Individuaaltöö aines

“Algoritmid ja andmestruktuurid”

Meetod, mis leiab etteantud sidusas lihtgraafis kahe etteantud tipu vahelise lühima tee

Koostaja: Kadri Jõgi

IT-süsteemide arenduse eriala

Juhendaja: Jaanus Pöial PhD

TALLINNA TEHNIKAÜLIKOOL

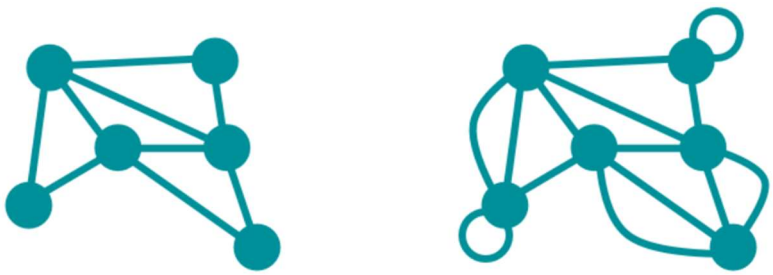
SÜGIS 2020

Sisukord

Mõistete selgitus

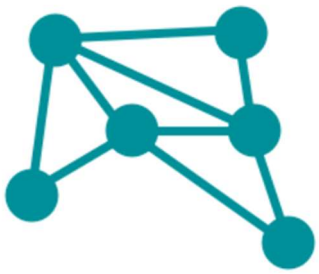
Lihtgraaf

Lihtgraafiks nimetatakse silmuste ja kordsete servadeta orienteerimata graafi. See tähendab, et mistahes kahe tipu vahel on vaid üks serv, ükski tipp ei ole serva abil ühendatud iseendaga ning servadel ei ole määratud kindlat suunda, vaid liikumine võib toimuda mõlemat pidi. Lihtgraafi näitena võib tuua kahesuunaliste maanteede võrgustiku, kus kahe linna vahel on vaid üks maantee ja igast linnast pääseb üht või mitut teed pidi teise linna.

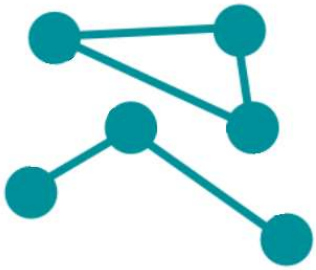


Joonis 1: Lihtgraaf ja multigraaf

Sidus graaf



Joonis 2: Sidus graaf



Joonis 3: Mittesidus graaf

Graafi nimetatakse sidusaks, kui leidub tee mistahes tipust mistahes teise tippu.

Sidusa graafi näiteks võiks olla Mandri-Eesti maanteevõrgustik, kus teid autoga läbides on (üldjuhul) võimalik igast algpunktist igasse lõpp-punkti pääseda. Samas kui näiteks Saaremaa teedevõrgustik on Mandri-Eestist eraldatud, nii et ainult autoga sõites sinna ei pääse.

Ülesande püstitus

Käesoleva töö eesmärk on kirjeldada meetodit, mis leiab lühima tee etteantud sidusas lihtgraafis kahe etteantud tipu vahel. Kuna eesmärk on leida lühim tee, eeldatakse, et graafi servad on erineva kaaluga, just nagu teed on eri pikkusega. Seega tuleb leida tee, kus kõikide teel läbitud servade kaalude summa on minimaalne.

Antud töös käsitletakse tee- ja servade pikkusi ainult naturaalarvude ehk positiivsete täisarvudena.

Lahenduse kirjeldus

Antud töö eesmärk on leida lühim tee kahe ette antud punkti vahel. Tegu on ühe graafiteooria põhiprobleemiga, mille lahendamise klassikaliseks võtteks on kasutada Dijkstra algoritmi. Sidusa positiivsete kaaludega lihtgraafi puhul leiab Dijkstra algoritm tee algpunktist kõikidesse teistesse graafi punktidesse. Algoritmi põhimõte on dünaamiliselt teepikkusi ümber hinnata ja lõpp-punkti viivaid pikemaid teid välistada.

Programmi kirjeldus

Lühima tee leidmiseks Dijkstra algoritmi abil kasutasin kolme andmeklassi: *Graph* (eesti k graaf), *Vertex* (tipp) ja *Arc* (kaar). Need hoiavad endas algoritmi lahendamiseks vajalikke andmevälju ja meetodeid.

Graph

Antud ülesandes esindab graafe ehk teedevõrgustikku klass *Graph*. Selle alla kuuluvad kõik tipud (*Vertex*) ja kaared (*Arc*), mille abil modelleerida eri teid ja verstaposte teede vahel. Klassi *Graph* kuuluvad kõik meetodid, mis tegelevad lühima tee arvutamisega nii manuaalselt sisestatud kui ka automaatselt genereeritud andmete põhjal ning selle välja arutatud lühima tee kujutamisega.

