Tallinna Tehnikaülikool

Individuaaltöö aines „Algoritmid ja andmestruktuurid“

Autor: Nikita Kums

Autori kood: IADB179097

Autori rühm: IADB32

Eriala: IT süsteemide arendus

Juhendaja: Jaanus Pöial

Tallinn 2018

Sisukord

Ülesande püstitus.

Individuaalülesandeks oli koostada meetod, mis leiab etteantud sidusas lihtgraafis kahe etteantud tipu vahelise lühima tee.

Lihtgraafiks nimetatakse silmusteta ehk puudub tipu seos iseendaga ja kordsete servadeta orienteerimata graafi, mis on selline graaf, kus kõik kaared on suunamata ja neid esitatakse graafi joonisel kahte tippu ühedava lihtsa joonega. Orienteerimata graaf osutub sidusaks, kui leidub tee mistahes tipust mistahes teisse tippu.

Õppejõu poolt oli ette antud programmitoorik, mis moodustab sidusa lihtgraafi kasutades klasse *Vertex*, *Arc* ning *Graph*.

Klass *Vertex* kujutab endast graafi ühte tippu.

Klass *Arc* kujutab endast seost graafi kahe tipu vahel. Selleks, et kujutada suunamata kaart on kaks tippu seotud mõlemat pidi.

Klass *Graph* kujutab endast graafi, mis koosneb tippudest (*Vertex*) ja neid ühendavatest seostest (*Arc*).

Individuaalülesande lahendamiseks kasutasin *Java* programmeerimiskeelt ning programmi *IntelliJ IDEA*.

Lahenduse kirjeldus

Ülesande lahendamiseks kasutasin *Breadth First Search* ehk graafi läbimist laiuti. Vastav algoritm alustab etteantud tipust, läbib kõik selle tipu naabrid ning seejärel vastavate tippude naabrid, kuni kõik võimalikud tipud on läbitud. Kuna tegemist on sidusa lihtgraafiga tagab esmakordne tipuni jõudmine selle tipuni lühima tee algustipust.

Lühima tee leidmiseks koostasin meetodi *findShortestPath*, mida kutsutakse välja graafi objekti abil ning parameetritena antakse tipu algustippu ja lõpptippu nimi, mille vahel tuleb leida lühim tee.

Esiteks teostasin kontrolli, et graafi tipud oleksid erinevate nimedega, poleks tühjad ning et nad oleksid olemas antud graafis. Viimane kontroll võimaldas samaaegselt leida parameetritena antud graafi tippudele vastavad objektid graafis endas.

Võtsin kasutusele atribuudid *visited* ning *parent*, millest esimene määras ära, kas vastav graafi tipp on juba läbitud ning teine näitas, milline on kõige lähim naaber-tipp vastavale tipule algustipu poolt.

Samuti kasutasin *LinkedList<Vertex>*, mille abil pidasin järjekorda tippudest, mida polnud veel läbitud.

Esimeseks tipuks oli tipp, mille nimi oli antud esimese parameetrina, mille lisasin järjekorda.

Iga iteratsiooni korral võtsin järjekorrast kõige esimese tipu ning kasutades meetodit *getVertexVertices* leidsin kõik selle tipuga seotud tipud. Seejärel läbisin kõik tipud, mis olin saanud meetodist *getVertexVertices* ning kui vastav tipp polnud veel läbitud, määrasin tippu läbituks, lisasin selle järjekorda ning lühimaks teeks selle tipuni määrasin tipu, millega kutsusin välja meetodi *getVertexVertices*. Kogu protsess kordus, kuni järjekord polnud tühi.

Võtsin kasutusele *List<Arc> path*, kus hoidsin lühimat teed tippe ühendava joone (*Arc*) objekti kujul.

Seejärel toimus iteratsioon alustades lõpptipust. Kui tippu naabriks osutus *null*, siis see tähendas seda, et olen jõudnud algustippu tagasi ning tee on leitud. Vastasel juhul käisin läbi kõik praeguse tipuga ühendatud jooned ning kontrollisin, kas mingi joon on ühendatud selle tipu naabriga. Kui olin vajaliku joone üles leidnud, lisasin selle joone *path listi* kasutades *getOtherEndOfArc* meetodit, mis tagastas selle sama joone, kuid sellisel kujul, et joon algas naabertippust ning suubus praegu käsil olevasse tippu.

