Tallinna Tehnikaülikool

Infotehnoloogia teaduskond

Algoritmid ja andmestruktuurid

Individuaaltöö

Koostaja: Mihhail Skripnik

Juhendaja: Jaanus Pöial

Tallinn 2020

1. Introduction

In computer science, a graph is an abstract data type that is meant to implement the undirected graph and directed graph concepts from the field of graph theory within mathematics.

A graph data structure consists of a finite (and possibly mutable) set of vertices (also called nodes or points), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges (also called links or lines), and for a directed graph are also known as arrows. The vertices may be part of the graph structure or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.). [1]

2. Solution of the problem

2.1 Description of the problem

Imagine that we have a graph of geographic points, where each vertex of the graph has an altitude parameter that displays the height of the point above sea level

My task is to write an algorithm for finding a path in the graph, where the maximum value of the height parameter at the highest point of the route will be as low as possible. According to the assignment, my algorithm should be based on the breadth-first search method. If there is no direct path from the start point to the destination, the heights of the start and end points are ignored when analyzing the highest point of the route.

This algorithm, after some changes, can be useful in the field of cargo transportation in mountainous countries (Norway, Switzerland, Chile, etc.), where height differences can significantly affect the fuel consumption of freight transport.

2.2 Breadth-first search algorithm

The principle of the Breadth First Search algorithm is quite simple, it can be divided into 3 steps:

1. The starting vertex in the graph is selected and placed in the queue.

2. All the vertices of the graph that have connection with the current vertex are placed in the queue, after which this vertex is checked for equality with the destination vertex and if it does not match, it is excluded from the queue.

3. Step 2 is performed for all vertices in the queue.

2.3 Algorithm for solving this problem

The very first the preparatory method comes into operation, which checks whether the given vertices exist in the given graph. Then the main algorithm starts its work. I have divided his activities into 3 steps:

STEP 1:

Breadth First Search algorithm (BFS) begins traversing the graph looking for a path to its destination. In most cases, the number of such "passes" is more than one.

After the first pass through the graph, the algorithm starts taking into account the "threshold point height" parameter, which forces it to exclude from the list points whose height parameter value is greater than or equal to the previous highest point in the path. Thus, each subsequent path, if found, will have a maximum point lower than the previous one.

STEP 2:

After the BFS algorithm has established that the path from point A to point B is possible, all the passed points (which contain our path) go through the so-called "cleaning". It works like this:

The algorithm goes through the entire list from end to beginning and checks each previous point for a connection with the current point. If connection between these two points is not found, then this previous point is eliminated from the storage of route points.

If there is a connection, then the previous point becomes the current one and so on until the last point in the list. As a result, we have an array of points connected to each other, from start to destination.

STEP 3:

The algorithm saves the found path and goes through its points, determining the highest point of the route, which will be the new threshold value for the height for future paths, if they exist.

REPEAT UNTIL…:

These iterations continue until the last possible path is found. After that, the algorithm checks the highest points of each found path and, based on this, returns the most optimal route to the user, where the highest point is the lowest possible.

Post scriptum:

If after the first iteration of the BFS the algorithm does not find any paths, then the execution of the program stops, because it means that the starting point and the destination point are not connected by any road.

2.4 Used data structures

Throughout the algorithm work, I use only ArrayList to store vertices, arcs, and resulting paths. This data type works quickly, makes it easy to get the required values by index, add and remove data from storage. This makes it the most suitable in my case.

The BFS algorithm of path finding possibility uses a LinkedList inherited from the Queue interface based on a queue (FIFO, first in, first out) adding elements to the end of the list and taking from the end if it’s necessary.

3. User program guide

3.1 How to get algorithm result

To get the result, everything you need is already prepared in the run() method. You can change the size of the dataset; the maximum values are also indicated in the comment in the code.

All you need to do to get the result is to specify the names of the start and destination vertices. The names of vertices by default start with the small letter “v” and then indicate the vertex number. Just run the main() method, that will activate our run() method.

The program after processing will return the most suitable path with an object of the Path class, by default I have set the display of the path arcs and vertices, and some useful information about the path is also indicated in the console. Path class contains two ArrayList (vertices and arcs), integer value that indicates the highest point of path and integer value of the unique path id. This class also contains two methods: toString() outputs path information using vertices, printArcPath() using arcs.

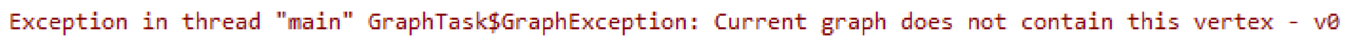
Below is the method to use to get the result:

```java
/** Actual main method to run examples and everything. */
public void run() {
   Graph g = new Graph (“A”); // enter your Graph name
   g.createRandomSimpleGraph (10, 45); // max (2500, 3123475) generate graph values
   System.out.println(g.toString()); // it shows you all vertices and their arcs in the graph
   Path optimizedPath = getOptimizedPath(g, “v1”, “v2”); // put your graph, start and destination
                                                                                            values
   System.out.println(“\nArc representation of best path:\n” +
   optimizedPath.printArcPath());
   System.out.println(“\nVertex representation of best path:\n” + optimizedPath.toString());
}
```

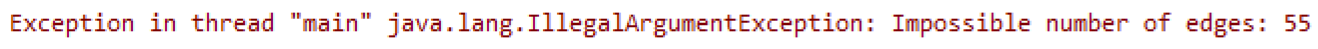
3.2 Possible errors when starting the program

During program operation, errors may occur if the vertices or dataset sizes are entered incorrectly. Errors are presented by classes of IllegalArgumentException, GraphPathException and GraphException, which contain the cause of the error and some information that can help in fixing the problem.

