```
    adjMat[start][end] = 1;
    adjMat[end][start] = 1;
  }

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

  public void bfc() { // обход в глубину
    vertexList[0].setWasVisited(true);
    displayVertex(0);
    queue.add(0);
    int v2;

    while (!queue.isEmpty()) {
      int v = queue.remove();

      while((v2 = getAdjUnvisitedVertex(v))!=-1) {// цикл будет работать, пока все
смежные вершины не будут найденны, и не будут добавлены в очередь
        vertexList[v2].wasVisited =true;
        displayVertex(v2);
        queue.add(v2);
      }
    }

    for (int j = 0; j < nVerts; j++) {  // сброс флагов
      vertexList[j].wasVisited = false;
    }

  }

  private int getAdjUnvisitedVertex(int v) {
    for (int j = 0; j < nVerts; j++) {
      if (adjMat[v][j] == 1 && vertexList[j].wasVisited == false) {
        return j; //возвращает первую найденную вершину
      }
    }
    return -1;
  }
}
</integer>
```

Класс Vertex идентичен классу из алгоритма про поиск в глубину. Давайте приведем данный алгоритм в действие: **public class** Solution {

```
  public static void main(String[] args) {
    Graph graph = new Graph();
    graph.addVertex('A'); //0
```

```
    graph.addVertex('B'); //1
    graph.addVertex('C'); //2
    graph.addVertex('D'); //3
    graph.addVertex('E'); //4
    graph.addVertex('F'); //5
    graph.addVertex('G'); //6

    graph.addEdge(0,1);
    graph.addEdge(0,2);
    graph.addEdge(0,3);
    graph.addEdge(1,4);
    graph.addEdge(3,5);
    graph.addEdge(5,6);

    System.out.println("Visits: ");
    graph.bfc();
  }
```
} Вывод в консоль:

Visits: A B C D E F G

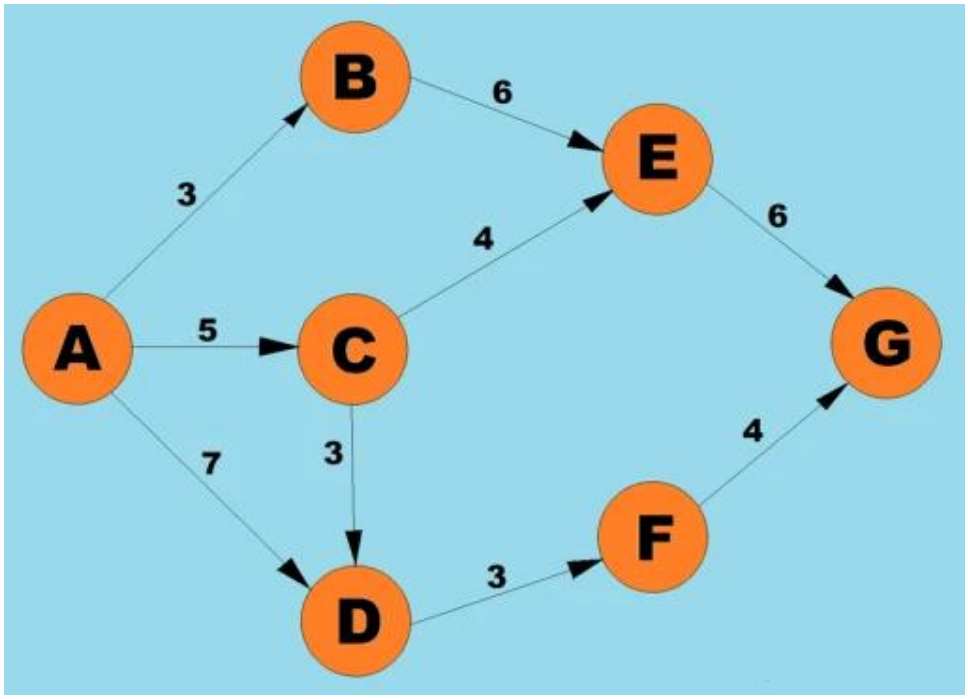
Опять же: мы имеем матрицу смежности и используем цикл, вложенный в цикл, поэтому $O(N^2)$ — временная сложность приведенного алгоритма.