Seejärel pöörasin *path* pahupidi, sest lühima tee otsimine toimus alates lõpptipust.

Viimaseks tagastab meetod lühima tee *List<Arc>* kujul.

Programmi kasutamisjuhend

Graafi moodustamiseks tuleb minna meetodisse *run().*

Olles meetodis *run()* saab graafi omistada kasutades konstruktorit *Graph()* ehk näiteks *Graph* ***aGraaf*** *= new Graph(„**b**“)*, kus ***aGraaf*** on graafi objekti nimi ning ***b*** on graafi nimi.

Seejärel tuleb kutsuda välja meetod *createRandomSimpleGraph(**x**, **y**)* kasutades eelnevalt loodud graafi objekti (siin näites ***aGraaf***) kus parameeter ***x*** määrab graafi tippude arvu ning parameeter ***y*** graafi tipude vaheliste seoste arvu ehk näiteks ***aGraaf***.*createRandomSimpleGraph(7, 14)*.

Selleks, et graafi väljastada võib kasutada *System.out.println(**aGraaf**)*, mis väljastab graafi külgnevusstruktuuri abil ehk graafi tipp ning kõik selle tipuga seotud teised tipud.

Nüüd, kui olete loonud graafi, mis koosneb ***x*** tippus ning ***y*** seosest, võib leida lühima tee graafi tippude vahel. Selleks tuleb kutsuda välja meetod *findShortestPath(start, end)* kasutades teie loodud graafi, kus meetodi *findShortestPath* parameeter ***start*** on algustipu nimi sõne kujul ning parameeter ***end*** on lõpptipu nimi sõne kujul ehk näiteks kui meil on 7 tippuline graaf, siis graafi tippude nimed on vastavalt v1, v2, ..., v7. Selleks, et leida lühim tee tipu v1 ja v7 vahel tuleb (kastudes graafina ***aGraaf***):

aGraaf.findShortestPath(„v1“, „v7“).

Kuid kuna meetod tagastab lühima tee *List<Arc>* kujul, siis tuleks meetodi *findShortestPath* tulemus salvestada vastavasse muutujasse*:*

List<Arc> ***lühimTee*** *= aGraaf.findShortestPath(„v1“, „v7“).*

Lühima tee printimiseks on kaks võimalust:

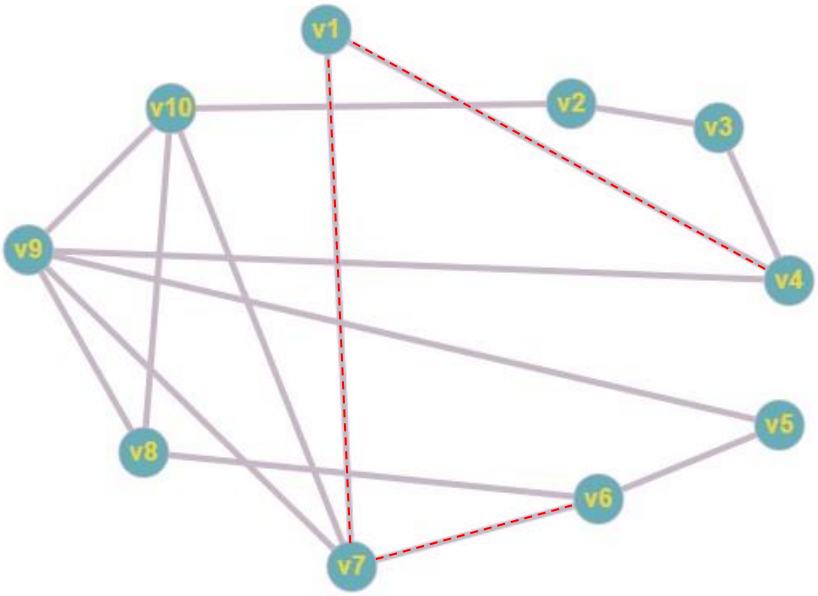
*System.out.println(**lühimTee**);*

System.out.println(aGraaf.findShortestPath(„v1“, „v7“));