Below are a couple of examples of possible errors:



```
Exception in thread "main" GraphTask$GraphException: Current graph does not contain this vertex - v0
```



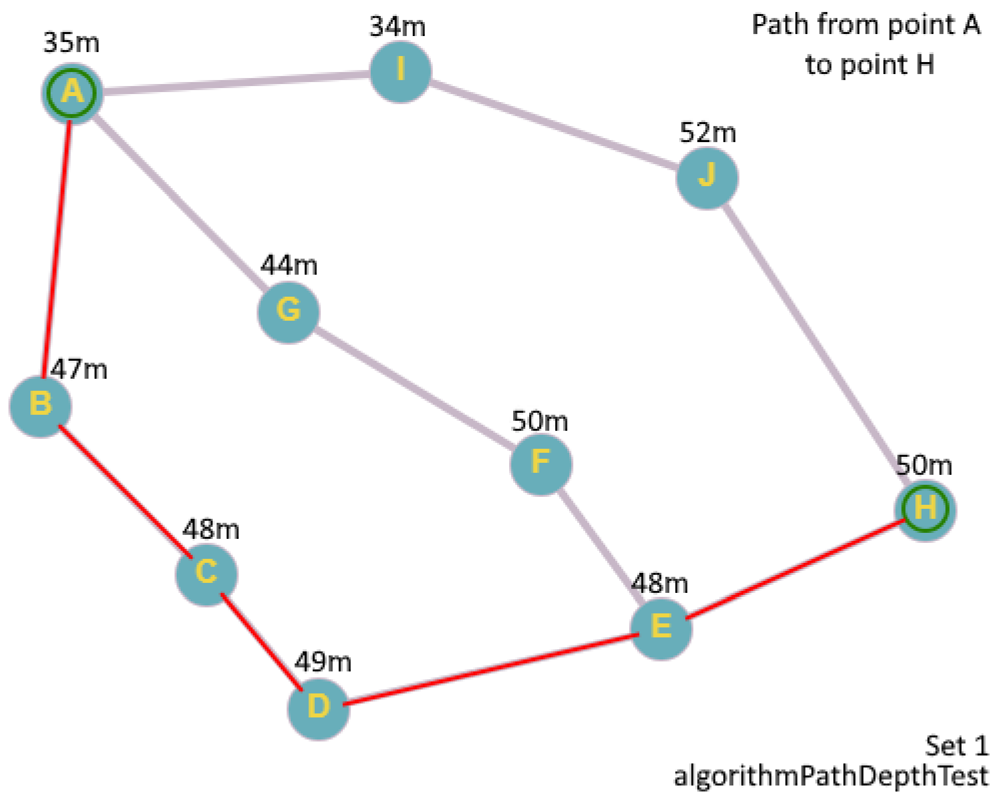
```
Exception in thread "main" java.lang.IllegalArgumentException: Impossible number of edges: 55
```

4. Program testing

The test code is presented in the next paragraph.

4.1 Test with small dataset v1

In order to check the correctness of my algorithm, I prepared a simple graph with 3 possible paths. 2 of 3 paths have starting points lower in height than the most optimal path, which is a good test of the algorithm for how it looks "in depth".



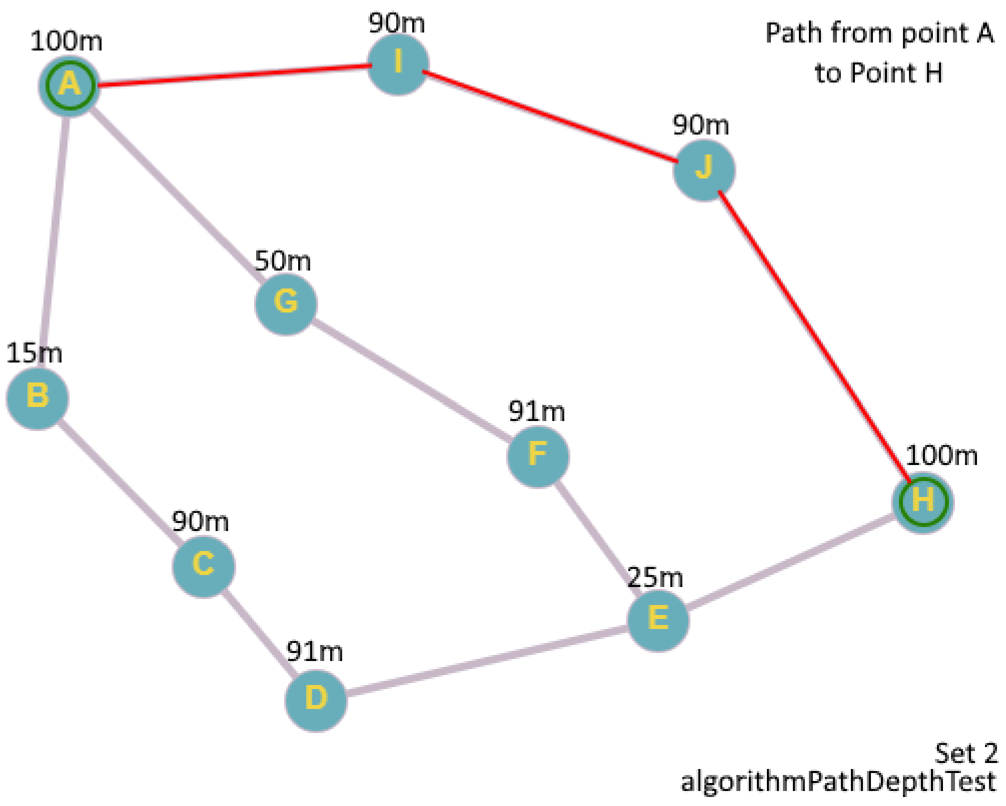
Result output:

```
Path from A to H
Paths found: 1
1: (A: 35m) ==> (B: 47m) ==> (C: 48m) ==> (D: 49m) ==> (E: 48m) ==> (H: 50m)
Highest point of path: 49m [Not included start and destination points if vertices more than 2]
Points in the route: 6

1: A---B : B---C : C---D : D---E : E---H
Elapsed time: 28 ms
```

