Tallinna Tehnikaülikool

Individuaaltöö aines „Algoritmid ja andmestruktuurid“

Koostaja: Helen Lepiku
IADB
Juhendaja: Jaanus Pöial

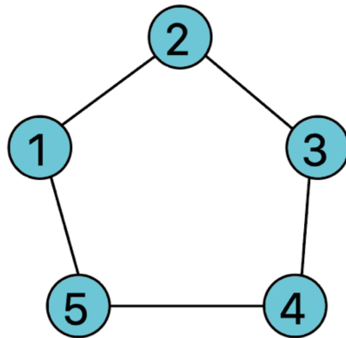
Tallinn 2020

1. Ülesande kirjeldus

Ülesandeks oli koostada meetod, mis leiab etteantud sidusas lihtgraafis lühima tee kahe teineteisest maksimaalselt kaugel paikneva tipu valel.

Graaf koosneb tippudest (V) ja sevadest (E). Ühendatud tipupaari nimetatakse naabertippudeks. Kui E elemente vaadeldakse kaheelemendiliste hulkadena, siis nimetatakse graafi orienteerimata graafiks ning hulka E servade hulgaks. Orienteerimata graafil on võimalik liikuda mööda teed nii edasi kui tagasi. Orienteerimata graafi nimetatakse siduvaks, kui leidub tee mistahes tipust mistahes teise tippu.[1]

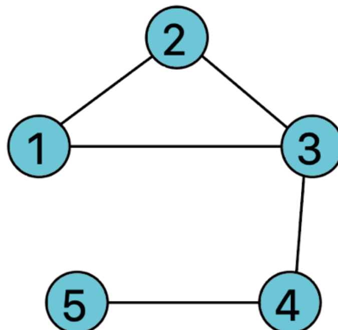
Antud programm peab leidma kaugeima tipu ja väljastama lühima tee kaugeimasse tippu. Kui programm leiab mitu sama pikka teed, siis väljastatakse vaid esimene.



Joonis 1

Joonisel 1 toodud graafi puhul oleks pikimaks teeks kaks valikut:

1. Tipust 1 → 2 tipust 2 → 3
2. Tipust 1→ 5 tipust 5 → 4



Joonis 2

Joonisel 2 toodud graafi puhul leidub vaid üks kaugeim tipp ja see oleks 5. Lühim tee kaugeimasse tippu tuleks tipust 1 → 3 tipust 3 → 4 tipust 4 → 5.

2. Lahenduse kirjeldus

Graafi lühima tee leidmine kaugeimasse tippu tuleks esmalt hakata graafi järjest läbima. Parim viis kaugeim tipp leida on laiuti läbida graaf.

Programmi käivitades vaatleme algselt, kas graafil üldse on kaari. Kui ei ole tagastame vastava täiendava teksti. Lisame esimese tipu vertexListi ja märgime tipu külastatuks.

```
if(first.first == null){
    System.out.println("Empty graph " + first);
}

Vertex vertex = first;
int nextVertex = 0;
vertexList.add(vertex);
vertex.visited = true;
```

Joonis 3

Programmis arc tähistab hetkel läbivaadatatavat kaart ja vertex hetkel vaadeldavat tippu. Esmalt on meil joonis 2 põhjal esimeseks tipuks 1 ja esimeseks kaareks 1 → 2. Kõigepeal kontrollime, kas vertexQueue sisaldab antud kaare tippu ehk tipp kahte ja kas vertexQueues on juba lisatud antud kaar. Kui ei ole, siis lisame kaare listi. Kui tipp kaks pole meie poolt eenlnevalt juba vaadeltud, siis märgimie visited tõeseks ja lisame tipu tippude järjekorda. Seejärel võtame ette tipu ühe järgneva kaare ja teeme kõik samamoodi läbi ja seda niikaua, kui tipul 1 rohkem kaari pole.

```
while(vertex != null){

    Arc arc = vertex.first;
      while (arc != null) {


          if(!vertexQueue.contains(arc.target.id) && !vertexQueue.contains(arc.id) ){
              addArcToLIst(arc, vertex);
          }

          if(!vertexQueue.contains(arc.id)){
              vertexQueue.add(arc.id);
          }
          if(!arc.target.visited){
              arc.target.visited = true;
              vertexList.add(arc.target);
          }
          arc = arc.next;

          if(arc == null){
              break;
          }

      }
```

Joonis 4. Programmi näide kaarte läbikäimiseks.