Graafi kirjeldatakse antud ülesandes järgmiste parameetritega:

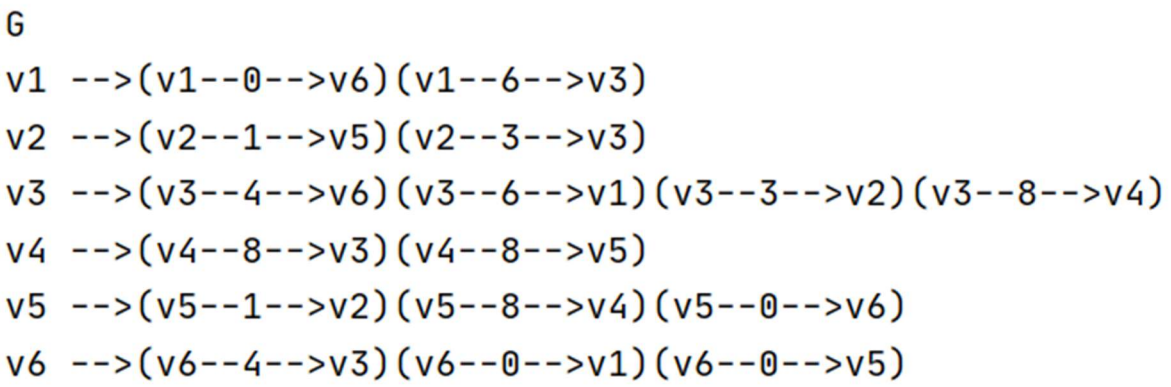
- *name* – igal graafil on sõnaline nimi
- *start* – leitava tee alguspunkt
- *finish* – leitava tee lõpp-punkt
- *vertexArray* – kogum tippe, millest graaf koosneb

Konstruktorid

Graafi loomiseks on olemas kaks võimalust. Graafi saab luua, tähistades seda ainult nimega või ka nime, algus ja lõpp-punktiga. Nende kahe eri mooduse jaoks on loodud kaks eraldi konstruktorit.

toString

Selleks, et graafe visuaalselt parem jälgida oleks, on loodud meetod *toString*, mis graafi teksti kujul esitab. Selle abil kujutatakse kõik graafi tipud ja neid ühendavad kaared koos iga kaare kaaluga.



Joonis 3: Graafi trükikuju. Tipud v1-v6 ja nendest lähtuvad kaared alg- ja lõpp-punkti ning kaaluga.

getterid, setterid

Graafi andmeväljadelt info saamiseks ja selle sinna seadmiseks on eri *get*- ja *set*-meetodid.

contains

Meetod kontrollimaks, kas graafi tippudenimekirjas on juba olemas tipp, mida lisada soovitakse.

reverse

Meetod, mis hõlbustab graafi lühima tee väljatrükki tagantpoolt ettepoole, keerates kogu teekonna ümber, et seda oleks parem jälgida.

shortestPathFrom

Antud ülesande põhimeetod, mis arvutab graafis välja lühima tee algpunktist *start* lõpp-punkti *finish.* Meetod toimib alguspunktist iga graafitipuni jõudva tee pideval ümberhindamisel. Meetodi alguses määratakse kõikide tippude kauguseks algpunktiks väga suur arv, mida tähistatakse nimega *INFINITY* ning on teada, et nii suurt pikkust ühelgi kaarel olla ei saa. Samuti nullitakse ära iga tipu eellane (*previous*), et saaks hakata kõiki vahemaid hindama. Algtipu kaugus iseendast määratakse nulliks.

Kõigepealt hinnatakse algpunkti. Selle eellane jääb endiselt nulliks ja kaugus iseendast samuti. Abimuutuja *minLen* abil jäädvustatakse seni teadaolev lühim teepikkus algpunktist ning *minVert* abil seni teadaolev lähim tipp.

Meetodi käigus vaadeldakse kõiki graafi tippe. Tippude vaheliseks kauguseks on neid ühendavate kaarte pikkus. Esimest korda mingisse tippu jõudes on selle kauguseks algpunktiks lõputult suur arv *INFINITY*, nii nagu meetodi alguses määrati. Selle info põhjal võib otsustada, et selles tipus pole varem käidud ning see tipp lisatakse töötlemata tippude loendisse *vq*. Vq-s olevate tippude puhul hinnatakse seni teadaolevat kaugust algpunktist. Kui see on väiksem muutujast *minLen*, siis uueks *minVert*i-iks saab vaatluse all olev tipp.

Seejärel uuritakse *minVert*ist väljuvaid kaari. Vahemuutuja *newLen* salvestab seni teadaoleva lühima tee algpunktist (*minLen*) ja kaare pikkuse summa ja *Vertex to* salvestab tipu, kuhu see kaar suubub. Kui vahemuutuja newLen on väiksem kui tipu *to* kaugus algpunktist, siis määratakse *newLen* tipu *to* uueks kauguseks algpunktist ja *minVert*-ist saab tipu *to* uus eellane.

Ehk kokkuvõtlikult toimub tipukauguste pidev ümberhindamine, kuni kõik tipud on töödeldud.

Algoritm arvutab alguspunkti kauguse kõikidest tippudest. Järgmine meetod *displayShortestPath* selekteerib nende seast välja lõpp-punkti, mida tähistab graafi väli *finish* ning kuvab tee ja teepikkuse ainult sinna tippu.

displayShortestPath

Antud ülesande üks põhimeetodeid, mis pärast lühima tee arvutamist selle välja kuvab. Meetod arvestab sellega, et pärast lühima tee arvutamist meetodi *shortestPathFrom* abil, on lõpp- ja alguspunkti vahele tekkinud tippude rada, kuna iga selle rajal oleva tipu andmeväljale previous on salvestatud sellele eelnev algpunktile lähim tipp ning andmeväljale vertexDistanceFromStart selle lähima tipu kaugus algpunktist. Meetod alustab lõpp-punktist ning liigub mööda tippudest rada tagasi algpunkti, joonistades selle raja sõnalisele kujule ja pöörates ülevaatlikkuse huvides pärast tagurpidi.