4.2 Test with small dataset v2

Test similar to the previous one, but with a modified dataset.



Result output:

```
Path from A to H
Paths found: 2
Elapsed time: 31 ms
2: (A: 100m) ==> (I: 90m) ==> (J: 90m) ==> (H: 100m)
Highest point of path: 90m [Not included start and destination points if vertices more than 2]
Points in the route: 4

2: A---I : I---J : J---H
```

4.3 Behavior of the algorithm when points are not connected

A simple test when two points exist, but they have no connection. The method should return a GraphPathException. The dataset used in this test is similar to the previous one. Starting point is “I” vertex, destination is “B” vertex.

4.4 Test with the same vertex's height

Test with values of all vertices equal to zero. The number of paths is expected to be 1 in all cases, and the returned path is the first possible path. Thus, this test verifies the path filtering according to the principle of the "maximum height" of the previous path.

Result output:

```
Path from A to H
Paths found: 1
Elapsed time: 25 ms
1: (A: 0m) ==> (B: 0m) ==> (C: 0m) ==> (D: 0m) ==> (E: 0m) ==> (H: 0m)
Highest point of path: 0m [Not included start and destination points if vertices more than 2]
Points in the route: 6

1: A---B : B---C : C---D : D---E : E---H
```

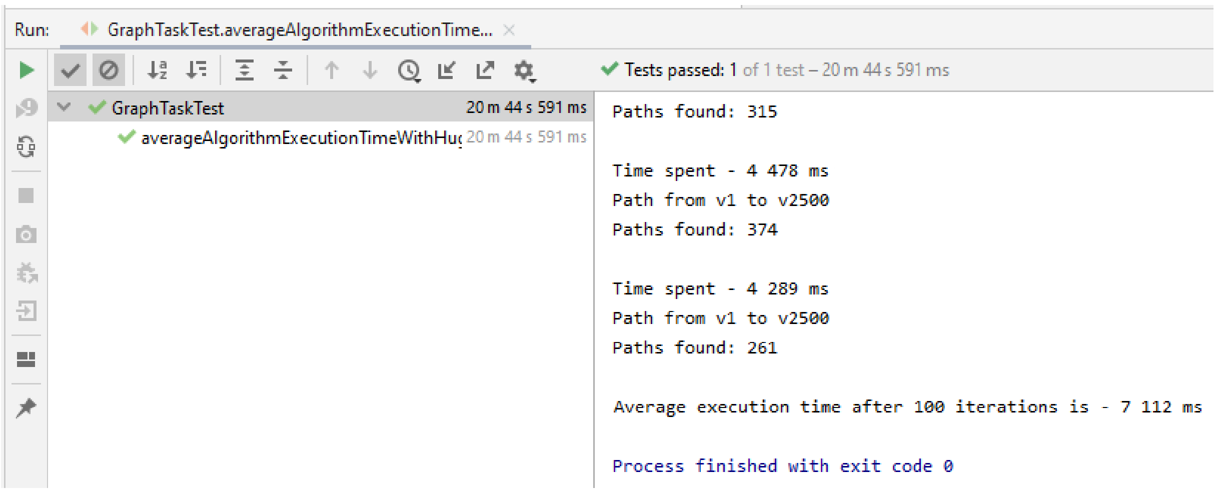
4.5 Algorithm time execution

I was interested in how fast and optimized my algorithm was as in result, and therefore I conducted a small test by doing 100 iterations to find a path from point A to point B with the largest possible dataset (2500 vertices and 3123475 connections).

The result was ambiguous. The average execution time of my program was 7.112 seconds, which, in general, the result is not bad, but on closer examination, I found that the variation in execution time is very high. it ranged from 30-40 milliseconds to 25-27 seconds in the worst cases. I can conclude that the result strongly depends on the values of the vertices and the list of arcs at the vertices. Considering that in the worst case, the algorithm can traverse the graph up to 2500 times, without considering the passage of the obtained points to obtain the exact path.

Also, since computer parameters can affect the performance of the program, I also indicate the processor of the computer on which the tests were run: Intel Core i7-8750H 2.20GHz

Dataset generation time is not included during program execution.