Kui esimesel tipul järgnev kaar puudus, siis lisame tipu vertexQueue'sse, et märgistada, et see tipp on käidud ja selle juurde tagasi tulla pole vaja. Kui vertexListis on veel käimata tippe võtame järgmise tipu. NextVertex annab teada, mitmenda tipu juures listis me parasjagu oleme.

```
 vertexQueue.add(vertex.id);
if(nextVertex + 1  >= vertexList.size()){
   break;
}else {
   nextVertex += 1;
  vertex = vertexList.get(nextVertex);
}
```

Joonis 5. Programmi näide tippude vahetamisest.

Uueks tipuks on nüüd 2. Suundume joonis 4 juurde tagasi ja kordame kõike eelnevat. Esimeks kaareks tipul 2 on 2 → 1. Kuna aga 1 on juba lisatud VertexQueue'sse siis seda kaart me vaatlema ei hakka ja võtame järgmise. Järgmiseks kaareks on 2 → 3. Tipp kolm ei kuulu vaadeldavate tippude hulka ja seega lisame tipu tippude järjekorda.

```
public void addArcToLIst(Arc arc, Vertex vertex) {

   ArrayList<Arc> path = new ArrayList<>();

   if (vertex.id.equals(vertexList.get(0).id)) {
      path.add(arc);
      distance.add(path);


   } else {
      for (ArrayList<Arc> paths : distance) {

         if(paths.get(paths.size() - 1).target.id.equals(vertex.id)){

            paths.add(arc);


         } else if(paths.size() > 1) {
               if (paths.get(paths.size() - 2).target.id.equals(vertex.id)){
                  ArrayList<Arc> arrayListClone = (ArrayList<Arc>) paths.clone();
                  arrayListClone.remove( index: paths.size() - 1);
                  arrayListClone.add(arc);
                  distance.add(arrayListClone);
                  break;
               }
         }
```

Joonis 6. Näide kaarte listide koostamisest.

Esimene tingimus vaatab, kas kaare tipp on kõige esimene tipp vertexListist, kui ei ole, suundub edasi. Kuna eelnevalt sai juba esimese tipu kaared lisatud, siis programm läheb edasi ja hakkab läbi käima kõiki kaari, mis on lisatud listi. Programmile on kaasa antud kaar ja selle tipp. Programm vaatleb kas listi lisatud kaare suubuv tipp on sama, mis kaasa antud kaare tipp, kui jah, siis lisab kaare samma listi. Kui programm ei leia sama tipuga kaart, siis vaadatakse, kus oli eelviimasel elemendil sama tipp ja tehakse sellest teekonnast koopia ja lisatakse uus kaar otsa. Koopia tuleb teha, kuna on leitud uus hargnevus, mida eelnevalt pole kirjas. Antud kaarel on programmis olemas juba eelnev kaar, milleks oli 1 → 2 ja seega praegune kaar 2 → 3 lisatakse sellele listile.

Kui kaar on listi lisatud naastakse joonis 3 juurde ja võetakse järgmine kaar ja kui puudub järgmine kaar, siis võetakse uus tipp. Kui tipud on kõik läbi käidud, siis hakatakse läbi käima kõiki listi kaarte kogumeid.

```
    ArrayList<Arc> smallestPath = new ArrayList<>();
    ArrayList<Arc> result = new ArrayList<>();

    for (ArrayList<Arc> paths : distance) {
        if (paths.get(paths.size() - 1).target.id.equals(vertexList.get(vertexList.size() -1).id)) {
            result = paths;

            if (smallestPath.size() == 0) {
                smallestPath = result;
            }

        }
        if (result.size() < smallestPath.size()) {
            smallestPath = result;
        }
    }

    return smallestPath;

}
```

Joonis 7. Näide programmi lühima listi teekondade leidmiseks.