Näide graafi lühima tee ja selle kumulatiivse pikkuse kuvamisest meetodi *displayShortestPath* abil:

```
Shortest path from A to F with cumulative and total distance:

PATH:   A  -->  B  -->  E  -->  D  -->  F
DIST:   0       1       2       5       6

TOTAL:  6
```

Joonis 4: Lühima tee kujutamine. Joonisel on kuvatud tee, kumulatiivne vahemaa ja kogu teepikkus.

Automaatne graafi loomine

Meetodid *createVertex, createWeighedArc, createRandomTree, createAdjMatrix* ja *createRandomGraph* on mõeldud programmi testimise eesmärgil eri suuruses graafide ning nendes olevate tippude ja kaarte loomiseks ning ühendamiseks.

Vertex

Klass Vertex kujutab antud ülesandes kõiki tippe või verstaposte teel alguspunktis lõppu. Klassis hoitakse kõiki kaartepaare, mis tippe ühendavad ja võimaldavad mööda tippe igas suunas liikuda.

Klassi *Vertex* objekte kujutavad järgmised andmed:

- *name* – igal tipul on sõnaline nimi
- *previous* – teekonda läbides märgitakse igale tipule alguspunktile lähim eelmine tipp ning graafi läbides ja uusi lühemaid teid leides vahetatakse neid eelmisi tippe ka vastavalt välja.
- *first* – tipust väljuv esimene kaar

- *vertexDistanceFromStart* – graafi läbides märgitakse igale tipule kaugus algpunktist. Ka see info võib muutuda, kui leitakse mõni lühem tee.
- *arcs* – loend kaartest, mis on selle tipuga ühendatud

Konstruktorid

Tippu Vertex saab luua kahel moel: kas ainult nime abil või kohe määrates, milline on selle tipu eelmine tipp ja esimene kaar (*previous* ja *first*).

toString

Meetodi *toString* kaudu luuakse tipu trükikuju, milleks on antud juhul selle tipu nimi.

setterid, getterid

Eri *get*- ja *set*-meetodid määravad *Vertex*-klassi andmeväljade privaatse info või pärivad seda teiste meetodite jaoks.

containsArc

Antud meetod kontrollib, kas mingi kaar on juba tipu kaarteloendus *arcs*.

addArc

Selle meetodi abil lisatakse tipu kaarteloendisse *arcs* uus kaar.

Arc

Arc klassi objekt tähistab graafil olevaid tippe ühendavaid kaari. Kaks kaart koos tähistavad kahe tipu vahelist ühendust ehk graafi serva, mille kaudu võib liikuda mõlemas suunas.

Arc klassi objekte iseloomustab järgmine info:

- *name* – sõnaline nimi
- *target* – *Vertex* ehk tipp, kuhu kaar suubub
- *next* – järgmine kaar
- *length* – kaare pikkus

Konstruktorid

Arc objekti saab luua kahte moodi: ainult nime abil või nime, *target*i ja *next* abil.

toString

Arc objekti trükikuju luuakse toString meetodi abil, kuvades kaare nime, pikkust ja tippu, kuhu kaar suubub.

getterid, setterid

Eri *get*- ja *set*-meetodid määravad *Arc*-klassi andmeväljade privaatse info või pärivad seda teiste meetodite jaoks.

Programmi kasutusjuhend

1. Programmi kasutamisjuhend (liidese kirjeldus lihtkasutaja jaoks - näiteks millist meetodit milliste parameetritega kävitada, et tulemust saada)

Graafi loomiseks käsitsi on meetod *initYourGraph()*

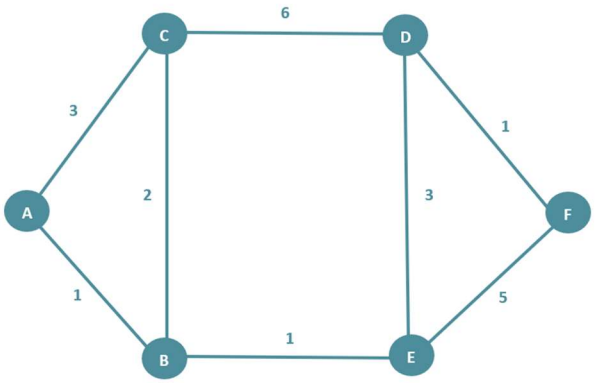
```
public void initYourGraph(Graph g) {

   //INITIATE YOUR GRAPH HERE!
   //Use variable and object names that suit your solution

   // 1. CREATE VERTICES
   //    Example:
   //    Vertex a = new Vertex("A",null, null);

   // 2. SET START AND FINISH
   //    Example:
   //    g.start = a;
   //    g.finish = f;

   // 3. SET GRAPH VERTEXES
   //    Example:
   //    g.setVertexArray(new Vertex[]{a, b, c, d, e, f>});

   // 4. CREATE ARC PAIRS, ADD ARC WEIGHT
   //    Example:
   //    Arc ab = g.createArc("A", a, b, 1);
   //    ab.setTargetVertex(b);
   //    Arc ba = g.createArc("B", b, a, 1);
   //    ba.setTargetVertex(a);

   // 5. CONNECT ARCS WITH VERTICES
   //    Example:
   //    a.addArc(ab);
   //    a.setFirstArc(ab);
   //    a.addArc(ac);

}
```

