Tallinna Tehnikaülikool

Individuaaltöö aines "Algoritmid ja andmestruktuurid"

Aruanne

Triinu Malv

IT-süsteemide arendus

Juhendaja Jaanus Pöial

Tallinn 2020

Ülesande püstitus

Ülesande aluseks on geograafiliste punktide graaf. Selle iga kaarega (*x*, *y*) on seotud väli (*x*, *y*).*l* - langus liikumisel punktist *x* punkti *y*. Kuna languse väärtus iseloomustab liikumist n-ö allikast sihtkohta, on tegu orienteeritud graafiga[1]. Ohutuimaks teeks kahe punkti vahel nimetatakse sellist teed, mille suurim lokaalne langus (punktist järgmisse punkti) on minimaalne. Kirjutada graafi laiuti läbimisel[2] põhinev algoritm, mille käigus leitakse ohutuim tee antud punktist *a* antud punkti *b*. Seega tuleb algoritmil leida sellistest kaartest koosnev teekond antud punktide vahel, mille välja "langus" summa oleks vähim võimalik, kasutades selleks graafi laiuti läbimist.

[1] *Orienteeritud graaf* - *graaf, mille tippe kujutatakse joonisel punktidena ja kaari nooltena tipust x tippu y. Kaare (x, y) algustipuks on tipp x ja lõpptipuks on tipp y, kusjuures y on tipu x naaber ehk (vahetu) järglane ning tipp x on tipu y (vahetu) eellane. (J. Kiho, 2003)*

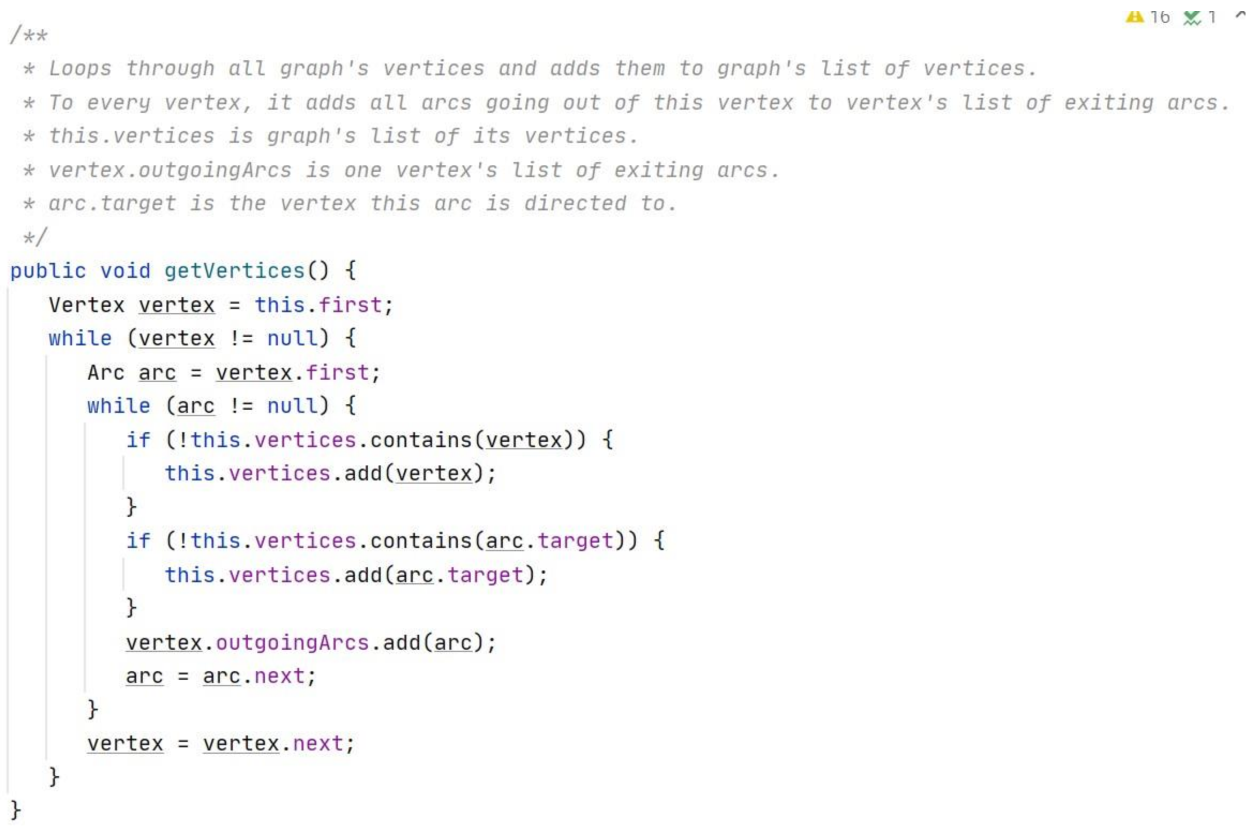
[2] *Graafi laiuti läbimine* - *põhitsükli igal sammul vaadeldakse juba vaatlusele võetud tippude hulga kõiki "naabreid". (J. Kiho, 2003)*

Lahenduse kirjeldus

Lahendus põhineb funktsioonil *safestPath(Vertex from, Vertex to)*, mis saab sisendiks 2 graafi tippu: teekonna algus ja lõpp, ning tagastab listi kaartega, mis näitavad ohutuimat teed algustipust lõpptippu.

Antud funktsioon kasutab lahenduseks omakorda 2 funktsiooni.

