TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Kahe tipu vahelise pikima tee leidmine kaaludega tsükliteta orienteeritud graafides

Individuaaltöö aines "Algoritmid ja andmestruktuurid"

Koostaja: Asko Laaniste

Eriala: IT-süsteemide arendus

Rühm: 192863IADB

Juhendaja: Jaanus Põial

Tallinn 2021

1 Ülesande püstitus

Antud töö eesmärgiks oli valmis kirjutada algoritm leidmaks pikimat teed ilma kaaludega tükliteta orienteeritud graafis. Ülesande idee on võetud raamatust „*Data Structures and Algorithms in Java*“, ülesandest C-13.19[1].

Graaf on kõige üldisemal tasemel seoste kujutamise viis. Graaf koosneb tippudest ja servadest: Tipud kujutavad mingeid eraldiseisvaid olemeid, kas siis abstraktseid nagu programmi alammoodulid või füüsilisi nagu näiteks inimesed, linna vms.[2,3] Servad kujutavad endast mingit kahe tipu vahelist seost. Näiteks kui meil on graaf, millel on tippudeks erinevad inimesed, siis nendevahelisi sõprussuhteid saab servadega näidata. Kui kaks inimest teavad üksteist siis neid kahte inimest kujutavate tippude vahele saame tõmmata serva.

Joonis 1: Kahe sõbra vaheline seos.

Orienteeritud graafis on servade asemel aga hoopis kaared. Kaar näitab suunatud suhet kahe tipu vahel. Näiteks samamoodi inimeste graafil saab suunatud graafi panna isikult A isikule B nõnda, et isik A teab isikut B, ent isik B ei tea isikut A. Reaalelulise näitena saame tuua kuulsad inimesed, kes vaevalt teavad suuremat osa inimesi, kes kõik on kuulsaid inimesi näiteks telesaadetes näinud.

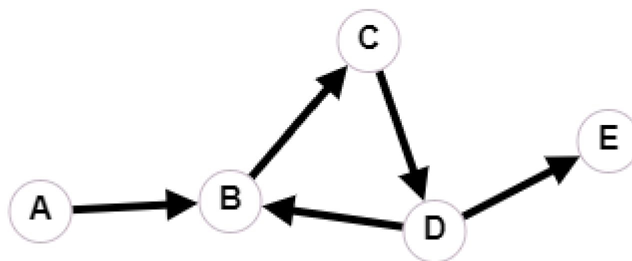
Joonis 2: Kuulsa inimese ja talle tundmatu inimese vaheline seos.

Antud näide aga ammendab ennast, kuna antud ülesande on vajalik tsükliteta graafe kasutada. Tsükliga on orienteeritud graaf siis, kui ühe või enama tipu juurest saab leida sellise kaarte hulga, mis viib tagasi sellesama tipuni. Võtame näiteks eelmise näite. Kindlasti teab kuulsus kedagi, kes teab ka teda. Näiteks tema parim sõber. Seega saab

nendevahelist suhet iseloomustada kahe kaarega. Samuti teab kuulus inimene ka ise ennast. Sellist suhet nimetatakse silmuseks.

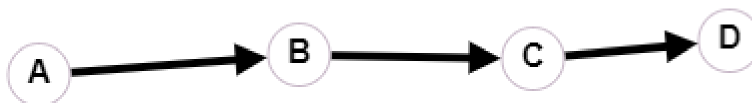
Joonis 3: Kuulsa inimese ja tema sõbra vaheline seos.

Antud ülesandes ei tohi graafis tsükleid esineda, kuna sellisel juhul võib vabalt esineda juhtum, kus pikima tee leidmine osutub võimatuks, kuna võime jääda tsüklisse lõputult ketrama ja seeläbi teekonda lõpmatult pikaks muutma, ilma, et kunagi kohale jõuaksime.



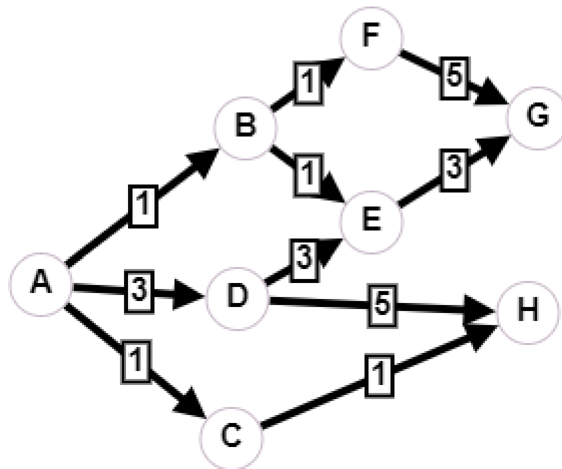
Joonis 4: Lõputu tsükliga teekond punktist A punkti E.