Eeldades, et lühim tee tipust v1 tippu v7 on läbi tipu v6, siis mistahes printimise tulemuseks oleks: **[av1_v6, av6_v7]**

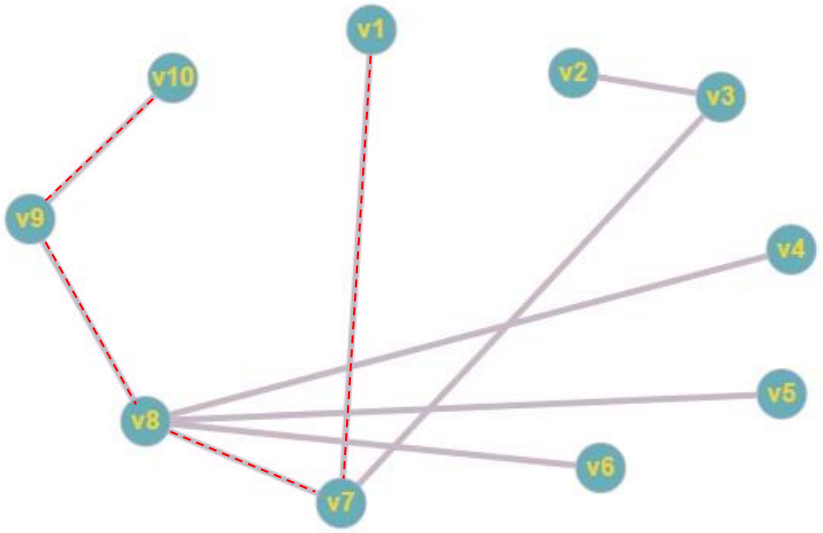
Testimiskava

Koostasin createRandomSimpleGraph(10, 15) meetodi abil graafi, millel on 10 tippu ja 15 seost.



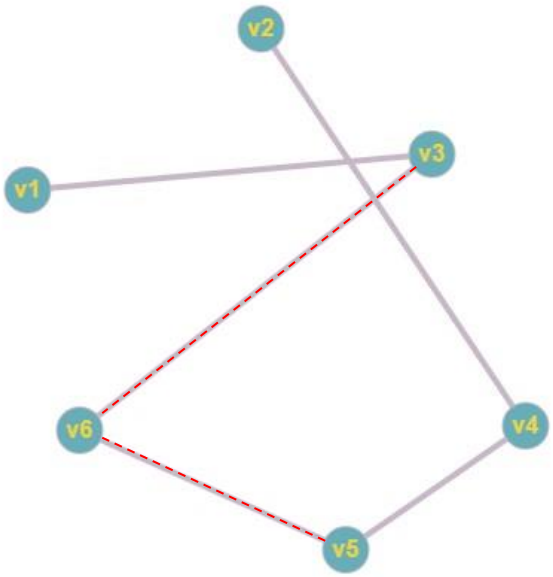
findShortestPath(„v4“, „v6“) (lühim tee tippude v4 ja v6 vahel) meetodi tulemuseks peab olema punasega märgitud tee (vt. lahendusnäide 1).

Koostasin createRandomSimpleGraph(10, 9) meetodi abil graafi, millel on 10 tippu ja 9 seost.



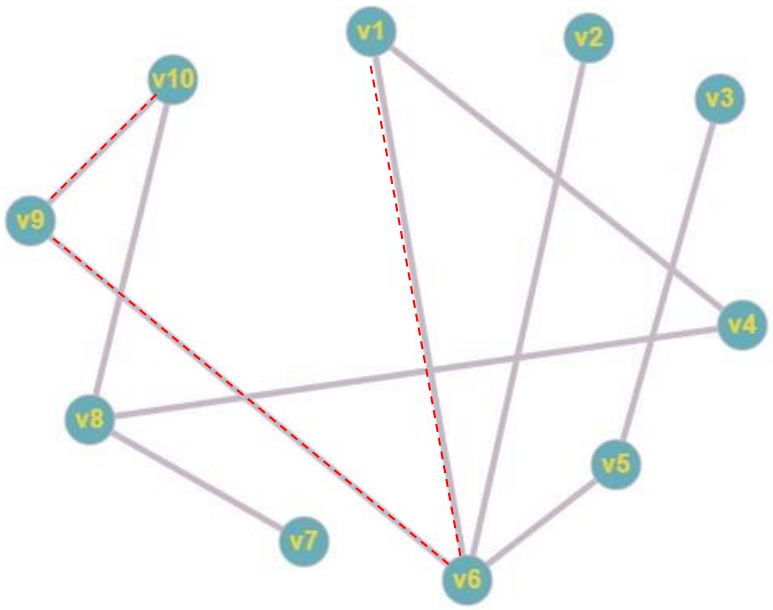
findShortestPath(„v1“, „v10“) (lühim tee tippude v1 ja v10 vahel) meetodi tulemuseks peab olema punasega märgitud tee (vt. lahendusnäide 2).