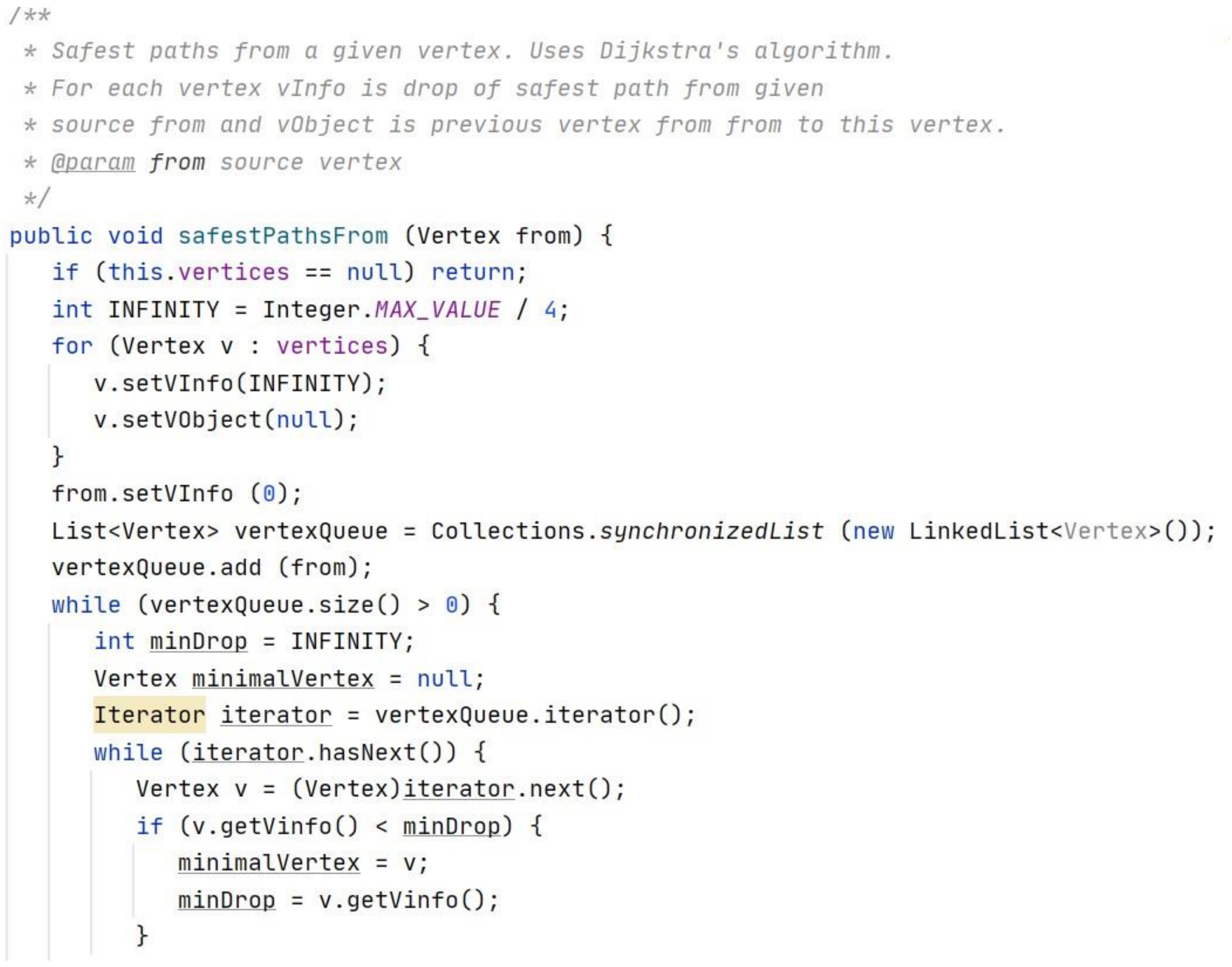
Kõigepealt kutsutakse välja funktsioon *getVertices()*. See käib kahekordse *while*-tsükliga graafi läbi ning lisab graafi väljale "*vertices*" listi graafi tippudega ja iga tipu väljale "*outgoingArcs*" listi sellest graafi tipust väljuvate kaartega.



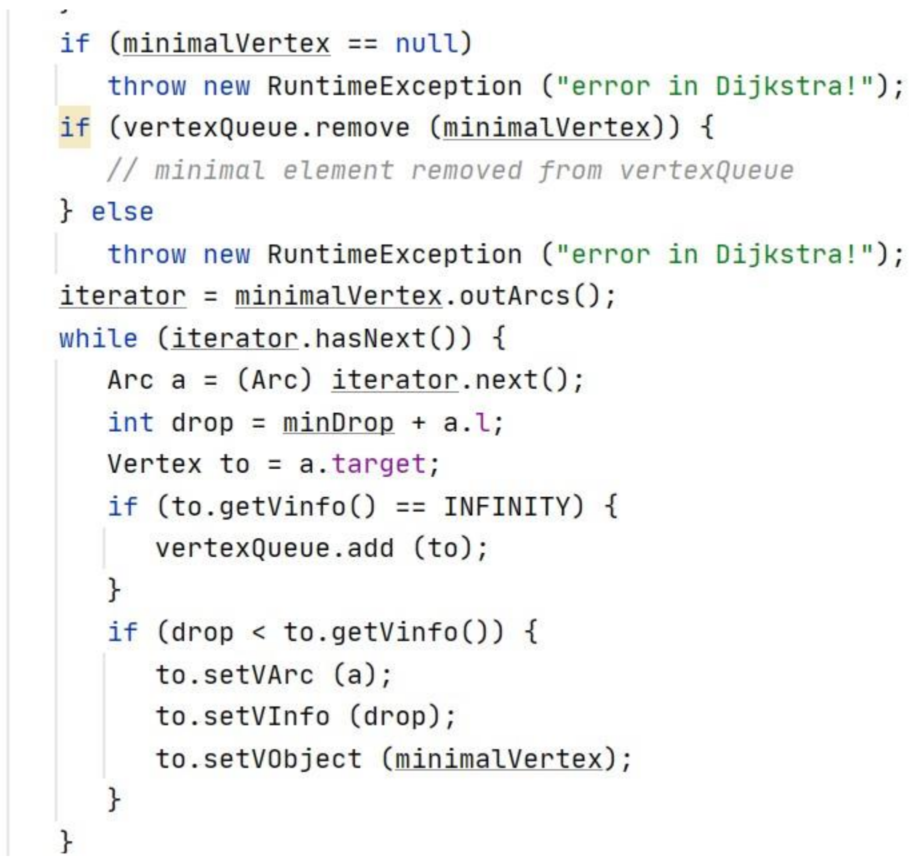
```java
/**
 * Loops through all graph's vertices and adds them to graph's list of vertices.
 * To every vertex, it adds all arcs going out of this vertex to vertex's list of exiting arcs.
 * this.vertices is graph's list of its vertices.
 * vertex.outgoingArcs is one vertex's list of exiting arcs.
 * arc.target is the vertex this arc is directed to.
 */
public void getVertices() {
   Vertex vertex = this.first;
   while (vertex != null) {
      Arc arc = vertex.first;
      while (arc != null) {
         if (!this.vertices.contains(vertex)) {
            this.vertices.add(vertex);
         }
         if (!this.vertices.contains(arc.target)) {
            this.vertices.add(arc.target);
         }
         vertex.outgoingArcs.add(arc);
         arc = arc.next;
      }
      vertex = vertex.next;
   }
}
```