Läbi käies vaadeldakse, kas viimane kaare suubumise tipp on vertexListi viimane tipp. Kuna programm läbis laiuti graafi, siis viimane vertexListi tipp on ühtlasi ka kaugeim tipp. Kui kaare lõpp-tipp on ka kaugeim tipp, siis salvestatakse see resulti. Kui smallestPathi pikkus on null, siis omistatakse esimene resulti tulemus sellega. Hiljem hakatakse võrdlema, kas resulti pikkus on suurem või väiksem kui smallestPathi oma. Kui result on lühem, siis seega on leitud hetkelolevast teest lühem ja smallestPath vahetab salvestatud listi resulti listi vastu. Kui kõik elemendid on läbi vaadatud, on leitud ka lühim list, mis koosneb kaarte teekonnast kaugeimasse tippu.

Leitud list saadetakse programmi distanceToString, et kuvada välja selgemini loetav lõpptulemus. Programm vaatab iga kaare läbi ja lisab → kaarte vahele, tehes seda niikaua, kuni kaari rohkem pole ja seejärel kuvatakse tulemus konsooli.

```
public String distanceToString(ArrayList<Arc> path){

    String result = "";
    int count = 0;
    for (Arc each : path) {
        result += each.toString();
        if(path.size() -1 > count){
            result += " -> ";
            count++;
        }
    }

    return result;
}
```

Joonis 8. Näide graafi teekonna välja printimisest.

3. Programmi kasutusjuhend

Kõigepealt tuleks mõista graafi programmi kujutamisviisi. Joonisel 1 kujutatud graaf näeks programmi sisestamisel välja selline:

```
Graph g7 = new Graph( s: "G7");
Vertex e5 = g7.createVertex( vid: "5");
Vertex d4 = g7.createVertex( vid: "4");
Vertex c3 = g7.createVertex( vid: "3");
Vertex b2 = g7.createVertex( vid: "2");
Vertex a1 = g7.createVertex( vid: "1");
g7.createArc( aid: "1-2", a1, b2);
g7.createArc( aid: "2-1", b2, a1);
g7.createArc( aid: "2-3", b2, c3);
g7.createArc( aid: "3-2", c3, b2);
g7.createArc( aid: "3-4", c3, d4);
g7.createArc( aid: "4-3", d4, c3);
g7.createArc( aid: "4-5", d4, e5);
g7.createArc( aid: "5-4", e5, d4);
g7.createArc( aid: "5-1", e5, a1);
g7.createArc( aid: "1-5", a1, e5);


System.out.println(g7.distanceToString(g7.breadthFirstSearch()));
```

Joonis 9. Näide graafi loomisest.

Kasutaja saab ise valida, kuidas ta soovib graafi märgistada. Nimelt ei pea olema tippudeks numbrid, võivad olla hoopiski tähed.

Iga tipp tuleks eraldi luua, samuti tuleks luua ka kõik kaared. Ei tohi unustada, et orienteerimata graafil on mõlemas suunas kaar. Seega tipust 1 → 2 minev kaar tuleb ka märkida tipust 2 → 1 kaarena.

Kui sisu on loodud, tuleks see käivitada.

```
System.out.println(g7);
System.out.println(g7.distanceToString(g7.breadthFirstSearch()));
```

Joonis 10. Näide programmi käivitamisest.

Antud käivitamismeetod kutsub välja kõik eelpool mainitud programme.

Peale edukat käivitamist kuvatakse konsooli lõpptulemus. Antud programmi korral näeks see välja järgnev:

```
G7
1 --> 1-5 (1->5) 1-2 (1->2)
2 --> 2-3 (2->3) 2-1 (2->1)
3 --> 3-4 (3->4) 3-2 (3->2)
4 --> 4-5 (4->5) 4-3 (4->3)
5 --> 5-1 (5->1) 5-4 (5->4)

Graph shortest path
1-2 -> 2-3
```

Joonis 11. Näide graafi väljundist.