Koostasin createRandomSimpleGraph(6, 5) meetodi abil graafi, millel on 6 tippu ja 5 seost.



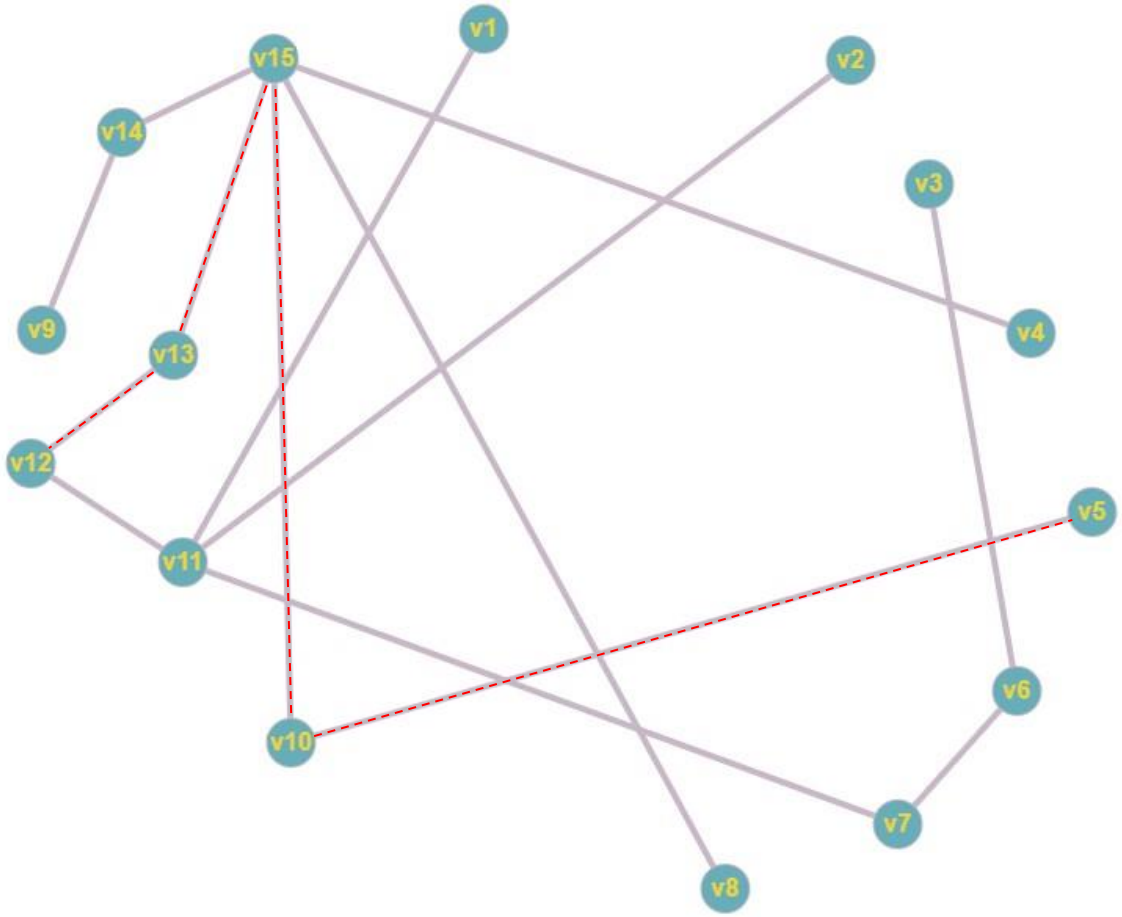
findShortestPath(„v5“, „v3“) (lühim tee tippude v5 ja v3 vahel) meetodi tulemuseks peab olema punasega märgitud tee (vt. lahendusnäide 3).