Enne edasisi tegevusi kontrollib *safestPath(Vertex from, Vertex to)*, kas uuritav graaf sisaldab tippe *from* ja *to*. Kui ei, antakse vastav veateade.

Seejärel leitakse ohutuimad teed algustipust kõikidesse teistesse graafi tippudesse. Selleks kutsub *safestPath(Vertex from, Vertex to)* välja funktsiooni *safestPathsFrom(Vertex from)*, andes talle sisendiks kaasa vajaliku algustipu. Antud funktsioon kasutab lahendusel Dijkstra algoritmil põhinevat laiuti läbimist. See käib järjest läbi kõik võimalikud teekonnad algustipust teistesse tippudesse. Igal tipul on väljad "*info*" (näitab seni leitud ohutuima distantsi väärtust ehk hetkese teekonna moodustavate kaarte väljade "*l*" summat) ja "*safestArc*" (hetkeseisuga minimaalseima summa andev välja "*l*" väärtusega antud tippu viiv kaar). Iga uue teekonna leidmisel kontrollib funktsioon, kas seda kaart pidi minnes oleks "*l*" väljade summa väärtus praegusest madalam - seega väiksema langusega ehk ohutum. Kui nii on, antakse väljadele "*info*" ja "*safestArc*" uus väärtus. Ühtlasi määratakse uuritavale tipule uus eellane (äsja leitud kaare "*safestArc*" teises otsas olev tipp).

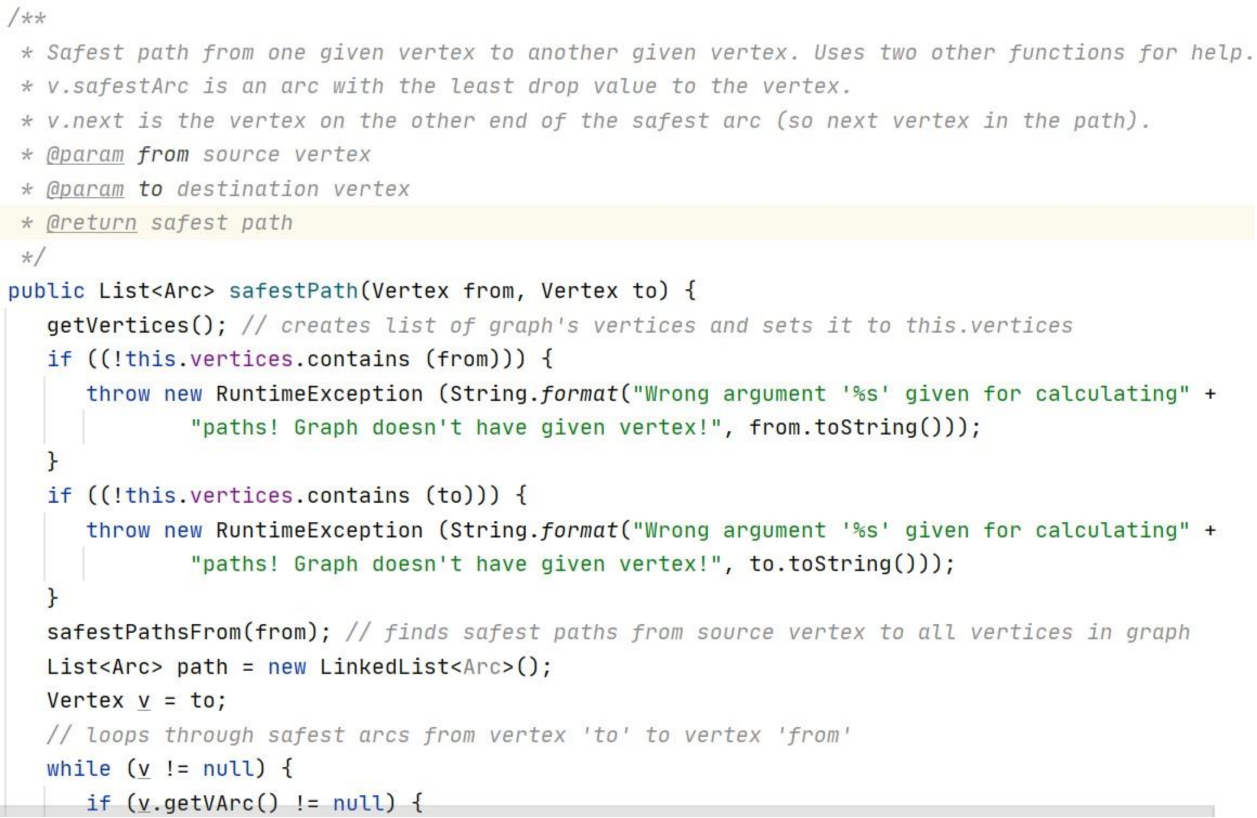


```
/**
 * Safest paths from a given vertex. Uses Dijkstra's algorithm.
 * For each vertex vInfo is drop of safest path from given
 * source from and vObject is previous vertex from from to this vertex.
 * @param from source vertex
 */
public void safestPathsFrom (Vertex from) {
   if (this.vertices == null) return;
   int INFINITY = Integer.MAX_VALUE / 4;
   for (Vertex v : vertices) {
      v.setVInfo(INFINITY);
      v.setVObject(null);
   }
   from.setVInfo (0);
   List<Vertex> vertexQueue = Collections.synchronizedList (new LinkedList<Vertex>());
   vertexQueue.add (from);
   while (vertexQueue.size() > 0) {
      int minDrop = INFINITY;
      Vertex minimalVertex = null;
      Iterator iterator = vertexQueue.iterator();
      while (iterator.hasNext()) {
         Vertex v = (Vertex)iterator.next();
         if (v.getVinfo() < minDrop) {
            minimalVertex = v;
            minDrop = v.getVinfo();
         }
```