4. Testimiskava

Test 1
Ühe tipuga graaf ja mitte ühtegi kaart. Oodatav tulemus, teekond puudub.

```
G2
v1 -->

Empty graph v1
Graph shortest path
```

Test 2
Neli tippu ja neli kaart. Oodatud tulemus. Lühim tee läbib 3 tippu, mööda kahte kaart.

```
G3
v1 --> av1_v4 (v1->v4) av1_v2 (v1->v2)
v2 --> av2_v1 (v2->v1) av2_v4 (v2->v4)
v3 --> av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v2 (v4->v2) av4_v3 (v4->v3)

Graph shortest path
av1_v4 -> av4_v3
```

Test 3
Kolm tippu ja kaks kaart. Oodatud teepikkus on kaks kaart.

```
G4
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)

Graph shortest path
av1_v3 -> av3_v2
```

Test 4
Seitse tippu ja kaheksa kaart. Lühim tee tuli neli kaart.

```
G5
v1 --> av1_v3 (v1->v3) av1_v2 (v1->v2)
v2 --> av2_v1 (v2->v1) av2_v4 (v2->v4)
v3 --> av3_v1 (v3->v1) av3_v5 (v3->v5)
v4 --> av4_v7 (v4->v7) av4_v2 (v4->v2) av4_v5 (v4->v5)
v5 --> av5_v3 (v5->v3) av5_v4 (v5->v4) av5_v7 (v5->v7)
v6 --> av6_v7 (v6->v7)
v7 --> av7_v4 (v7->v4) av7_v5 (v7->v5) av7_v6 (v7->v6)

Graph shortest path
av1_v2 -> av2_v4 -> av4_v7 -> av7_v6
```

Test 5
Seitse tippu ja kolm kaart. Tulemus - liiga vähe kaari.

```
Exception in thread "main" java.lang.IllegalArgumentException: Impossible number of edges: 3
    at GraphTask$Graph.createRandomSimpleGraph(GraphTask.java:336)
    at GraphTask.run(GraphTask.java:138)
    at GraphTask.main(GraphTask.java:15)
```

Test 6
2000 tippu ja 2000 kaart. Tulemusele kulunud aeg.

```
Time: 1605878320542 milliseconds
```