Run:
GraphTaskTest.averageAlgorithmExecutionTime...
Tests passed: 1 of 1 test – 20 m 44 s 591 ms
GraphTaskTest 20 m 44 s 591 ms
averageAlgorithmExecutionTimeWithHu 20 m 44 s 591 ms

```
Paths found: 315

Time spent - 4 478 ms
Path from v1 to v2500
Paths found: 374

Time spent - 4 289 ms
Path from v1 to v2500
Paths found: 261

Average execution time after 100 iterations is - 7 112 ms

Process finished with exit code 0
```

5. References

1. https://dev.to/huyddo/graph-disjoint-set-2ldc - meaning of the graph term

6. Program text

6.1 GraphTask text

```java
import java.util.*;

/** Container class to different classes, that makes the whole
 * set of classes one class formally. */
public class GraphTask {

   /** Main method. */
   public static void main (String[] args) {
      GraphTask a = new GraphTask();
      a.run();
   }

   /** Actual main method to run examples and everything. */
   public void run() {
      Graph g = new Graph ("A");
      g.createRandomSimpleGraph (10, 45); // max (2500, 3123475)
      System.out.println(g.toString());
      Path optimizedPath = getOptimizedPath(g, "v1", "v2"); // find path from first vertex
with id v1 to vertex with id v2
      System.out.println("\nArc representation of best path:\n" +
optimizedPath.printArcPath());
      System.out.println("\nVertex representation of best path:\n" +
optimizedPath.toString());
   }

   /** Method that returns the best possible path, where the highest point is the lowest.
    * @param source an object containing vertices and arcs.
    * @param startId a vertex id value that is the starting point of the path search.
    * @param destinationId the vertex id value that is the end point of the path.
    * @return Path class object that contains vertices and arcs from start point to
destination point.
    */
   public Path getOptimizedPath(Graph source, String startId, String destinationId) {
      GraphPathManager graphPathManager = new GraphPathManager();
      Vertex start = findVertex(source, startId);
      Vertex destination = findVertex(source, destinationId);
      List<Path> paths = new ArrayList<>();
      int index = 0;
      long startTime = System.currentTimeMillis();

      while (true) {
         if (source.findPath(graphPathManager, start, destination)) {
            graphPathManager.pathPointsCorrection(startId, destinationId);
            paths.add(index, graphPathManager.getPath());
            index++;
         } else {
            break;
         }
      }

      int highestPoint = 10000;
      Path bestPath = null;
      for (Path path : paths) {
         //System.out.println(path.toString()); // use it if you want to see all possible
```

```
vertex paths
         //System.out.println(path.printArcPath()); // use it if you want to see all
possible arc paths
         if (path.highestPoint <= highestPoint) {
            highestPoint = path.highestPoint;
            bestPath = path;
         }
      }

      System.out.printf("Time spent - %,d ms", System.currentTimeMillis() - startTime);
      System.out.println("\nPath from " + startId + " to " + destinationId);
      System.out.println("Paths found: " + paths.size());

      assert bestPath != null;
      return bestPath;
   }

   /** Custom exception class for error displaying that occur when finding a path. */
   class GraphPathException extends RuntimeException
      public GraphPathException(String message, String startId, String endId) {
         super(message + "\nStart vertex: " + startId + "\nDestination vertex: " + endId);
      }
   }

   /** Custom exception class for error displaying that occur when
    * finding a vertex or arc in graph. */
   class GraphException extends RuntimeException {
      public GraphException(String message) {
         super(message);
      }
   }

   /** Method searches for the required vertex in the given graph, if there is no such
vertex
    * it returns GraphException.
    * @param source search graph.
    * @param targetId id of the requested vertex. */
   public Vertex findVertex(Graph source, String targetId) {
      Vertex vertex = source.first;
      while (vertex != null) {
         if (vertex.id.equals(targetId)) return vertex;
         vertex = vertex.next;
      }
      throw new GraphException("Current graph does not contain this vertex - " + targetId);
   }

   /** Class that generates realistic elevation values ranging from -7 to 5642 meters
    * (the lowest and highest point in Europe above sea level). */
   static class HeightGenerator {
      static int previousHeight = 143; // 143m height of the center of Tallinn above sea
level
      static int maxValueElevation = 10; // 10m maximal difference between 2 vertices
      static int heightMin = -7; // The lowest point -7m is in the north of Rotterdam
      static int heightMax = 5642; // The height of Mount Elbrus is 5642m above sea level

      /** Method that randomly generates the height for a point, taking
       * into account the above parameters. */
      static int getHeight() {
         Random rand = new Random();
         int value = rand.nextInt(maxValueElevation);
         int direction = rand.nextBoolean() ? 1 : -1;
         value *= direction;
         if (previousHeight + value >= heightMin && previousHeight + value <= heightMax) {
            previousHeight += value;
         } else if (previousHeight + value < heightMin) {
            previousHeight = heightMin;
         } else {
            previousHeight = heightMax;
         }
         return previousHeight;
      }
   }

   /** Base vertex class with an additional height parameter to compose
    * algorithm that solves my problem. */
   class Vertex {

      public final String id;
      private Vertex next;
      private Arc first;
      private int info;
      private int height = HeightGenerator.getHeight(); // param height from custom height
generator

      Vertex (String s, Vertex v, Arc e) {
         id = s;
         next = v;
         first = e;
      }

      Vertex (String s) {
         this (s, null, null);
      }

      Vertex (String s, int h) {
         this.id = s;
         this.height = h;
      }

      public void setArc(Arc arc) { this.first = arc; }

      public void setNextVertex(Vertex nextVertex) { this.next = nextVertex; }

      @Override
      public String toString() {
         return id;
      }
   }

   /** Arc represents one arrow in the graph. Two-directional edges are
    * represented by two Arc objects (for both directions). */