Järelikult tükliteta[4] orienteeritud graaf on näiteks teadusprojekti edukaks lõpetamiseks vajalike sammude struktuur.[5] Üks samm viib järgmise sammuni, mis omakorda võib avada mitu erinevat suunda, ent tagasi eelnevate läbitud sammude peale ei suuna enam ükski kaar, kuna juba läbitud rada pole enam mõtet läbida, tulemused on juba olemas.



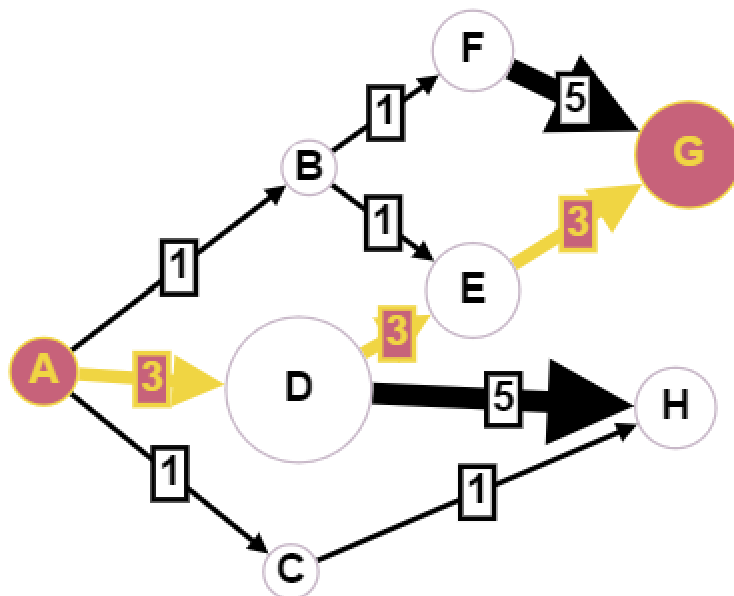
Joonis 5: Teadusprojekti teostumiseks vajalikud sammud ja nende järjekord.

Kui nüüd igale teadusprojekti kaarele lisada orienteeruv ajakulu, mis on vajalik, et jõuda järgmise projekti sammuni, saamegi kaaludega tsükliteta orienteeritud graafi. Oletame, et projekti täitmiseks on mitu erinevat võimalust ning ka mitu võimalikku tulemust, nagu on tihti vaja arvesse võtta mahukatel projektidel, et maandada riske. Sel juhul on meil võimalik sama tulemuseni jõuda mitut eri teed pidi ning kahe vaheetapi vahelised sammud ei ole sama kaaluga, vaid mõnele võib kuluda oluliselt enam aega kui teistele.



Joonis 6: Erinevad teed sama eesmärgini. [6]

Antud ülesande kontekstis saaksime leida näiteks antud projekti läbimiseks kõige ajamahukamad sammud ning juba ennetavalt nendele sammudele enim rõhku panna, et uurida juba varakult välja nende sammude mõttekus. Projekti teostatavuse hindajana võib samuti graafipõhine lähenemine anda väärtuslikku infot projekti teostatavuse usutavusse.



Joonis 7: Kõige ajamahukam teekond.

2 Lahenduse kirjeldus

2.1 Tipud ja kaared

Programmi seisukohalt on tipud ja kaared lahendatud kahe klassiga nimega *Vertex* (tipp) ja *Arc* (kaar). Igal tipul on viide tema naabertipule, kuniks nad otsa saavad ja siis on viide tühi. Tipu sees on kasutusel ka abimuutujad *topOrderIndex* ja *visited*, mis on vajalikud hilisemate otsingualgoritmide täitmiseks. Samuti on igal tipul viide tema esimesele kaarele (*first*). Kaarte järjekord ei ole kuidagi iseloomustav kaare olulisust, vaid on ainult tehnilise lahenduse tõttu vajalik.

Iga kaar omab viidet tipule, kuhu ta suubub (*target*) ja järgmisele naaberkaarele (*next*), kes suubub samuti välja samast tipust kust käesolev kaar ise. Kui naaberkaari rohkem pole, on muutuja tühi. Kaarel on ka kaal (*weight*), mille alusel otsustame hiljem teekonna pikkuse.