Koostasin createRandomSimpleGraph(10, 10) meetodi abil graafi, millel on 10 tippu ja 10 seost.



findShortestPath(„v1“, „v10“) (lühim tee tippude v1 ja v10 vahel) meetodi tulemuseks peab olema punasega märgitud tee (vt. lahendusnäide 4).

Koostasin createRandomSimpleGraph(15, 14) meetodi abil graafi, millel on 15 tippu ja 14 seost.



findShortestPath(„v5“, „v12“) (lühim tee tippude v5 ja v12 vahel) meetodi tulemuseks peab olema punasega märgitud tee (vt. lahendusnäide 5).

Koostan createRandomSimpleGraph(2000, 2302) meetodi abil graafi, millel on 2000 tippu ja 2302 seost, et mõõta aega, mis kulub lühima tee leidmiseks.

Kolme katse tulemusena osutus keskmiseks ajaks 15.6ms (vt. lahendusnäide 6, lahendusnäide 7, lahendusnäide 8).

Kasutatud allikad

Veebilehed:

https://et.wikipedia.org/wiki/Laiuti_otsing

https://stackoverflow.com/questions/8379785/how-does-a-breadth-first-search-work-when-looking-for-shortest-path

http://enos.itcollege.ee/~jpoial/algoritmid/graafid.html

Lisad

Lisa 1. Programmi täielik tekst

```
import java.util.*;
//https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
//https://stackoverflow.com/questions/8379785/how-does-a-breadth-first-
search-work-when-looking-for-shortest-path
/** 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() {
   }

   /**
    * Vertex represents one node in the graph.
    * Connected via Arcs to other vertexes.
    */
   class Vertex {

      private String id;
      private Vertex next;
      private Arc first;
      private int info = 0;
      public boolean visited = false;
      Vertex parent = null;
      // You can add more fields, if needed

      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;
      }

   }
```

```
/**
 * 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 info = 0;
   // 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;
   }

}


class Graph {

   private String id;
   private Vertex first;
   private int info = 0;
   // You can add more fields, if needed

   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");
   StringBuilder sb = new StringBuilder(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;
}
```

```java
/**
 * 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
   }
}
```

```
      /**
       * Find shortest path from one vertex to other vertex from graph.
       *
       * @param startingVertexName starting vertex name
       * @param endingVertexName ending vertex name
       * @return List of arcs that form the shortest path
       */
      public List<Arc> findShortestPath(String startingVertexName, String
endingVertexName) {

         //empty name supplied
         if (startingVertexName.trim().isEmpty() ||
endingVertexName.trim().isEmpty()) {
            throw new RuntimeException("Empty vertex name supplied.");
            //System.out.println("Empty vertex name supplied.");
            //System.exit(0);
         }

         //check if starting vertex is the ending vertex
         if (startingVertexName.equals(endingVertexName)) {
            throw new RuntimeException("START vertex and END vertex have
same names!");
            //System.out.println("START vertex and END vertex are the same!
(Method was not executed)");
            //System.out.println(startingVertex + " -> " + endingVertex);
            //System.exit(0);
         }

         Vertex start = new Vertex(startingVertexName);
         Vertex end = new Vertex(endingVertexName);

         Vertex firstVertex = first;

         // get starting and ending vertex and validate vertexes from
parameters
         Vertex startVertex = null;
         Vertex endVertex = null;

         while (firstVertex != null) {
            if (firstVertex.id.equals(start.id)) {
               startVertex = firstVertex; // starting vertex
            }
            if (firstVertex.id.equals(end.id)) {
               endVertex = firstVertex; // ending vertex
            }
            firstVertex = firstVertex.next;
         }

         // vertexes do not exist in the graph
         if (startVertex == null || endVertex == null) {
            throw new RuntimeException("No such vertex named: " +
(startVertex == null ? start.id : end.id));
            //System.out.println("No such vertex named: " + (startVertex ==
null ? start.id : end.id));
            //System.exit(0);
         }
```

```
        LinkedList<Vertex> queue = new LinkedList<>(); // queue for bfs

        startVertex.visited = true; // starting vertex is visited
        startVertex.parent = null;
        queue.add(startVertex); // add starting vertex to queue

        while (queue.size() != 0) {
           start = queue.poll();
           List<Vertex> connectedVertexes = getVertexVertices(start); //
get all connected vertexes from current vertex
           for (Vertex vertex : connectedVertexes) {
              if (!vertex.visited) {
                 vertex.visited = true;
                 vertex.parent = start; // current 'parent' vertex is
closest way to this vertex
                 queue.add(vertex);
              }
           }
        }

        List<Arc> path = new ArrayList<>(); // holds path as arc objects

        // each vertex (VERTEX) has a parent vertex that is the closest
way of reaching VERTEX thus iterating over
        // parents will yield the closest way from end point till start
point.
        Vertex temp = endVertex;
        while (true){
           if(temp.parent == null){
              break;
           }
           Arc arc = temp.first;
           while (arc.target != temp.parent){
              arc = arc.next;
           }
           path.add(getOtherEndOfArc(arc, temp));
           temp = temp.parent;
        }
        Collections.reverse(path);
        return path;
     }

     /**
      * Find all vertexes who are connected to given vertex.
      *
      * @param vertex Vertex whose 'children' are to be found
      * @return List of all connected vertexes with given vertex
      */
     private List<Vertex> getVertexVertices(Vertex vertex) {
        List<Vertex> result = new ArrayList<>();
        Arc temp = vertex.first;
        while (temp != null) {
           result.add(temp.target);
           temp = temp.next;
        }
        return result;
     }
```

```
        /**
         * Find reverse arc to given arc.
         *
         * @param arc Arc that needs it's reverse to be found
         * @param end Vertex that the reversed arc must connect to
         * @return Arc that is reverse of given arc
         */
        private Arc getOtherEndOfArc(Arc arc, Vertex end){
            Arc result = arc.target.first;
            while (result.target != end){
                result = result.next;
            }

            return result;
        }

    }
}
```