   class Arc {

      private String id;
      private Vertex target;
      private Arc next;
      // You can add more fields, if needed

      Arc (String s, Vertex v, Arc a) {
         id = s;
         target = v;
         next = a;
      }

      Arc (String s) {
         this (s, null, null);
   }

      @Override
      public String toString() {
         return id;
      }

      // TODO!!! Your Arc methods here!
   }

   /** Class that contains two lists for conveniently representing the resulting path.
    * Has a height parameter for sorting and a unique id. */
   class Path {

      public final List<Vertex> pathVertexPoints; // list of path vertices
      public final List<Arc> pathArcPoints; // list of path arcs
      public final int highestPoint; // highest point of path
      public final int id; // unique id

      Path (List<Vertex> pathVertexPoints, List<Arc> pathArcPoints, int highestPoint, int
id) {
         this.pathVertexPoints = pathVertexPoints;
         this.pathArcPoints = pathArcPoints;
         this.highestPoint = highestPoint;
         this.id = id;
      }

      /** Method displaying path, displays arcs with template:
       * vertex1---vertex2 : vertex2---vertex3 : etc. */
      public String printArcPath() {
         StringBuilder result = new StringBuilder();
         result.append(id).append(": ");

        for (int i = pathArcPoints.size() - 1; i >= 0; i--) {
            result.append(pathArcPoints.get(i).id.replace("_", "---"));
            if (i != 0) result.append(" : ");
         }
         return result.toString();
      }

      /** Method that displays each vertex in the path and its height,
       * it also displays the highest point in the path and the total number of points.
       * Use this for more detailed information display. */
      @Override
      public String toString() {
         StringBuilder result = new StringBuilder();
         result.append(id);
         result.append(": ");
         for (int i = 0; i < pathVertexPoints.size(); i++) {
            result.append("(").append(pathVertexPoints.get(i).id)
                  .append(": ").append(pathVertexPoints.get(i).height).append("m)");
            if (i + 1 < pathVertexPoints.size()) result.append(" ==> ");
         }
         result.append("\nHighest point of path: ").append(highestPoint).append("m")
                  .append(" [Not included start and destination points if vertices more than
2]");
         result.append("\nPoints in the route:
").append(pathVertexPoints.size()).append("\n");
         return result.toString();
      }
   }
   /** Manager class that optimizes the found path and returns it as an object of the class
Path.
    * If one of the possible paths in the graph has a length of 2 vertices
    * (the beginning and immediately the end), then after finding this path, it must be
ignored
    * (arcToIgnore value) so that the method for finding the path does not fall into an
infinite loop.
    */
   class GraphPathManager {

      private int id = 1; // current path id
      private int highestPoint;
      private int heightBorder = 10000; // it will be used in path finding (Graph findPath
method)
      private String arcToIgnore = "";
      private List<Vertex> pathVertex = new ArrayList<>();
      private List<Arc> pathArc = new ArrayList<>();

      public void addVertex(Vertex vertex) {
         pathVertex.add(vertex);
      }

      /** Method returns an object of the class Path and clears variables for working with
coming data.
       * Returns an GraphPathException if the method is called with empty GraphPathManager
variables.
       * @return Path object. */
      public Path getPath() {
         if (pathVertex.size() == 0) {
        throw new GraphPathException("No available paths found!", "Null", "Null");
        }
         List<Vertex> pathVertexPoints = new ArrayList<>(pathVertex);
         List<Arc> pathArcPoints = new ArrayList<>(pathArc);
         if (pathVertex.size() == 2) arcToIgnore = pathArc.get(0).id;
         pathVertex.clear();
         pathArc.clear();
         return new Path(pathVertexPoints, pathArcPoints, highestPoint, id++);
      }

      /** Method checks each vertex from the end whether the previous vertex
       * has a connection with the next one.
       * If there is no connection, then this vertex is removed from the path,
       * if there is, then this point
       * becomes a next checkpoint until the end of list.
       * @param startId is used for highest point detection.
       * @param endId the same as previous. */
      public void pathPointsCorrection(String startId, String endId) {
         List<Vertex> verticesToRemove = new ArrayList<>();
         int current = pathVertex.size() - 1;
         while (current != 0)
            for (int i = 0; i < current; i++) {
               if (current - 1 - i >= 0 &&
                 !areVerticesConnected(pathVertex.get(current - 1 - i),
pathVertex.get(current))) {
                  verticesToRemove.add(pathVertex.get(current - 1 - i));
               }
               else {
                  current = current - 1 - i;
                  break;
               }
         }
         for (Vertex vertex : verticesToRemove) {
            pathVertex.remove(vertex);
         }

         setHighestPoint(startId, endId);
      }

      /** Method that checks whether two vertices are connected to each other, if it is true
it
       * also adds a connecting arc between these vertices to the pathArc list.
       * @param vertexFrom vertex that is checked for a connection with the next vertex.
       * @param vertexTo next vertex.
       * @return boolean vertices are connected or not. */
      private boolean areVerticesConnected(Vertex vertexFrom, Vertex vertexTo) {
         Arc arc = vertexFrom.first;
         while (arc != null) {
            if (arc.target.id.equals(vertexTo.id)) {
               pathArc.add(arc);
               return true;
            }
            arc = arc.next;
         }
         return false;
      }

      /** Method that sets the highest point in path. If the number of points in the path is
more than
       * 2, the method does not consider the height of the start and destination points.
       * @param startId id of start path point.
       * @param endId id of end path point. */
      private void setHighestPoint(String startId, String endId) {
         int value = -1000, startValue = 0, endValue = 0;
         for (Vertex vertex : pathVertex) {