Алгоритм Дейкстры.

Этапы алгоритма Дейкстры:

- Этап 1: поиск узла, переход к которому будет составлять наименьшую стоимость. Вы стоите в самом начале и думаете, куда направиться: к узлу A или к узлу B. Сколько времени понадобится, чтобы добраться до каждого из этих узлов?
- Этап 2: вычисление, сколько времени нужно, чтобы добраться до всех ещё не затронутых алгоритмом соседей B при переходе по ребру из B. Если же это новое время окажется меньше старого, путь через ребро B и станет новым кратчайшим путём для этой вершины.
- Этап 3: помечаем вершину B как пройденную.
- Этап 4: перейти к этапу 1.

Цикл этих этапов мы будем повторять до тех пор, пока все вершины не будут пройдены. Давайте рассмотрим следующий взвешенный направленный граф:



Итак, с помощью вышеприведенного алгоритма мы определим кратчайший путь от A до G:

1. Для вершины A есть три возможных пути: к B с весом 3, к C с весом 5 и к D с весом 7. Согласно первому пункту алгоритма выбираем узел с наименьшей стоимостью перехода — то есть к B.
2. Так как единственной непройденной вершиной-соседом для B будет вершина E, мы проверяем, каков будет путь при прохождении через эту вершину. 3(AB) + 6(BE) = 9.

Таким образом, мы отмечаем что текущий кратчайший путь до AE = 9.

3. Так как наша работа с вершиной B уже закончена, переходим к выбору следующей вершины с минимальным весом ребра до неё.

С вершин A и B это могут быть вершины D(7), C(5), E(6).
Наименьший вес ребра имеет C, поэтому к этой вершине и перейдем.

4. Далее, как и раньше, выясняем кратчайший путь к соседним вершинам при проходе через C:
    - AD = 5(AC) + 3(CD) = 8, но так как предыдущий кратчайший путь AC = 7, то есть меньше, чем этот через C, мы так и оставляем кратчайший путь AD = 7, без изменений.
    - CE = 5(AC) + 4(CE) = 9, этот новый кратчайший путь равен прежнему поэтому мы оставляем его без изменения тоже.
5. Из ближайших доступных вершин, E и D, выбираем вершину с наименьшим весом ребра, то есть D(3).
6. Выясняем кратчайший путь до его соседа — F.

AF = 7(AD) + 3(DF) = 10

7. Из ближайших доступных вершин E и F, выбираем вершину с наименьшим весом ребра к ней, то есть F(3).
8. Выясняем кратчайший путь до его соседа — G.

AG = 7(AD) + 3(DF) + 4(FG) = 14

Собственно, вот мы и нашли путь от A до G.
Но чтобы убедиться в том, что он кратчайший, мы должны прогнать наши этапы и для вершины E.

9. Так как у вершины G нет соседних вершин, к которым вел бы направленный путь, у нас остаётся только вершина E: выбираем ее.
10. Выясняем кратчайший путь до соседа — G.

AG = 3(AB) + 6(BE) + 6(EG) = 15, этот путь длиннее прежнего кратчайшего AG(14), поэтому данный путь мы оставляем без изменений.
Так как вершин, ведущих от G, нет, прогонять этапы для данной вершины не имеет смысла. Поэтому работу алгоритма можно считать законченной.
Как я и говорил ранее, помимо выяснения кратчайшего пути для AG, мы получили кратчайшие пути до всех вершин от вершины A (AB, AC, AD, AE, AF). Что ж, пришло время взглянуть, каким образом это возможно реализовать на Java. Для начала рассмотрим класс вершины: **public class** Vertex {

```
  private char label;
  private boolean isInTree;

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

  public char getLabel() {
     return label;
  }

  public void setLabel(char label) {
     this.label = label;
  }

  public boolean isInTree() {
     return isInTree;
  }

  public void setInTree(boolean inTree) {
     isInTree = inTree;
  }
```