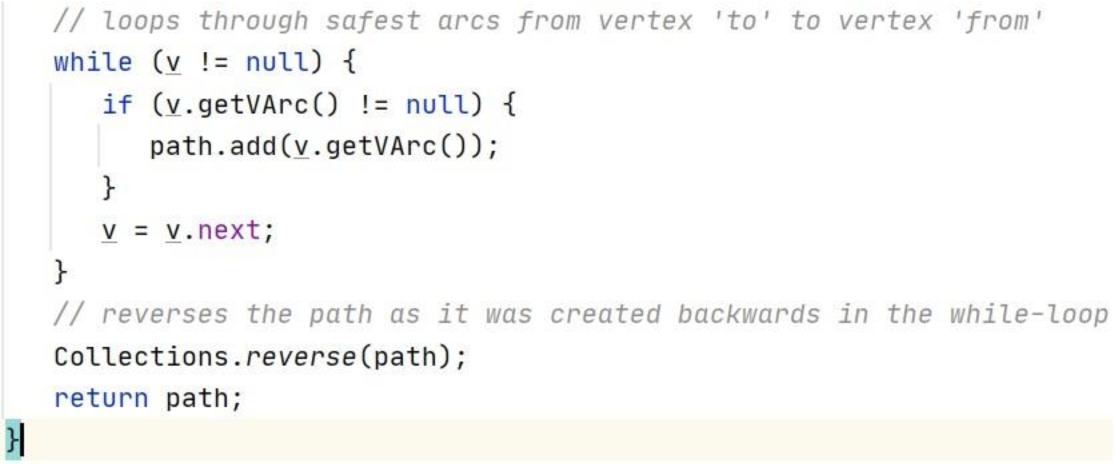


```
if (minimalVertex == null)
   throw new RuntimeException ("error in Dijkstra!");
if (vertexQueue.remove (minimalVertex)) {
   // minimal element removed from vertexQueue
} else
   throw new RuntimeException ("error in Dijkstra!");
iterator = minimalVertex.outArcs();
while (iterator.hasNext()) {
   Arc a = (Arc) iterator.next();
   int drop = minDrop + a.l;
   Vertex to = a.target;
   if (to.getVinfo() == INFINITY) {
      vertexQueue.add (to);
   }
   if (drop < to.getVinfo()) {
      to.setVArc (a);
      to.setVInfo (drop);
      to.setVObject (minimalVertex);
   }
}
```

Pärast neid funktsioone, kui graafi kaarte ja tippude vajalikud väljad on õigete väärtusega täidetud, loob funktsioon *safestPath(Vertex from, Vertex to)* edasiseks tegevuseks muutuja *path* (ArrayList<Arc> ehk nimekiri kaartest), kuhu omistada vajalik teekond ja *v* (Vertex ehk tipp), mille väärtuseks seatakse alguseks funktsioonile sisendina kaasaantud lõpptipp *to*. Seejärel käiakse *while*-tsükliga (tegutseb seni, kuni teekonnal enam järgmist tippu pole ehk jõutakse lõppu) läbi kõik tipud *to*-st *from*-ini, kusjuures igal ringil lisatakse töödeldava tipu "*safestArc*" listi *path*. Kuna sel viisil moodustub teekond tagurpidi, pööratakse see reverse() meetodiga ümber. Seejärel tagastabki funktsioon kahe tipu vahelise ohutuima teekonna.



```java
/**
 * Safest path from one given vertex to another given vertex. Uses two other functions for help.
 * v.safestArc is an arc with the least drop value to the vertex.
 * v.next is the vertex on the other end of the safest arc (so next vertex in the path).
 * @param from source vertex
 * @param to destination vertex
 * @return safest path
 */
public List<Arc> safestPath(Vertex from, Vertex to) {
   getVertices(); // creates list of graph's vertices and sets it to this.vertices
   if ((!this.vertices.contains (from))) {
      throw new RuntimeException (String.format("Wrong argument '%s' given for calculating" +
              "paths! Graph doesn't have given vertex!", from.toString()));
   }
   if ((!this.vertices.contains (to))) {
      throw new RuntimeException (String.format("Wrong argument '%s' given for calculating" +
              "paths! Graph doesn't have given vertex!", to.toString()));
   }
   safestPathsFrom(from); // finds safest paths from source vertex to all vertices in graph
   List<Arc> path = new LinkedList<Arc>();
   Vertex v = to;
   // loops through safest arcs from vertex 'to' to vertex 'from'
   while (v != null) {
      if (v.getVArc() != null) {
```



```java
   // loops through safest arcs from vertex 'to' to vertex 'from'
   while (v != null) {
      if (v.getVArc() != null) {
         path.add(v.getVArc());
      }
      v = v.next;
   }
   // reverses the path as it was created backwards in the while-loop
   Collections.reverse(path);
   return path;
}
```

Programmi kasutamisjuhend

Uue graafi loomiseks tuleb luua uus klassi *Graph* objekt, andes sellele sisendiks kaasa soovitud graafi n-ö nime sõne kujul. Seejärel on võimalik teha äsjaloodud graafile käsitsi tipud ja nendevahelised kaared või lasta seda automaatselt teha funktsioonil *createRandomSimpleGraph(int n, int m)*.