            if (vertex.height > value && !vertex.id.equals(startId) &&
!vertex.id.equals(endId)) {
               value = vertex.height;
            } else if (vertex.id.equals(startId)) {
               startValue = vertex.height;
            } else if (vertex.id.equals(endId)) {
               endValue = vertex.height;
            }
         }
         heightBorder = value != -1000 ? value : heightBorder;
         if (value == -1000 ) value = Math.max(startValue, endValue);
         highestPoint = value;
      }
   }

    class Graph {

      private String id;
      private Vertex first;
      private int info = 0;

      Graph (String s, Vertex v) {
         id = s;
         first = v;
      }

      Graph (String s) { this (s, null); }

      @Override
      public String toString() {
```

```java
      String nl = System.getProperty ("line.separator");
      StringBuffer sb = new StringBuffer (nl);
      sb.append(id);
      sb.append (nl);
      Vertex v = first;
      while (v != null) {
         sb.append (v.toString());
         sb.append (" -->");
         Arc a = v.first;
         while (a != null) {
            sb.append (" ");
            sb.append (a.toString());
            sb.append (" (");
            sb.append (v.toString());
            sb.append ("->");
            sb.append (a.target.toString());
            sb.append (")");
            a = a.next;
         }
         sb.append (nl);
         v = v.next;
      }
      return sb.toString();
   }

   public Vertex createVertex (String vid) {
      Vertex res = new Vertex (vid);
      res.next = first;
      first = res;
      return res;
   }

   public Arc createArc (String aid, Vertex from, Vertex to) {
      Arc res = new Arc (aid);
      res.next = from.first;
      from.first = res;
      res.target = to;
      return res;
   }

   /** Create a connected undirected random tree with n vertices.
    * Each new vertex is connected to some random existing vertex.
    * @param n number of vertices added to this graph. */
   public void createRandomTree (int n) {
      if (n <= 0)
         return;
      Vertex[] varray = new Vertex [n];
      for (int i = 0; i < n; i++) {
         varray [i] = createVertex ("v" + String.valueOf(n-i));
         if (i > 0) {
            int vnr = (int)(Math.random()*i);
            createArc ("a" + varray [vnr].toString() + "_"
               + varray [i].toString(), varray [vnr], varray [i]);
            createArc ("a" + varray [i].toString() + "_"
               + varray [vnr].toString(), varray [i], varray [vnr]);
         } else {}
      }
   }

   /** Create an adjacency matrix of this graph.
    * Side effect: corrupts info fields in the graph
    * @return adjacency matrix. */
   public int[][] createAdjMatrix() {
```

```
   info = 0;
   Vertex v = first;
   while (v != null) {
      v.info = info++;
      v = v.next;
   }
   int[][] res = new int [info][info];
   v = first;
   while (v != null) {
      int i = v.info;
      Arc a = v.first;
      while (a != null) {
         int j = a.target.info;
         res [i][j]++;
         a = a.next;
      }
      v = v.next;
   }
   return res;
}

/** Create a connected simple (undirected, no loops, no multiple
 * arcs) random graph with n vertices and m edges.
 * @param n number of vertices
 * @param m number of edges. */
public void createRandomSimpleGraph (int n, int m) {
   if (n <= 0)
      return;
   if (n > 2500)
      throw new IllegalArgumentException ("Too many vertices: " + n);
   if (m < n-1 || m > n*(n-1)/2)
      throw new IllegalArgumentException
         ("Impossible number of edges: " + m);
   first = null;
   createRandomTree (n);       // n-1 edges created here
   Vertex[] vert = new Vertex [n];
   Vertex v = first;
   int c = 0;
   while (v != null) {
      vert[c++] = v;
      v = v.next;
   }
   int[][] connected = createAdjMatrix();
   int edgeCount = m - n + 1;  // remaining edges
   while (edgeCount > 0) {
      int i = (int)(Math.random()*n);  // random source
      int j = (int)(Math.random()*n);  // random target
      if (i==j)
         continue;  // no loops
      if (connected [i][j] != 0 || connected [j][i] != 0)
         continue;  // no multiple edges
      Vertex vi = vert [i];
      Vertex vj = vert [j];
      createArc ("a" + vi.toString() + "_" + vj.toString(), vi, vj);
      connected [i][j] = 1;
      createArc ("a" + vj.toString() + "_" + vi.toString(), vj, vi);
      connected [j][i] = 1;
      edgeCount--;  // a new edge happily created
   }
}
```

```java
     /** Main method which uses the breadth-first search algorithm and writes the traversed
vertices
        * to the GraphPathManager. It considers the heightBorder, which is
        * the highest point of the previous path. If each new path exists, then it
        * passes through the highest point which is lower than in the previous path found.
        * Returns true on success. If there is no path from point a to b, it throws an
        * GraphPathException.
        * @param path manager that processes the found path if path searching is successful.
        * @param start vertex from which path search starts.
        * @param destination path endpoint vertex.
        * @return boolean was the path found or not. */
      public boolean findPath(GraphPathManager path, Vertex start, Vertex destination) {
         Queue<Arc> queue = new LinkedList<>();
         List<String> visitedPoints = new ArrayList<>();
         queue.add(start.first);
         path.addVertex(start);
         visitedPoints.add(start.id);
         while (queue.size() > 0) {
            Arc current = queue.remove();
            while (current != null) {
               if (current.id.equals(path.arcToIgnore)) {
                 current = current.next;
                 continue;
                 }
               Vertex neighbor = current.target;
               if (!visitedPoints.contains(neighbor.id) &&
(neighbor.height < path.heightBorder || neighbor.id.equals(destination.id))) {
                  queue.add(neighbor.first);
                  path.addVertex(neighbor);
                  visitedPoints.add(neighbor.id);
                  if (neighbor.id.equals(destination.id)) return true;
               }
               current = current.next;
            }
         }
         if (path.id == 1) throw new GraphPathException("\nStart point and destination are
not connected!", start.id, destination.id);
         return false;
      }
   }
}
```