Lisa 2. Lahendusnäidete tulemused

```
G
v1 --> av1_v4 (v1->v4) av1_v7 (v1->v7)
v2 --> av2_v3 (v2->v3) av2_v10 (v2->v10)
v3 --> av3_v2 (v3->v2) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v3 (v4->v3) av4_v9 (v4->v9)
v5 --> av5_v6 (v5->v6) av5_v9 (v5->v9)
v6 --> av6_v7 (v6->v7) av6_v5 (v6->v5) av6_v8 (v6->v8)
v7 --> av7_v6 (v7->v6) av7_v9 (v7->v9) av7_v1 (v7->v1) av7_v10 (v7->v10)
v8 --> av8_v10 (v8->v10) av8_v6 (v8->v6) av8_v9 (v8->v9)
v9 --> av9_v7 (v9->v7) av9_v4 (v9->v4) av9_v5 (v9->v5) av9_v8 (v9->v8) av9_v10 (v9->v10)
v10 --> av10_v8 (v10->v8) av10_v2 (v10->v2) av10_v7 (v10->v7) av10_v9 (v10->v9)

[av4_v1, av1_v7, av7_v6]
```

Lahendusnäide 1. Graaf, millel on 10 tippu ja 15 seost.

```
G
v1 --> av1_v7 (v1->v7)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v2 (v3->v2) av3_v7 (v3->v7)
v4 --> av4_v8 (v4->v8)
v5 --> av5_v8 (v5->v8)
v6 --> av6_v8 (v6->v8)
v7 --> av7_v1 (v7->v1) av7_v3 (v7->v3) av7_v8 (v7->v8)
v8 --> av8_v4 (v8->v4) av8_v5 (v8->v5) av8_v6 (v8->v6) av8_v7 (v8->v7) av8_v9 (v8->v9)
v9 --> av9_v8 (v9->v8) av9_v10 (v9->v10)
v10 --> av10_v9 (v10->v9)

[av1_v7, av7_v8, av8_v9, av9_v10]
```