1. Loo tipud
2. Määra alg- ja lõpp-punkt
3. Määra tipud graafi vertexArray-le
4. Loo kaarepaarid, lisades neile kaalud
5. Ühenda tipud kaarte abil

Testimiskava

Näidisgraaf 1

Testisin lühima tee leidmist järgmise joonisel kujutatud graafi puhul



Joonis 5: Esimene testgraaf

Graafi loomine toimub GraphTask klassis *initSampleGraph. Graph* klassi *run()* meetodis seda välja kutsudes ja käsitsiloodud graafi kuvamismeetodi *displayGraph* abil avanes järgmine pilt:

Sisend:

```
public void run() {

   //Sample graph 1
   Graph g = new Graph( s: "G");
   initSampleGraph(g);
   g.displayGraph();
```

Joonis 6: Näidisgraafi loomise ja trükikuju kuvamise käsud

Väljund:

```
A --> (A--1-->B)(A--3-->C)
B --> (B--1-->A)(B--2-->C)(B--1-->E)
C --> (C--3-->A)(C--2-->B)(C--6-->D)
D --> (D--6-->C)(D--3-->E)(D--1-->F)
E --> (E--1-->B)(E--3-->D)(E--5-->F)
F --> (F--1-->D)(F--5-->E)
```

Joonis 7: Graafi tipud ja nendest väljuvad kaared koos kaarepikkustega

Alltoodud käsuridadega arvutatakse lühim tee graafi alguspunktiks määratud punktist A lõpp-punktiks määratud punkti F.

```
public void run() {
    Graph g = new Graph( s: "G");

    initSampleGraph(g);

    g.shortestPathsFrom(g.getStart());

    g.displayShortestPath();
```

Joonis 8: Käsud näidisgraafi lühima teekonna arvutamiseks ja kuvamiseks

Programmi käivitamisel kuvatakse järgmine tee:

```
Shortest path from A to F with cumulative and total distance:

PATH:   A  -->  B  -->  E  -->  D  -->  F
DIST:   0       1       2       5       6

TOTAL:  6
```

Joonis 9: Näidisgraafi punktist A punkti F arvutatud lühim tee ja teepikkus.

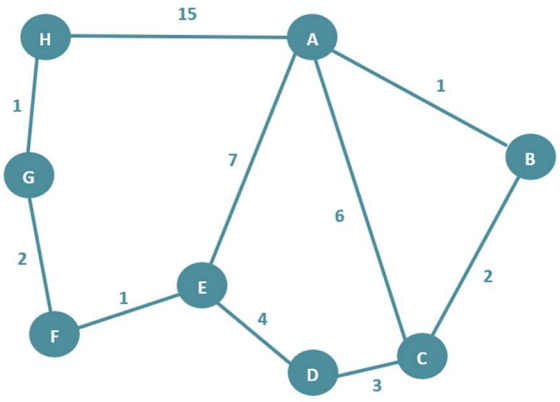
Testisin ka teiste punktide vahelise tee arvutamist. Näiteks punkti C ja E vaheline lühim tee on järgmine:

```
Shortest path from C to E with cumulative and total distance:

PATH:   C  -->  B  -->  E
DIST:   0       2       3

TOTAL:  3
```

Joonis 10: Näidisgraafi punktist C punkti E arvutatud lühim tee ja teepikkus

Näidisgraaf 2

Testisin meetodite toimimist ka järgmise joonisel kujutatud graafi peal. Näidisgraafi loomiseks vajalikud etapid on kirjas GraphTask meetodis *initSampleGraph2* (vt Lisa).



Joonis 11: Näidisgraaf 2

Sisend:

```
public void run() {

    //Sample graph 2
    Graph g = new Graph( s: "G");
    initSampleGraph2(g);
    g.displayGraph();
    g.shortestPathsFrom(g.getStart());
    g.displayShortestPath();
```

Joonis 12: Näidisgraafi 2 loomise ja lühima tee leidmise sisend.

Väljund:

```
A --> (A--1-->B)(A--6-->C)(A--7-->E)(A--15-->H)
B --> (B--1-->A)(B--2-->C)
C --> (C--6-->A)(C--2-->B)(C--3-->D)
D --> (D--3-->C)(D--4-->E)
E --> (E--7-->A)(E--4-->D)(E--1-->F)
F --> (F--2-->G)(F--1-->E)
G --> (G--2-->F)(G--1-->H)
H --> (H--1-->G)(H--15-->A)

Shortest path from A to H with cumulative and total distance:

PATH:   A  -->  E  -->  F  -->  G  -->  H
DIST:   0       7       8       10      11

TOTAL:  11
```

Joonis 13: Näidisgraafi 2 trükikuju ja lühim tee punktist A punkti H

Tühigraaf

Testisin ka tühigraafi, millele ei ole määratud algus- ega lõpp-punkti:

```
public void run() {
    Graph g = new Graph( s: "G");

    System.out.println(g);

    g.shortestPathsFrom(g.getStart());

    g.displayShortestPath();
```

Joonis 14: Tühigraafi loomise käsud

Tühigraafi trükikujuks on graafi nimi ning lühima arvutamisel ja kuvamisel ilmub tekst, et graaf on tühi.

```
G

Graph G is empty
```

Joonis 15: Tühigraafi trükikuju ja lühima tee arvutamisel saadud vastus

Automaatselt genereeritud väike graaf

Testisin lühima tee arvutamise ja kuvamise meetodeid automaatselt genereeritud väikeste graafidega:

Sisend:

```
public void run() {

    //Automatically generated graph small graph
    Graph g = new Graph( s: "G");
    g.createRandomGraph( vNum: 6,  eNum: 9,  maxLen: 10);
    System.out.println(g);
    g.shortestPathsFrom(g.getStart());
    g.displayShortestPath();
```

Joonis 16: Automaatselt genereeritud väikese graafi sisend

Väljund:

```
G
v1 -->(v1--4-->v4)(v1--3-->v2)(v1--3-->v3)
v2 -->(v2--3-->v1)(v2--9-->v4)(v2--4-->v3)
v3 -->(v3--8-->v6)(v3--3-->v1)(v3--4-->v2)(v3--4-->v5)
v4 -->(v4--4-->v1)(v4--9-->v2)(v4--9-->v5)
v5 -->(v5--4-->v3)(v5--9-->v4)(v5--9-->v6)
v6 -->(v6--8-->v3)(v6--9-->v5)


Shortest path from v1 to v6 with cumulative and total distance:

PATH:   v1  -->  v3  -->  v6
DIST:   0        3        11

TOTAL:  11
```

Joonis 17: Automaatselt genereeritud väikese graafi väljund

Automaatselt genereeritud suur graaf

Testisin meetodit ka automaatselt genereerides suuri graafe, kus tippude arv ulatub 2000-ni:

Sisend:

```
public void run() {

    //Automatically generated graph big graph
    Graph g = new Graph( s: "G");
    g.createRandomGraph( vNum: 2000,  eNum: 2002,  maxLen: 10);
    g.shortestPathsFrom(g.getStart());
    g.displayShortestPath();
```

Joonis 18: Automaatselt genereeritud graafi lühima tee arvutamise käsud

Väljund:

```
Shortest path from v1 to v2000 with cumulative and total distance:

PATH:   v1  -->  v1753  -->  v1873  -->  v1940  -->  v1948  -->  v1977  -->  v2000
DIST:   0           7           13          22          22          28          31

TOTAL:  31
```

Joonis 19: Automaatselt genereeritud graafi lühim tee

Kasutatud allikad

1. “Algoritmid ja andmestruktuurid konspekt,” Jaanus Pöial, TalTech 2020, https://enos.itcollege.ee/~jpoial/algoritmid/graafid.html
2. Joonised 1-3 - https://towardsdatascience.com/graph-theory-basic-properties-955fe2f61914
3. Dijkstra Shortest Path Algorithm in Java. – https://www.baeldung.com/java-dijkstra