6.2 GraphTaskTest text

```java
import static org.junit.Assert.*;
import org.junit.Test;
import java.util.*;

/** Test class.
 * @author Mihhail Skripnik. */
public class GraphTaskTest {

   @Test (timeout=1000)
   public void algorithmPathDepthTestSet1() {
      GraphTask task = new GraphTask();
      GraphTask.Graph graph = getExampleGraph(new int[]{35, 47, 48, 49, 48, 50, 44, 34, 52, 
50});
      var time = System.currentTimeMillis();
      GraphTask.Path path = task.getOptimizedPath(graph, "A", "H");
      System.out.println(path.toString());
      System.out.println(path.printArcPath());
      System.out.println("Elapsed time: " + (System.currentTimeMillis() - time) + " ms");
      // right vertex path is A, B, C, D, E, H
      String[] resultIndexes = new String[]{"A", "B", "C", "D", "E", "H"};
      for (int i = 0; i < path.pathVertexPoints.size(); i++) {
         assertEquals(path.pathVertexPoints.get(i).id, resultIndexes[i]);
      }
   }

   @Test (timeout=1000)
   public void algorithmPathDepthTestSet2() {
      GraphTask task = new GraphTask();
      GraphTask.Graph graph = getExampleGraph(new int[]{100, 15, 90, 91, 25, 91, 50, 90, 90, 
100});
      var time = System.currentTimeMillis();
      GraphTask.Path path = task.getOptimizedPath(graph, "A", "H");
      System.out.println("Elapsed time: " + (System.currentTimeMillis() - time) + " ms");
       System.out.println(path.toString());
       System.out.println(path.printArcPath());
      // right vertex path is A, I, J, H
      String[] resultIndexes = new String[]{"A", "I", "J", "H"};
      for (int i = 0; i < path.pathVertexPoints.size(); i++) {
         assertEquals(path.pathVertexPoints.get(i).id, resultIndexes[i]);
      }
   }

@Test
   public void graphWithSimilarHeights() {
       GraphTask task = new GraphTask();
       GraphTask.Graph graph = getExampleGraph(new int[]{0, 0, 0, 0, 0, 0, 0, 0, 0, 0});
       var time = System.currentTimeMillis();
       GraphTask.Path path = task.getOptimizedPath(graph, "A", "H");
       System.out.println("Elapsed time: " + (System.currentTimeMillis() - time) + " ms");
       System.out.println(path.toString());
       System.out.println(path.printArcPath());
       // right vertex path is A, B, C, D, E, H
       String[] resultIndexes = new String[]{"A", "B", "C", "D", "E", "H"};
       for (int i = 0; i < path.pathVertexPoints.size(); i++) {
           assertEquals(path.pathVertexPoints.get(i).id, resultIndexes[i]);
       }
   }

   @Test (expected= GraphTask.GraphPathException.class)
   public void graphDoesNotContainPathFromTo() {
       GraphTask task = new GraphTask();
       GraphTask.Graph graph = getExampleGraph(new int[]{100, 15, 90, 91, 25, 91, 50, 90,
90, 100});
       GraphTask.Path path = task.getOptimizedPath(graph, "I", "B");
   }

   /** Attention! Maximum test execution time is 3 minutes! */
   @Test (timeout = 240000)
   public void averageAlgorithmExecutionTimeWithHugeData10Times() {
      GraphTask task = new GraphTask();
      List<Long> executionTime = new ArrayList<>();
      for (int i = 0; i < 10; i++) {
         GraphTask.Graph graph = task.new Graph("TEST");
         graph.createRandomSimpleGraph(2500, 3123475);
         var time = System.currentTimeMillis();
         task.getOptimizedPath(graph, "v1", "v2500");
         time = System.currentTimeMillis() - time;
         executionTime.add(time);
         System.out.println();
      }
      double averageTime = executionTime.stream().mapToDouble(a -> a).sum() /
executionTime.size();
      System.out.printf("Average execution time after 10 iterations is - %,d ms",
(int)averageTime);
      assertTrue(averageTime < 10000); // 10000 ms is 10s.
   }

   /** ATTENTION! Maximum test execution time is 30 minutes! */
   @Test (timeout = 2400000)
   public void averageAlgorithmExecutionTimeWithHugeData100Times() {
      GraphTask task = new GraphTask();
      List<Long> executionTime = new ArrayList<>();
      for (int i = 0; i < 100; i++) {
         GraphTask.Graph graph = task.new Graph("TEST");
         graph.createRandomSimpleGraph(2500, 3123475);
         var time = System.currentTimeMillis();
         task.getOptimizedPath(graph, "v1", "v2500");
         time = System.currentTimeMillis() - time;
         executionTime.add(time);
         System.out.println();
      }
      double averageTime = executionTime.stream().mapToDouble(a -> a).sum() /
executionTime.size();
      System.out.printf("Average execution time after 100 iterations is - %,d ms",
(int)averageTime);
      assertTrue(averageTime < 10000); // 10000 ms is 10s
   }

   private GraphTask.Graph getExampleGraph(int[] heights) {
      GraphTask task = new GraphTask();
      GraphTask.Vertex vertexA = task.new Vertex("A", heights[0]);
      GraphTask.Vertex vertexB = task.new Vertex("B", heights[1]);
      GraphTask.Vertex vertexC = task.new Vertex("C", heights[2]);
      GraphTask.Vertex vertexD = task.new Vertex("D", heights[3]);
      GraphTask.Vertex vertexE = task.new Vertex("E", heights[4]);
      GraphTask.Vertex vertexF = task.new Vertex("F", heights[5]);
      GraphTask.Vertex vertexG = task.new Vertex("G", heights[6]);
      GraphTask.Vertex vertexI = task.new Vertex("I", heights[7]);
      GraphTask.Vertex vertexJ = task.new Vertex("J", heights[8]);
      GraphTask.Vertex vertexH = task.new Vertex("H", heights[9]);
      vertexA.setNextVertex(vertexB);
      vertexB.setNextVertex(vertexC);
      vertexC.setNextVertex(vertexD);
      vertexD.setNextVertex(vertexE);
```

```
        vertexE.setNextVertex(vertexF);
        vertexF.setNextVertex(vertexG);
        vertexG.setNextVertex(vertexH);
        vertexH.setNextVertex(vertexI);
        vertexI.setNextVertex(vertexJ);
        GraphTask.Arc arcAI = task.new Arc("A_I", vertexI, null);
        GraphTask.Arc arcAG = task.new Arc("A_G", vertexG, arcAI);
        GraphTask.Arc arcAB = task.new Arc("A_B", vertexB, arcAG);
        vertexA.setArc(arcAB);
        GraphTask.Arc arcBC = task.new Arc("B_C", vertexC, null);
        vertexB.setArc(arcBC);
        GraphTask.Arc arcCD = task.new Arc("C_D", vertexD, null);
        vertexC.setArc(arcCD);
        GraphTask.Arc arcDE = task.new Arc("D_E", vertexE, null);
        vertexD.setArc(arcDE);
        GraphTask.Arc arcEH = task.new Arc("E_H", vertexH, null);
        vertexE.setArc(arcEH);
        GraphTask.Arc arcGF = task.new Arc("G_F", vertexF, null);
        vertexG.setArc(arcGF);
        GraphTask.Arc arcFE = task.new Arc("F_E", vertexE, null);
        vertexF.setArc(arcFE);
        GraphTask.Arc arcIJ = task.new Arc("I_J", vertexJ, null);
        vertexI.setArc(arcIJ);
        GraphTask.Arc arcJH = task.new Arc("J_H", vertexH, null);
        vertexJ.setArc(arcJH);
        return task.new Graph("GRAPH", vertexA);
    }

}
```