} Класс вершины фактически идентичен классу вершины из поиска в глубину и ширину. Чтобы отобразить кратчайшие пути, нам понадобится новый класс, который будет содержать необходимые нам данные: **public class** Path { // объект данного класса содержащий расстояние и предыдущие и пройденные вершины

```
  private int distance; // текущая дистанция от начальной вершины
  private List<integer> parentVertices; // текущий родитель вершины

  public Path(int distance) {
     this.distance = distance;
```

```
        this.parentVertices = new ArrayList<>();
    }

    public int getDistance() {
        return distance;
    }

    public void setDistance(int distance) {
        this.distance = distance;
    }

    public List<integer> getParentVertices() {
        return parentVertices;
    }

    public void setParentVertices(List<integer> parentVertices) {
        this.parentVertices = parentVertices;
    }
}
</integer></integer></integer>
```

В данном классе мы можем видеть общую дистанцию пути и вершины, которые будут пройдены при проходе по найкрайчайшему пути. А сейчас хотелось бы рассмотреть класс в котором, собственно, и происходит кратчайший обход графа. Итак, класс графа: **public class** Graph {

```
    private final int MAX_VERTS = 10;// максимальное количество вершин
    private final int INFINITY = 100000000; // это число у нас будет служить в качестве
"бесконечности"
    private Vertex vertexList[]; // список вершин
    private int relationMatrix[][]; // матрица связей вершин
    private int countOfVertices; // текущее количество вершин
    private int countOfVertexInTree; // количество рассмотренных вершин в дереве
    private List<path> shortestPaths; // список данных кратчайших путей
    private int currentVertex; // текущая вершина
    private int startToCurrent; //расстояние до currentVertex

    public Graph() {
        vertexList = new Vertex[MAX_VERTS]; // матрица смежности
        relationMatrix = new int[MAX_VERTS][MAX_VERTS];
        countOfVertices = 0;
        countOfVertexInTree = 0;
        for (int i = 0; i < MAX_VERTS; i++) {// матрица смежности заполняется
            for (int k = 0; k < MAX_VERTS; k++) { // бесконечными расстояниями
                relationMatrix[i][k] = INFINITY; // задания значений по умолчанию
                shortestPaths = new ArrayList<>();// задается пустым
            }
```

```
    }
  }

  public void addVertex(char lab) {// задание новых вершин
    vertexList[countOfVertices++] = new Vertex(lab);
  }

  public void addEdge(int start, int end, int weight) {
    relationMatrix[start][end] = weight; // задание ребер между вершинами, с весом
между ними
  }

  public void path() { // выбор кратчайшего пути
    //  задание данных для стартовой вершины
    int startTree = 0; // стартуем с вершины 0
    vertexList[startTree].setInTree(true); // включение в состав дерева первого элемента
    countOfVertexInTree = 1;

    // заполнение коротких путей для вершин смежных с стартовой
    for (int i = 0; i < countOfVertices; i++) {
      int tempDist = relationMatrix[startTree][i];
      Path path = new Path(tempDist);
      path.getParentVertices().add(0);// первым родительским элементом, будет всегда
стартовая вершина
      shortestPaths.add(path);
    }
    // пока все вершины не окажутся в дереве
    while (countOfVertexInTree < countOfVertices) { // выполняем, пока количество
вершин в дереве не сравняется с общим количеством вершин
      int indexMin = getMin();//получаем индекс вершины с наименьшей дистанцией, из
вершин еще не входящих в дерево
      int minDist = shortestPaths.get(indexMin).getDistance();// минимальная дистанция
вершины, из тек которые ещё не в дереве

      if (minDist == INFINITY) {
        System.out.println("В графе пристувствуют недостижимые вершины");
        break;// в случае если остались непройденными только недостижимые
вершины, мы выходим из цикла
      } else {
        currentVertex = indexMin; // переводим указатель currentVert к текущей
вершине