Käsitsi tipu loomiseks tuleb kasutada funktsiooni *createVertex(String vid)*, millele tuleb sisendina kaasa anda sõnena tipu id ehk soovitud tipu tähis. Käsitsi kaare loomiseks on funktsioon *createArc(String aid, Vertex from, Vertex to, int drop)*, millele tuleb parameetritena kaasa anda soovitud kaare nimi sõnena, kaare algustipp ja kaare lõpptipp (mõlemad eelnevalt *createVertex()* meetodiga loodud tipud) ning täisarvuline väärtus kaare languse iseloomustamiseks.

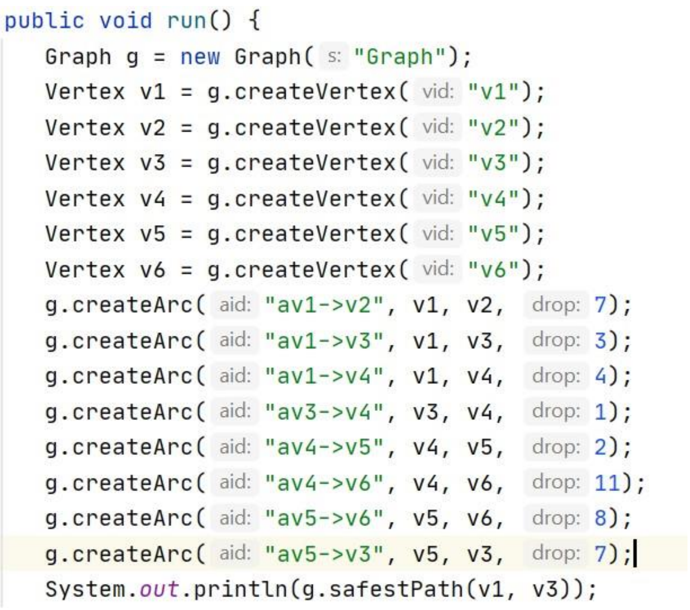
Automaatne graafi loomise funktsioon *createRandomSimpleGraph(int n, int m)* võtab sisendina *n* soovitud tippude arvu ning sisendina *m* soovitud kaarte arvu ning loob nendega juhusliku graafi, kusjuures loodud kaarte languste väärtused omistatakse selle käigus samuti juhuslikud.

Ohutuima tee leidmiseks loodud graafi ühest tipust teise, tuleb välja kutsuda meetod *safestPath(Vertex from, Vertex to)*, mille sisendiks tuleb anda soovitud algus- ja lõpptipp (Vertex objektidena). Funktsiooni tulemuse nägemiseks tuleb väljakutse ka välja printida (*System.out.println(graaf.safestPath(from, to))*). Funktsioon väljastab listi kaartest, mis viivad ohutuimalt algustipust lõpptippu.

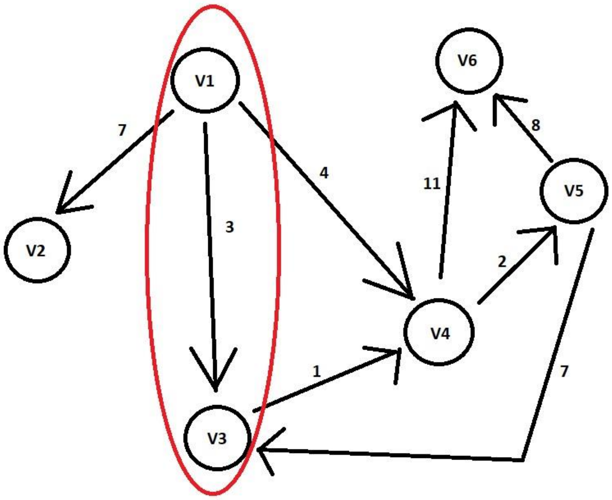
Testimiskava

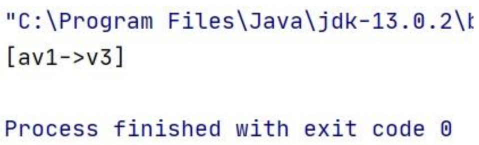
Testin algoritmi toimimist...

1) ...6 tipuga ja 8 kaarega graafil, kus ohutuim (minimaalse langusega) tee on ühtlasi vähima kaarte arvuga tee. Oodatav tulemus on joonisel märgitud punasega.



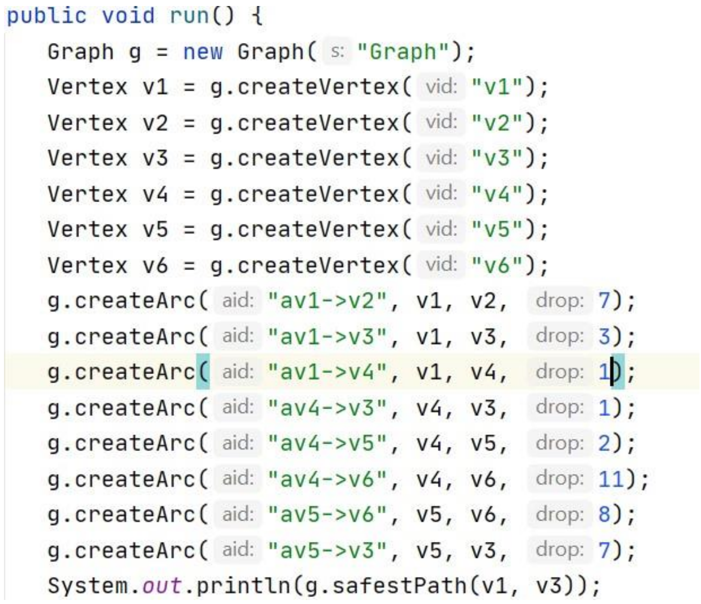
```
public void run() {
    Graph g = new Graph( s: "Graph");
    Vertex v1 = g.createVertex( vid: "v1");
    Vertex v2 = g.createVertex( vid: "v2");
    Vertex v3 = g.createVertex( vid: "v3");
    Vertex v4 = g.createVertex( vid: "v4");
    Vertex v5 = g.createVertex( vid: "v5");
    Vertex v6 = g.createVertex( vid: "v6");
    g.createArc( aid: "av1->v2", v1, v2,  drop: 7);
    g.createArc( aid: "av1->v3", v1, v3,  drop: 3);
    g.createArc( aid: "av1->v4", v1, v4,  drop: 4);
    g.createArc( aid: "av3->v4", v3, v4,  drop: 1);
    g.createArc( aid: "av4->v5", v4, v5,  drop: 2);
    g.createArc( aid: "av4->v6", v4, v6,  drop: 11);
    g.createArc( aid: "av5->v6", v5, v6,  drop: 8);
    g.createArc( aid: "av5->v3", v5, v3,  drop: 7);
    System.out.println(g.safestPath(v1, v3));
```