Lisad

Programmi kood:

```
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();
   }

   public void initYourGraph(Graph g) {

      //INITIATE YOUR GRAPH HERE!
      //Use variable and object names that suit your solution

      // 1. CREATE VERTICES
      //    Example:
      //    Vertex a = new Vertex("A",null, null);

      // 2. SET START AND FINISH
      //    Example:
      //    g.start = a;
      //    g.finish = f;

      // 3. SET GRAPH VERTEXES
      //    Example:
      //    g.setVertexArray(new Vertex[]{a, b, c, d, e, f>});

      // 4. CREATE ARC PAIRS, ADD ARC WEIGHT
      //    Example:
      //    Arc ab = g.createArc("A", a, b, 1);
      //    ab.setTargetVertex(b);
      //    Arc ba = g.createArc("B", b, a, 1);
      //    ba.setTargetVertex(a);

      // 5. CONNECT ARCS WITH VERTICES
      //    Example:
      //    a.addArc(ab);
      //    a.setFirstArc(ab);
      //    a.addArc(ac);

   }
```

```
public void initSampleGraph(Graph g) {
   Vertex a = new Vertex("A",null, null);
   Vertex b = new Vertex("B",null, null);
   Vertex c = new Vertex("C",null, null);
   Vertex d = new Vertex("D",null, null);
   Vertex e = new Vertex("E",null, null);
   Vertex f = new Vertex("F",null, null);

   g.start = c;
   g.finish = e;
   g.setVertexArray(new Vertex[]{a,b,c,d,e,f});

   // Create Arc pairs to represent Edges
   Arc ab = g.createArc("A", a, b, 1);
   ab.setTargetVertex(b);
   Arc ba = g.createArc("B", b, a, 1);
   ba.setTargetVertex(a);

   Arc ac = g.createArc("A", a, c, 3);
   ac.setTargetVertex(c);
   Arc ca = g.createArc("C", c, a, 3);
   ca.setTargetVertex(a);

   Arc bc = g.createArc("B", b, c, 2);
   bc.setTargetVertex(c);
   Arc cb = g.createArc("C", c, b, 2);
   cb.setTargetVertex(b);

   Arc cd = g.createArc("C", c, d, 6);
   cd.setTargetVertex(d);
   Arc dc = g.createArc("D", d, c, 6);
   dc.setTargetVertex(c);

   Arc be = g.createArc("B", b, e, 1);
   be.setTargetVertex(e);
   Arc eb = g.createArc("E", e, b, 1);
   eb.setTargetVertex(b);

   Arc de = g.createArc("D", d, e, 3);
   de.setTargetVertex(e);
   Arc ed = g.createArc("E", e, d, 3);
   ed.setTargetVertex(d);

   Arc df = g.createArc("D", d, f, 1);
   df.setTargetVertex(f);
   Arc fd = g.createArc("F", f, d, 1);
   fd.setTargetVertex(d);

   Arc ef = g.createArc("E", e, f, 5);
   ef.setTargetVertex(f);
   Arc fe = g.createArc("F", f, e, 5);
   fe.setTargetVertex(e);

   a.addArc(ab);
   a.setFirstArc(ab);
   a.addArc(ac);
```

```
    b.addArc(ba);
    b.addArc(bc);
    b.setFirstArc(bc);
    b.addArc(be);

    c.addArc(ca);
    c.addArc(cb);
    c.addArc(cd);
    c.setFirstArc(cd);

    d.addArc(dc);
    d.addArc(de);
    d.setFirstArc(de);
    d.addArc(df);

    e.addArc(eb);
    e.addArc(ed);
    e.setFirstArc(ef);
    e.addArc(ef);

    f.addArc(fd);
    f.addArc(fe);

}

public void initSampleGraph2(Graph graph) {

    Vertex a = new Vertex("A",null, null);
    Vertex b = new Vertex("B",null, null);
    Vertex c = new Vertex("C",null, null);
    Vertex d = new Vertex("D",null, null);
    Vertex e = new Vertex("E",null, null);
    Vertex f = new Vertex("F",null, null);
    Vertex g = new Vertex("G",null, null);
    Vertex h = new Vertex("H",null, null);

    graph.start = a;
    graph.finish = h;
    graph.setVertexArray(new Vertex[]{a,b,c,d,e,f,g,h});

    // Create Arc pairs to represent Edges
    Arc ab = graph.createArc("A", a, b, 1);
    ab.setTargetVertex(b);
    Arc ba = graph.createArc("B", b, a, 1);
    ba.setTargetVertex(a);

    Arc ac = graph.createArc("A", a, c, 6);
    ac.setTargetVertex(c);
    Arc ca = graph.createArc("C", c, a, 6);
    ca.setTargetVertex(a);

    Arc bc = graph.createArc("B", b, c, 2);
    bc.setTargetVertex(c);
    Arc cb = graph.createArc("C", c, b, 2);
    cb.setTargetVertex(b);

    Arc cd = graph.createArc("C", c, d, 3);
```

```
cd.setTargetVertex(d);
Arc dc = graph.createArc("D", d, c, 3);
dc.setTargetVertex(c);

Arc de = graph.createArc("D", d, e, 4);
de.setTargetVertex(e);
Arc ed = graph.createArc("E", e, d, 4);
ed.setTargetVertex(d);

Arc ae = graph.createArc("A", a, e, 7);
ae.setTargetVertex(e);
Arc ea = graph.createArc("E", e, a, 7);
ea.setTargetVertex(a);

Arc ef = graph.createArc("E", e, f, 1);
ef.setTargetVertex(f);
Arc fe = graph.createArc("F", f, e, 1);
fe.setTargetVertex(e);

Arc gf = graph.createArc("G", g, f, 2);
gf.setTargetVertex(f);
Arc fg = graph.createArc("F", f, g, 2);
fg.setTargetVertex(g);

Arc gh = graph.createArc("G", g, h, 1);
gh.setTargetVertex(h);
Arc hg = graph.createArc("H", h, g, 1);
hg.setTargetVertex(g);

Arc ha = graph.createArc("H", h, a, 15);
ha.setTargetVertex(a);
Arc ah = graph.createArc("A", a, h, 15);
ah.setTargetVertex(h);

a.addArc(ab);
a.setFirstArc(ab);
a.addArc(ac);
a.addArc(ae);
a.addArc(ah);

b.addArc(ba);
b.addArc(bc);
b.setFirstArc(bc);

c.addArc(ca);
c.addArc(cb);
c.addArc(cd);
c.setFirstArc(cd);

d.addArc(dc);
d.addArc(de);
d.setFirstArc(de);

e.addArc(ea);
e.addArc(ed);
e.setFirstArc(ef);
e.addArc(ef);
```

```
    f.addArc(fg);
    f.setFirstArc(fg);
    f.addArc(fe);

    g.addArc(gf);
    g.addArc(gh);
    g.setFirstArc(gh);

    h.addArc(hg);
    h.addArc(ha);
    h.setFirstArc(ha);

    ab.setNextArc(bc);
    bc.setNextArc(cd);
    cd.setNextArc(de);
    de.setNextArc(ef);
    ef.setNextArc(fg);
    fg.setNextArc(gh);
    gh.setNextArc(ha);
    ha.setNextArc(ae);
    ae.setNextArc(ac);
}


/**
 * Actual main method to run examples and everything.
 */
public void run() {

    //Sample graph 1
    Graph g = new Graph("G");
    initSampleGraph(g);
    g.displayGraph();
    g.shortestPathsFrom(g.getStart());
    g.displayShortestPath();

    //Sample graph 2
    /*Graph g = new Graph("G");
    initSampleGraph2(g);
    g.displayGraph();
    g.shortestPathsFrom(g.getStart());
    g.displayShortestPath();*/

    //Automatically generated graph small graph
    /*Graph g = new Graph("G");
    g.createRandomGraph(6, 9, 10);
    System.out.println(g);
    g.shortestPathsFrom(g.getStart());
    g.displayShortestPath();*/

    //Automatically generated graph big graph
    /*Graph g = new Graph("G");
    g.createRandomGraph(2000, 2002, 10);
    g.shortestPathsFrom(g.getStart());
    g.displayShortestPath();*/

}
```

```
/**
 * Graphs are objects that contain all the Vertices and Arcs needed
 * to represent all the possible paths between different destinations.
 * All the methods for finding the shortest path and displaying it are
 * contained in the
 * Graph class.
 */
class Graph {

   private final String name;
   private Vertex start;
   private Vertex finish;
   private Vertex[] vertexArray;

   Graph(String id, Vertex start, Vertex finish) {
      this.name = id;
      this.start = start;
      this.finish = finish;
   }

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

   @Override
   public String toString() {
      String nl = System.getProperty ("line.separator");
      StringBuilder sb = new StringBuilder(nl);
      sb.append (name);
      sb.append (nl);
      Vertex v = start;
      while (v != null) {
         sb.append (v.toString());
         sb.append (" -->");
         Arc a = v.first;
         while (a != null) {
            sb.append(a);
            a = a.next;
         }
         sb.append (nl);
         v = v.previous;
      }
      return sb.toString();
   }

   public Vertex getStart() {
      return this.start;
   }

   public Vertex[] getVertexArray() {
      return vertexArray;
   }

   public void setVertexArray(Vertex[] vertexArray) {
      this.vertexArray = vertexArray;
   }

   public boolean contains(Vertex v) {
```

```
      for (Vertex each : getVertexArray()
      ) {
         if (each.getName().equals(v.getName())) {
            return true;
         }
      }