5. Kasutatud allikad

1. J.Pöial. Graafid https://enos.itcollege.ee/~jpoial/algoritmid/graafid.html

2. J.Pöial. Breadth-First Search
   https://enos.itcollege.ee/~jpoial/algoritmid/lobjects/Graph/pdf/BFS.pdf

3. https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

6. Lisa

Programmi tekst

```
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 g1 = new Graph("G1");
      Vertex e5 = g1.createVertex("5");
      Vertex d4 = g1.createVertex("4");
      Vertex c3 = g1.createVertex("3");
      Vertex b2 = g1.createVertex("2");
      Vertex a1 = g1.createVertex("1");
      g1.createArc("1-2", a1, b2);
      g1.createArc("2-1", b2, a1);
      g1.createArc("2-3", b2, c3);
      g1.createArc("3-2", c3, b2);
      g1.createArc("3-4", c3, d4);
      g1.createArc("4-3", d4, c3);
      g1.createArc("4-5", d4, e5);
      g1.createArc("5-4", e5, d4);
      g1.createArc("5-1", e5, a1);
      g1.createArc("1-5", a1, e5);*/

     /* System.out.println(g1);
      System.out.println(g1.distanceToString(g1.breadthFirstSearch()));

      Graph g2 = new Graph("G2");
      g2.createRandomSimpleGraph(1,0);
      System.out.println(g2);
      System.out.println(g2.distanceToString(g2.breadthFirstSearch()));*/

     /* Graph g3 = new Graph("G3");
      g3.createRandomSimpleGraph(4,4);
      System.out.println(g3);
      System.out.println(g3.distanceToString(g3.breadthFirstSearch()));*/

      /*Graph g4 = new Graph("G4");
      g4.createRandomSimpleGraph(3,2);
      System.out.println(g4);
      System.out.println(g4.distanceToString(g4.breadthFirstSearch()));*/

     /* Graph g5 = new Graph("G5");
      g5.createRandomSimpleGraph(7,8);
```

```
   System.out.println(g5);
   System.out.println(g5.distanceToString(g5.breadthFirstSearch()));*/

  /*  Graph g6 = new Graph("G6");
   g6.createRandomSimpleGraph(7,3);
   System.out.println(g6);
   System.out.println(g6.distanceToString(g6.breadthFirstSearch()));*/

   Graph g7 = new Graph("G6");
   g7.createRandomSimpleGraph(10,10);
   System.out.println(g7);
   System.out.println(g7.distanceToString(g7.breadthFirstSearch()));
   /*Graph g7 = new Graph("G7");*/
/*    g7.createRandomSimpleGraph(2000,2000);
   Long time = System.currentTimeMillis();
   System.out.println(g7);
   System.out.println("Time: " + time + " milliseconds");*/

 }

 private ArrayList<ArrayList<Arc>> distance = new ArrayList<>();
 List<String> vertexQueue = new ArrayList<>();
 List<Vertex> vertexList = new ArrayList<>();
 class Vertex {

   private String id;
   private Vertex next;
   private Arc first;
   private int info = 0;
   boolean visited;


   // 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;
   }

   // TODO!!! Your Vertex methods here!
 }


 /**
  * 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;
   }

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


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");
      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();
   }
```

```java
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;
}
```

```java
      /**
       * 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
         }


      }

     /* Going through all arcs and vertexes to find breadth first search through
shortest way*/

      public  ArrayList<Arc> breadthFirstSearch(){


         if(first.first == null){
            System.out.println("Empty graph " + first);
         }

         Vertex vertex = first;
         int nextVertex = 0;
         vertexList.add(vertex);
         vertex.visited = true;

         while(vertex != null){

               Arc arc = vertex.first;
                  while (arc != null) {
```

```
                    if(!vertexQueue.contains(arc.target.id) &&
!vertexQueue.contains(arc.id) ){
                       addArcToLIst(arc, vertex);
                    }

                    if(!vertexQueue.contains(arc.id)){
                       vertexQueue.add(arc.id);
                    }
                    if(!arc.target.visited){
                       arc.target.visited = true;
                       vertexList.add(arc.target);
                    }
                    arc = arc.next;

                    if(arc == null){
                       break;
                    }

                 }

           vertexQueue.add(vertex.id);
          if(nextVertex + 1  >= vertexList.size()){
             break;
          }else {
             nextVertex += 1;
            vertex = vertexList.get(nextVertex);
          }

        }

        ArrayList<Arc> smallestPath = new ArrayList<>();
        ArrayList<Arc> result = new ArrayList<>();

        for (ArrayList<Arc> paths : distance) {
           if (paths.get(paths.size() -
1).target.id.equals(vertexList.get(vertexList.size() -1).id)) {
              result = paths;

              if (smallestPath.size() == 0) {
                 smallestPath = result;
              }

           }
           if (result.size() < smallestPath.size()) {
              smallestPath = result;
           }
        }

        return smallestPath;

     }

   /*   Adds new arc to the list.*/
     public void addArcToLIst(Arc arc, Vertex vertex) {

        ArrayList<Arc> path = new ArrayList<>();

        if (distance.size() == 0 || vertex.id.equals(vertexList.get(0).id)) {
           path.add(arc);
           distance.add(path);


        } else {
           for (ArrayList<Arc> paths : distance) {

              if(paths.get(paths.size() - 1).target.id.equals(vertex.id)){
```

```java
                    paths.add(arc);


                } else if(paths.size() > 1) {
                    if (paths.get(paths.size() - 2).target.id.equals(vertex.id)){
                        ArrayList<Arc> arrayListClone = (ArrayList<Arc>)
paths.clone();
                        arrayListClone.remove(paths.size() - 1);
                        arrayListClone.add(arc);
                        distance.add(arrayListClone);
                        break;
                    }
                }

            }
        }

    }


    /**
     * Will return arc list in string format.
     */
    public String distanceToString(ArrayList<Arc> path){

        System.out.println("Graph shortest path");
        String result = "";
        int count = 0;
        for (Arc each : path) {
            result += each.toString();
            if(path.size() -1 > count){
                result += " -> ";
                count++;
            }
        }

        return result;
    }

  }

}
```