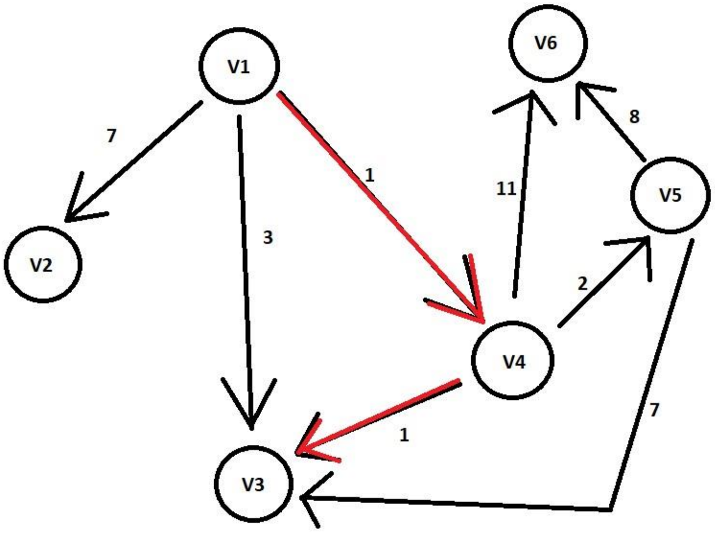


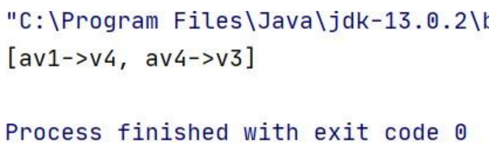
```
"C:\Program Files\Java\jdk-13.0.2\
[av1->v3]

Process finished with exit code 0
```

2) ...6 tipuga ja 8 kaarega graafil, kus ohutuim (minimaalse langusega) tee ei ole vähima kaarte arvuga tee. Oodatav tulemus on joonisel märgitud punasega.



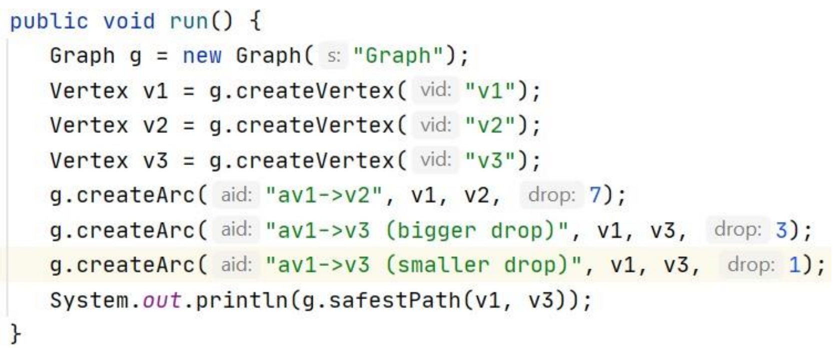
```
public void run() {
    Graph g = new Graph( s: "Graph");
    Vertex v1 = g.createVertex( vid: "v1");
    Vertex v2 = g.createVertex( vid: "v2");
    Vertex v3 = g.createVertex( vid: "v3");
    Vertex v4 = g.createVertex( vid: "v4");
    Vertex v5 = g.createVertex( vid: "v5");
    Vertex v6 = g.createVertex( vid: "v6");
    g.createArc( aid: "av1->v2", v1, v2,  drop: 7);
    g.createArc( aid: "av1->v3", v1, v3,  drop: 3);
    g.createArc( aid: "av1->v4", v1, v4,  drop: 1);
    g.createArc( aid: "av4->v3", v4, v3,  drop: 1);
    g.createArc( aid: "av4->v5", v4, v5,  drop: 2);
    g.createArc( aid: "av4->v6", v4, v6,  drop: 11);
    g.createArc( aid: "av5->v6", v5, v6,  drop: 8);
    g.createArc( aid: "av5->v3", v5, v3,  drop: 7);
    System.out.println(g.safestPath(v1, v3));
```



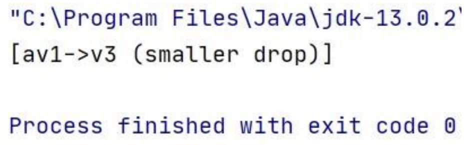


```
"C:\Program Files\Java\jdk-13.0.2\
[av1->v4, av4->v3]

Process finished with exit code 0
```

3) ...graafil, kus algustipust lõpp-punkti viib kaks erineva väärtusega otseteed (st 2 kaart otse allikast sihtpunkti, kuid vaja on valida väiksema langusega variant). Oodatav tulemus on loodud kaar nimega *„av1->v3 (smaller drop)“*.