        startToCurrent = shortestPaths.get(indexMin).getDistance();// задаем дистанцию к
текущей вершине
      }

      vertexList[currentVertex].setInTree(true);  //включение текущей вершины в дерево
      countOfVertexInTree++; // увеличиваем счетчик вершин в дереве
```

```
            updateShortestPaths(); // обновление списка кратчайших путей
        }

        displayPaths(); // выводим в консоль результаты
    }

    public void clean() { // очиска дерева
        countOfVertexInTree = 0;
        for (int i = 0; i < countOfVertices; i++) {
            vertexList[i].setInTree(false);
        }
    }

    private int getMin() {
        int minDist = INFINITY; // за точку старта взята "бесконечная" длина
        int indexMin = 0;
        for (int i = 1; i < countOfVertices; i++) {// для каждой вершины
            if (!vertexList[i].isInTree() && shortestPaths.get(i).getDistance() < minDist) { // если
вершина ещё не ве дереве и её растояние меньше старого минимума
                minDist = shortestPaths.get(i).getDistance(); // задаётся новый минимум
                indexMin = i; // обновление индекса вершины содержащую минимаьную
дистанцию
            }
        }
        return indexMin; //возвращает индекс вершины с наименшей дистанцией, из
вершин еще не входящих в дерево
    }

    private void updateShortestPaths() {
        int vertexIndex = 1; // стартовая вершина пропускается
        while (vertexIndex < countOfVertices) { // перебор столбцов

            if (vertexList[vertexIndex].isInTree()) { // если вершина column уже включена в
дерево, она пропускается
                vertexIndex++;
                continue;
            }
            // вычисление расстояния для одного элемента sPath
            // получение ребра от currentVert к column
            int currentToFringe = relationMatrix[currentVertex][vertexIndex];
            // суммирование всех расстояний
            int startToFringe = startToCurrent + currentToFringe;
            // определение расстояния текущего элемента vertexIndex
            int shortPathDistance = shortestPaths.get(vertexIndex).getDistance();

            // сравнение расстояния через currentVertex с текущим расстоянием в вершине с
индексом vertexIndex
```

```
        if (startToFringe < shortPathDistance) {// если меньше, то у вершины под индексом vertexIndex будет задан новый кратчайший путь
            List<integer> newParents = new ArrayList<>(shortestPaths.get(currentVertex).getParentVertices());//создаём копию списка родителей вершины currentVert
            newParents.add(currentVertex);// задаём в него и currentVertex как предыдущий
            shortestPaths.get(vertexIndex).setParentVertices(newParents); // соохраняем новый маршут
            shortestPaths.get(vertexIndex).setDistance(startToFringe); // соохраняем новую дистанцию
        }
        vertexIndex++;
    }
  }

  private void displayPaths() { // метод для вывода кратчайших путей на экран
    for (int i = 0; i < countOfVertices; i++) {
      System.out.print(vertexList[i].getLabel() + " = ");
      if (shortestPaths.get(i).getDistance() == INFINITY) {
        System.out.println("0");
      } else {
        String result = shortestPaths.get(i).getDistance() + " (";
        List<integer> parents = shortestPaths.get(i).getParentVertices();
        for (int j = 0; j < parents.size(); j++) {
          result += vertexList[parents.get(j)].getLabel() + " -> ";
        }
        System.out.println(result + vertexList[i].getLabel() + ")");
      }
    }
  }

public class Solution { 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.addVertex('F');

graph.addVertex('G');

graph.addEdge(0, 1, 3); graph.addEdge(0, 2, 5); graph.addEdge(0, 3, 7); graph.addEdge(1, 4, 6); graph.addEdge(2, 4, 4); graph.addEdge(2, 3, 3); graph.addEdge(3, 5, 3); graph.addEdge(4, 6, 6); graph.addEdge(5, 6, 4);
```

System.out.println("Элементы имеют кратчайшие пути из точки A: ");

graph.path();

graph.clean(); } }

$O(N^2)$, так как у нас есть циклы, вложенные в цикл.