7. Example of execution

Below are some examples of program execution (console output):

First:

```
Time spent - 0 ms
Path from v1 to v10
Paths found: 3

Arc representation of best path:
2: av1---v8 : av8---v10

Vertex representation of best path:
2: (v1: 167m) ==> (v8: 154m) ==> (v10: 151m)
Highest point of path: 154m [Not included start and destination points if vertices more than 2]
Points in the route: 3
```

Second:

```
Time spent - 3 486 ms
Path from v1 to v2500
Paths found: 323

Arc representation of best path:
323: av1---v2225 : av2225---v2500

Vertex representation of best path:
323: (v1: 264m) ==> (v2225: 21m) ==> (v2500: 142m)
Highest point of path: 21m [Not included start and destination points if vertices more than 2]
Points in the route: 3
```

Third:

```
Time spent - 15 ms
Path from v1 to v1000
Paths found: 12

Arc representation of best path:
12: av1---v182 : av182---v439 : av439---v417 : av417---v111 : av111---v1000

Vertex representation of best path:
12: (v1: 77m) ==> (v182: 15m) ==> (v439: 3m) ==> (v417: 27m) ==> (v111: 8m) ==> (v1000: 148m)
Highest point of path: 27m [Not included start and destination points if vertices more than 2]
Points in the route: 6
```