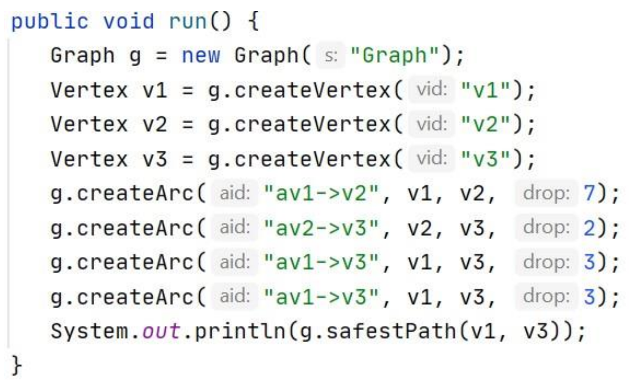


```
public void run() {
    Graph g = new Graph( s: "Graph");
    Vertex v1 = g.createVertex( vid: "v1");
    Vertex v2 = g.createVertex( vid: "v2");
    Vertex v3 = g.createVertex( vid: "v3");
    g.createArc( aid: "av1->v2", v1, v2,  drop: 7);
    g.createArc( aid: "av1->v3 (bigger drop)", v1, v3,  drop: 3);
    g.createArc( aid: "av1->v3 (smaller drop)", v1, v3,  drop: 1);
    System.out.println(g.safestPath(v1, v3));
}
```

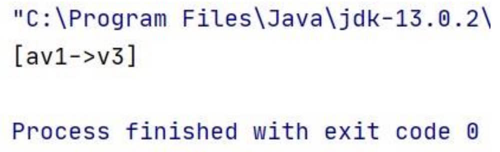


```
"C:\Program Files\Java\jdk-13.0.2'
[av1->v3 (smaller drop)]

Process finished with exit code 0
```

4) ...graafil, kus algustipust lõpptippu viivad kaks erinevat sama langusega teed ja kolmas suurema langusega tee (teen kindlaks, et algoritm oskab valida ükskõik kumma kahest vähimast). Oodatav tulemus on *„av1->v3“*.

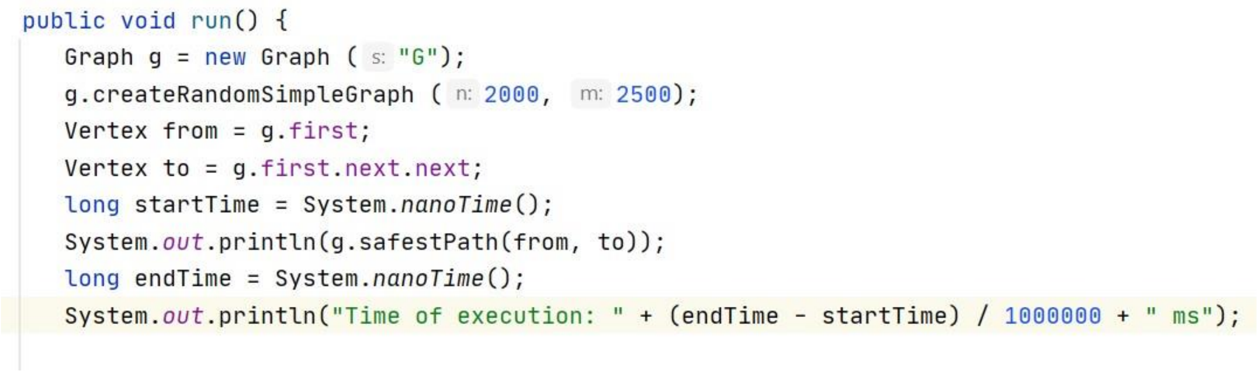


```
public void run() {
    Graph g = new Graph( s: "Graph");
    Vertex v1 = g.createVertex( vid: "v1");
    Vertex v2 = g.createVertex( vid: "v2");
    Vertex v3 = g.createVertex( vid: "v3");
    g.createArc( aid: "av1->v2", v1, v2,  drop: 7);
    g.createArc( aid: "av2->v3", v2, v3,  drop: 2);
    g.createArc( aid: "av1->v3", v1, v3,  drop: 3);
    g.createArc( aid: "av1->v3", v1, v3,  drop: 3);
    System.out.println(g.safestPath(v1, v3));
}
```

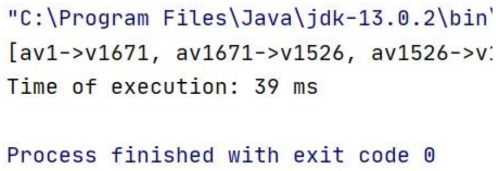


```
"C:\Program Files\Java\jdk-13.0.2\
[av1->v3]

Process finished with exit code 0
```

5) ...2000 tipu ja 2500 kaarega graafil.



```
public void run() {
   Graph g = new Graph ( s: "G");
   g.createRandomSimpleGraph ( n: 2000,  m: 2500);
   Vertex from = g.first;
   Vertex to = g.first.next.next;
   long startTime = System.nanoTime();
   System.out.println(g.safestPath(from, to));
   long endTime = System.nanoTime();
   System.out.println("Time of execution: " + (endTime - startTime) / 1000000 + " ms");
```



```
"C:\Program Files\Java\jdk-13.0.2\bin
[av1->v1671, av1671->v1526, av1526->v
Time of execution: 39 ms

Process finished with exit code 0
```

Kasutatud allikad

1. „Algoritmid ja andmestruktuurid. Ülesannete kogu.“, J. Kiho 2005. (http://kodu.ut.ee/~kiho/ads/spring10/Kirjandus/e-koopiaid/adsYlesannetKogu2005.pdf?fbclid=IwAR13DYrfq2OkzchQPZvNJZ_bnBQORCuJbmenPPC3e9IGe5x1AvTCmEdGtB8)

2. „Algoritmid ja andmestruktuurid“, J. Kiho 2003. (http://dspace.ut.ee/bitstream/handle/10062/16872/9985567676.pdf?sequence=1&isAllowed=y&fbclid=IwAR0MkDwrtHwGfEovjlYAKrzTNzMwXcu7BJDU_xuFi7k23hVSS7Snz6k37Ok)

3. „Graaf“, J. Pöial. (https://enos.itcollege.ee/~jpoial/algoritmid/graafid.html)