      return false;
   }

   public String reverse(String name) {
      StringBuilder sb = new StringBuilder();
      sb.append(name);
      return sb.reverse().toString();
   }

   public Vertex createVertex(String vid) {
      Vertex res = new Vertex(vid);
      res.previous = start;
      start = res;
      return res;
   }

   public String repeat(String s, int n) {
      StringBuilder sb = new StringBuilder();
      for (int i = 0; i < n; i++) {
         sb.append(s);
      }
      return sb.toString();
   }

   public void displayGraph() {
      for (Vertex v : vertexArray) {
         StringBuilder sb = new StringBuilder();
         sb.append(v);
         sb.append(" --> ");
         for (Arc a : v.getArcs()) {
            sb.append(a);
         }
         System.out.println(sb);
      }
   }

   /**
    * Shortest paths from a given vertex. Uses Dijkstra's algorithm.
    * For each vertex vInfo is length of shortest path from given
    * start s and vObject is previous vertex from s to this vertex.
    *
    * @param s start vertex
    */
   public void shortestPathsFrom(Vertex s) {
      if (getVertexArray() == null) return;
      if ((!contains(s)) // | (s.getGraph() != this)
         throw new RuntimeException("wrong argument to Dijkstra!!");
      int INFINITY = Integer.MAX_VALUE / 4; // big enough!!!

      for (Vertex v : getVertexArray()) {
         v.setVDistFromStart(INFINITY);
         v.setPreviousVertex(null);
```

```
    }

    s.setVDistFromStart(0);
    List vq = Collections.synchronizedList(new LinkedList());
    vq.add(s);
    while (vq.size() > 0) {
       int minLen = INFINITY;
       Vertex minVert = null;
       for (Object vertex : vq) {
          Vertex v = (Vertex) vertex;
          if (v.getVDistFromStart() < minLen) {
             minVert = v;
             minLen = v.getVDistFromStart();
          }
       }

       if (minVert == null)
          throw new RuntimeException("error in Dijkstra!!");
       if (vq.remove(minVert)) {
          // minimal element removed from vq
       } else
          throw new RuntimeException("error in Dijkstra!!");

       for (Arc arc : minVert.getArcs()) {
          int newLen = minLen + arc.getArcLength();
          Vertex to = arc.getTargetVertex();
          if (to.getVDistFromStart() == INFINITY) {
             vq.add(to);
          }
          if (newLen < to.getVDistFromStart()) {
             to.setVDistFromStart(newLen);
             to.setPreviousVertex(minVert);
          }
       }
    }
} // end of shortestPathsFrom()

public void displayShortestPath() {

    StringBuilder path = new StringBuilder();
    String nl = System.getProperty ("line.separator");

    if (start == null || finish == null) {
       System.out.println("Graph " + name + " is empty");
       return;
    }

    Vertex v = finish;
    path.append(reverse(finish.getName()));

    for (int i = 0; i < vertexArray.length; i++) {
       path.append("  >--  ");
       path.append(reverse(v.getPreviousVertex().getName()));
       if (v.getPreviousVertex().equals(start)) {
          break;
       }
       v = v.getPreviousVertex();
    }
```

```
    StringBuilder distance = new StringBuilder();
    v = finish;

    for (int i = 0; i < vertexArray.length; i++) {
       int spaceCount = 7 - (String.valueOf(v.getVDistFromStart()).length());
       distance.append(repeat(" ", spaceCount));
       int spaceCount2 = v.getName().length();
       distance.append(repeat(" ", spaceCount2));
       distance.append(reverse(String.valueOf(v.getVDistFromStart())));
       if (v.getPreviousVertex().equals(start)) {
          break;
       }
       v = v.getPreviousVertex();
    }
    int spaceCount = 7 - (String.valueOf(v.getVDistFromStart()).length());
    distance.append(repeat(" ", spaceCount));
    int spaceCount2 = v.getName().length();
    distance.append(repeat(" ", spaceCount2));
    distance.append(start.getVDistFromStart());

    path = path.reverse();
    path.append(nl);
    distance = distance.reverse();

    StringBuilder result;
    result = new StringBuilder();
    result.append(nl);
    result.append("Shortest path from ");
    result.append(start);
    result.append(" to ");
    result.append(finish);
    result.append(" with cumulative and total distance:");
    result.append(nl);
    result.append(nl);
    result.append("PATH:");
    result.append("\t");
    result.append(path);
    result.append("DIST:");
    result.append("\t");
    result.append(distance);
    result.append(nl);
    result.append(nl);
    result.append("TOTAL:");
    result.append("\t");
    result.append(finish.getVDistFromStart());

    System.out.println(result.toString());
}

public Arc createArc(String aid, Vertex from, Vertex to, int length) {
    Arc res = new Arc(aid);
    res.next = from.first;
    from.first = res;
    res.target = to;
    res.setArcLength(length);
    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, int maxLength) {

   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);
         int length = (int) (Math.random() * maxLength);

         Arc a = createArc(varray[vnr].toString(), varray[vnr],
                 varray[i], length);
         varray[vnr].addArc(a);

         Arc b = createArc(varray[i].toString(), varray[i],
                 varray[vnr], length);
         varray[i].addArc(b);
      }
   }

   finish = varray[0];
   start = varray[n-1];

   setVertexArray(varray);
}

/**
 * Create an adjacency matrix of this graph.
 * Side effect: corrupts info fields in the graph
 *
 * @return adjacency matrix
 */
public int[][] createAdjMatrix() {
   int info = 0;
   Vertex v = start;
   while (v != null) {
      v.setVDistFromStart(info++);
      v = v.getPreviousVertex();
   }
   int[][] res = new int[info][info];
   v = start;
   while (v != null) {
      int i = v.getVDistFromStart();
      Arc a = v.first;
      while (a != null) {
         int j = a.getTargetVertex().getVDistFromStart();
         res[i][j]++;
         a = a.next;
```

```
            }
            v = v.getPreviousVertex();
        }
        return res;
    }

    /**
     * Create a connected simple (undirected, no loops, no multiple
     * arcs) random graph with n vertices and m edges.
     *
     * @param vNum number of vertices
     * @param eNum    number of edges
     * @param maxLen        maximum length of the arcs
     */
    public void createRandomGraph(int vNum, int eNum, int maxLen) {
        if (vNum <= 0)
            return;
        if (vNum > 2500)
            throw new IllegalArgumentException("Too many vertices: " + vNum);
        if (eNum < vNum - 1 || eNum > vNum * (vNum - 1) / 2)
            throw new IllegalArgumentException
                    ("Impossible number of edges: " + eNum);
        start = null;
        createRandomTree(vNum, maxLen); // n-1 edges created here
        Vertex[] vert = new Vertex[vNum];
        Vertex v = start;
        int c = 0;
        while (v != null) {
            vert[c++] = v;
            v = v.getPreviousVertex();
        }
        int[][] connected = createAdjMatrix();
        int edgeCount = eNum - vNum + 1;  // remaining edges
        while (edgeCount > 0) {
            int i = (int) (Math.random() * vNum);  // random start
            int j = (int) (Math.random() * vNum);  // 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];

            int randomWeight = (int) (Math.random() * maxLen);

            Arc a = createArc(vi.toString(), vi, vj, randomWeight);
            connected[i][j] = 1;
            vi.addArc(a);

            Arc b = createArc(vj.toString(), vj, vi, randomWeight);
            connected[j][i] = 1;
            vj.addArc(b);

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