Lahendusnäide 2. Graaf, millel on 10 tippu ja 9 seost.

```
G
v1 --> av1_v3 (v1->v3)
v2 --> av2_v4 (v2->v4)
v3 --> av3_v1 (v3->v1) av3_v6 (v3->v6)
v4 --> av4_v2 (v4->v2) av4_v5 (v4->v5)
v5 --> av5_v4 (v5->v4) av5_v6 (v5->v6)
v6 --> av6_v3 (v6->v3) av6_v5 (v6->v5)

[av5_v6, av6_v3]
```

Lahendusnäide 3. Graaf, millel on 6 tippu ja 5 seost.

```
G
v1 --> av1_v6 (v1->v6) av1_v4 (v1->v4)
v2 --> av2_v6 (v2->v6)
v3 --> av3_v5 (v3->v5)
v4 --> av4_v1 (v4->v1) av4_v8 (v4->v8)
v5 --> av5_v3 (v5->v3) av5_v6 (v5->v6)
v6 --> av6_v1 (v6->v1) av6_v2 (v6->v2) av6_v5 (v6->v5) av6_v9 (v6->v9)
v7 --> av7_v8 (v7->v8)
v8 --> av8_v4 (v8->v4) av8_v7 (v8->v7) av8_v10 (v8->v10)
v9 --> av9_v6 (v9->v6) av9_v10 (v9->v10)
v10 --> av10_v8 (v10->v8) av10_v9 (v10->v9)

[av1_v6, av6_v9, av9_v10]
```

Lahendusnäide 4. Graaf, millel on 10 tippu ja 10 seost.

```
G
v1 --> av1_v11 (v1->v11)
v2 --> av2_v11 (v2->v11)
v3 --> av3_v6 (v3->v6)
v4 --> av4_v15 (v4->v15)
v5 --> av5_v10 (v5->v10)
v6 --> av6_v3 (v6->v3) av6_v7 (v6->v7)
v7 --> av7_v6 (v7->v6) av7_v11 (v7->v11)
v8 --> av8_v15 (v8->v15)
v9 --> av9_v14 (v9->v14)
v10 --> av10_v5 (v10->v5) av10_v15 (v10->v15)
v11 --> av11_v1 (v11->v1) av11_v2 (v11->v2) av11_v7 (v11->v7) av11_v12 (v11->v12)
v12 --> av12_v11 (v12->v11) av12_v13 (v12->v13)
v13 --> av13_v12 (v13->v12) av13_v15 (v13->v15)
v14 --> av14_v9 (v14->v9) av14_v15 (v14->v15)
v15 --> av15_v4 (v15->v4) av15_v8 (v15->v8) av15_v10 (v15->v10) av15_v13 (v15->v13) av15_v14 (v15->v14)

[av5_v10, av10_v15, av15_v13, av13_v12]
```

Lahendusnäide 5. Graaf, millel on 15 tippu ja 14 seost.

```
2000 vertex graph time test.
Method running time for Graph:
Vertexes: 2000
Arcs: 2302
Shortest path from v1 to v2000: [av1_v1774, av1774_v1982, av1982_v1990, av1990_v1997, av1997_v1999, av1999_v2000]
Result: 15ms
```

Lahendusnäide 6. 2000 tipust ja 2302 seoset koosneva graafi lühima tee leidmiseks kuluv aeg (Katse 1).

```
2000 vertex graph time test.
Method running time for Graph:
Vertexes: 2000
Arcs: 2302
Shortest path from v1 to v2000: [av1_v744, av744_v1044, av1044_v1391, av1391_v1950, av1950_v1962, av1962_v1979, av1979_v2000]
Result: 16ms
```

Lahendusnäide 7. 2000 tipust ja 2302 seoset koosneva graafi lühima tee leidmiseks kuluv aeg (Katse 2).

```
2000 vertex graph time test.
Method running time for Graph:
Vertexes: 2000
Arcs: 2302
Shortest path from v1 to v2000: [av1_v12, av12_v393, av393_v843, av843_v448, av448_v1886, av1886_v1924, av1924_v1988, av1988_v1996, av1996_v2000]
Result: 16ms
```

Lahendusnäide 8. 2000 tipust ja 2302 seoset koosneva graafi lühima tee leidmiseks kuluv aeg (Katse 3).