Lisad

Programmi tekst:

```
package kt6.src;

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("Graph");
        Vertex v1 = g.createVertex("v1");
        Vertex v2 = g.createVertex("v2");
        Vertex v3 = g.createVertex("v3");
        g.createArc("av1->v2", v1, v2, 7);
        g.createArc("av2->v3", v2, v3, 2);
        g.createArc("av1->v3", v1, v3, 3);
        g.createArc("av1->v3", v1, v3, 3);
        System.out.println(g.safestPath(v1, v3));
    }

    /** Field safestArc represents the arc entering to vertex that creates the path with the least
      * known drop.
      * List outgoingArcs holds the list of arcs exiting the vertex and is needed when traversing the graph.
      * */
    class Vertex {

        private final String id;
        private Vertex next;
        private Arc first;
        private int info = 0;
        private Arc safestArc;
        private final List<Arc> outgoingArcs = new LinkedList<Arc>();

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

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


    /** Method to set new value to vertex's info field.
      * @param i is the new value.
      * */
    public void setVInfo(int i) {
        this.info = i;
    }

    /** Method to get the value of vertex's info field.
      * */
    public int getVinfo() {
        return this.info;
    }

    /** Method to set new vertex object to vertex's next field. This is needed to form a path
      * between vertices.
      * @param v is the new vertex.
      * */
    public void setVObject(Vertex v) {
        this.next = v;
    }

    /** Method to get list of arcs exiting the vertex.
      * */
      public Iterator<Arc> outArcs() {
          return this.outgoingArcs.iterator();
      }

    /** Method to set new arc to vertex's safestArc field.
      * @param a is the new arc.
      * */
    public void setVArc(Arc a) {
        this.safestArc = a;
    }

    /** Method to get the arc of vertex's safestArc field.
      * */
    public Arc getVArc() {
        return this.safestArc;
    }
}


/** 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;
    private int l;

    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;
    }
}


class Graph {

    private String id;
    private Vertex first;
    private int info;
    private ArrayList<Vertex> vertices = new ArrayList<Vertex>();

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

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

    @Override
    public String toString() {
        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.l = new Random().nextInt(30); // creates random int drop value for this arc
        res.next = from.first;
        from.first = res;
        res.target = to;
        return res;
    }

    /**
     * Create new arc with certain drop value (chosen by user input).
     * @param aid arc id
     * @param from source vertex
     * @param to destination vertex
     * @param drop drop value for this arc
     */
    public Arc createArc (String aid, Vertex from, Vertex to, int drop) {
        Arc res = new Arc (aid);
        res.l = drop;
        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
     */
```

```java
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;
            edgeCount--;   // a new edge happily created
        }
    }

    /**
     * Loops through all graph's vertices and adds them to graph's list of vertices.
     * To every vertex, it adds all arcs going out of this vertex to vertex's list of exiting arcs.
     * this.vertices is graph's list of its vertices.
     * vertex.outgoingArcs is one vertex's list of exiting arcs.
     * arc.target is the vertex this arc is directed to.
     */
    public void getVertices() {
        Vertex vertex = this.first;
        while (vertex != null) {
            Arc arc = vertex.first;
            while (arc != null) {
                if (!this.vertices.contains(vertex)) {
                    this.vertices.add(vertex);
                }
                if (!this.vertices.contains(arc.target)) {
                    this.vertices.add(arc.target);
                }
                vertex.outgoingArcs.add(arc);
                arc = arc.next;
            }
            vertex = vertex.next;
        }
```

```java
}

/**
 * Safest path from one given vertex to another given vertex. Uses two other functions for help.
 * v.safestArc is an arc with the least drop value to the vertex.
 * v.next is the vertex on the other end of the safest arc (so next vertex in the path).
 * @param from source vertex
 * @param to destination vertex
 * @return safest path
 */
public List<Arc> safestPath(Vertex from, Vertex to) {
    getVertices(); // creates list of graph's vertices and sets it to this.vertices
    if ((!this.vertices.contains (from))) {
        throw new RuntimeException (String.format("Wrong argument '%s' given for calculating" +
                   "paths! Graph doesn't have given vertex!", from.toString()));
    }
    if ((!this.vertices.contains (to))) {
        throw new RuntimeException (String.format("Wrong argument '%s' given for calculating" +
                   "paths! Graph doesn't have given vertex!", to.toString()));
    }
    safestPathsFrom(from); // finds safest paths from source vertex to all vertices in graph
    List<Arc> path = new LinkedList<Arc>();
    Vertex v = to;
    // loops through safest arcs from vertex 'to' to vertex 'from'
    while (v != null) {
        if (v.getVArc() != null) {
            path.add(v.getVArc());
        }
        v = v.next;
    }
    // reverses the path as it was created backwards in the while-loop
    Collections.reverse(path);
    return path;
}

/**
 * Safest paths from a given vertex. Uses Dijkstra's algorithm.
 * For each vertex vInfo is drop of safest path from given
 * source from and vObject is previous vertex from from to this vertex.
 * @param from source vertex
 */
public void safestPathsFrom (Vertex from) {
    if (this.vertices == null) return;
    int INFINITY = Integer.MAX_VALUE / 4;
    for (Vertex v : vertices) {
        v.setVInfo(INFINITY);
        v.setVObject(null);
    }
    from.setVInfo (0);
    List<Vertex> vertexQueue = Collections.synchronizedList (new LinkedList<Vertex>());
    vertexQueue.add (from);
    while (vertexQueue.size() > 0) {
        int minDrop = INFINITY;
```

```
                Vertex minimalVertex = null;
                Iterator iterator = vertexQueue.iterator();
                while (iterator.hasNext()) {
                    Vertex v = (Vertex)iterator.next();
                    if (v.getVinfo() < minDrop) {
                        minimalVertex = v;
                        minDrop = v.getVinfo();
                    }
                }
                if (minimalVertex == null)
                    throw new RuntimeException ("error in Dijkstra!");
                if (vertexQueue.remove (minimalVertex)) {
                    // minimal element removed from vertexQueue
                } else
                    throw new RuntimeException ("error in Dijkstra!");
                iterator = minimalVertex.outArcs();
                while (iterator.hasNext()) {
                    Arc a = (Arc) iterator.next();
                    int drop = minDrop + a.l;
                    Vertex to = a.target;
                    if (to.getVinfo() == INFINITY) {
                        vertexQueue.add (to);
                    }
                    if (drop < to.getVinfo()) {
                        to.setVArc (a);
                        to.setVInfo (drop);
                        to.setVObject (minimalVertex);
                    }
                }
            }
        }
    }

}
```