```java
/**
 * Class Vertex for representing all the milestones along the path
 * from start to finish. Vertex class holds all the Arcs that derive
 * from it and connect it to the other vertices.
 */
class Vertex {

   private final String name;
   private Vertex previous;
   private Arc first;
   private int vDistFromStart = 0;
   private final ArrayList<Arc> arcs = new ArrayList<>();

   Vertex(String s, Vertex v, Arc e) {
      name = s;
      previous = v;
      first = e;
   }

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

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

   public String getName() {
      return this.name;
   }

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

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

   public int getVDistFromStart() {
      return this.vDistFromStart;
   }

   public void setPreviousVertex(Object o) {
      this.previous = (Vertex) o;
   }

   public Vertex getPreviousVertex() {
      return this.previous;
   }

   public ArrayList<Arc> getArcs() {
      return this.arcs;
   }

   public boolean containsArc(Arc arc) {
      for (Arc a : arcs) {
```

```
            if (a.name.equals(arc.name) &&
a.getTargetVertex().equals(arc.getTargetVertex())) {
                return true;
            }
         }
         return false;
      }

      public void addArc(Arc arc) {
         if (!containsArc(arc)) {
            this.arcs.add(arc);
         }
      }
   }


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

      private final String name;
      private Vertex target;
      private Arc next;
      private int length = 0;

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

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

      @Override
      public String toString() {
         StringBuilder sb;
         sb = new StringBuilder();
         sb.append("(");
         sb.append(name);
         sb.append("--");
         sb.append(length);
         sb.append("-->");
         sb.append(target);
         sb.append(")");

         return sb.toString();
      }

      public Vertex getTargetVertex() {
         return this.target;
      }

      public void setTargetVertex(Vertex target) {
         this.target = target;
```

```
        }

        public int getArcLength() {
            return this.length;
        }

        public void setArcLength(int length) {
            this.length = length;
        }

        public void setNextArc(Arc arc) {
            this.next = arc;
        }

        public Arc getNextArc() {
            return this.next;
        }

    }

}
```