2.1.1 Koodinäide: Tipud koodis

```
/**
 * Vertex represent one intersection or endpoint in graph.
 * Vertex has helper attributes "topological order index" and
 * "visited" status for use in Graph functions.
 */
class Vertex {

    private final String id;
    private Vertex next;
    private Arc first;
    private int topOrderIndex = -1;
    private boolean visited = false;

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

    Vertex(String s) { this(s, v: null, e: null); }

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

}
```

2.1.2 Koodinäide: Kaared koodis

```java
/**
 * Arc represents one arrow in the graph.
 * Arcs can have weights to represent distances between
 * two vertices (default weight is 1).
 */
class Arc {

    private final String id;
    private Vertex target;
    private Arc next;
    private final int weight;

    Arc(String s, Vertex v, Arc a) {
        id = s;
        target = v;
        next = a;
        this.weight = 1;
    }

    Arc(String s, int weight) {
        id = s;
        target = null;
        next = null;
        this.weight = weight;
    }

    Arc(String s) { this(s, v: null, a: null); }

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

}
```

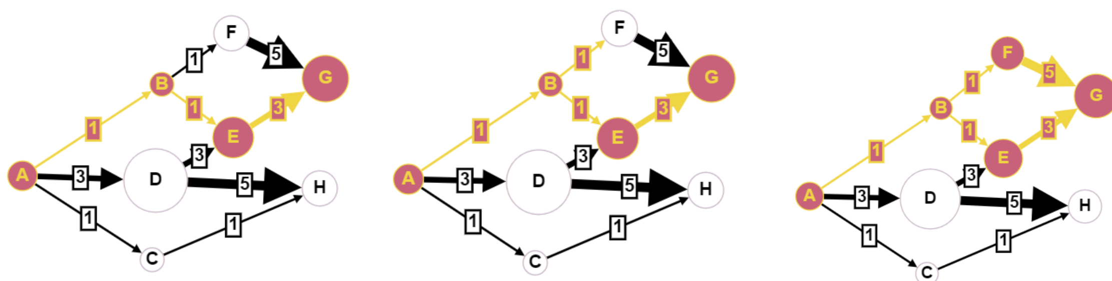
2.2 Lahenduse üldine struktuur

Edaspidi nimetame alguspunktiks graafi tippu, millest alustame pikima tee otsingut ja lõpp-punktiks graafi tippu, milleni otsime pikimat teed. Lahenduseks on algoritm, mis leiab pikima teekonna kaarte kaale arvestades alguspunktist lõpp-punkti. Antud lahendus on kokku võetav ülevaatlikult järgnevate punktidega:

1) Esmalt sügavuti otsing alguspunktist leidmaks lõpp-punktini viivaid kõikvõimalikke teid.
2) Seejärel topoloogilise järjestuse koostamine otsingu tulemuste alusel.
3) Seejärel topoloogilises järjestuses kõikidele tippudele teekonna pikkuse leidmine alguspunktist.
4) Lõpp-punkti pikima tee leidmine, alustades lõpp-punktist ja tagurpidi tagasi sammudes kuni alguspunktini.

2.3 Sügavuti otsing

Sügavuti otsing valib alguspunktist suvalise väljuva kaare ja jätkab iga järgneva tipuga samamoodi, kuni ei leita enam ühtegi väljuvat kaart. Seejärel kerib algoritm tagasi eelneva tipuni ning otsib sealt edasi nii sügavuti kui võimalik.



Joonis 8: Sügavuti otsing.

Koodifragment sügavuti otsingu teostamiseks:

2.4 Topoloogiline järjestus

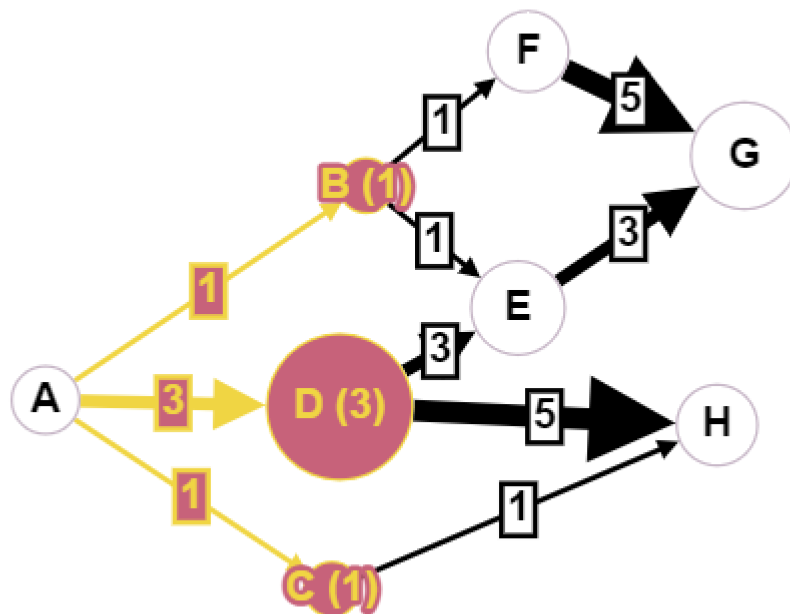
Topoloogiline järjestus on graafi tippude selline järjestus, et igasse eelnevasse tippu ei ole võimalik järgnevast tipust enam pääseda. Näiteks antud näitegraafi üks võimalik[7] topoloogiline järjestus oleks A, B, F, D, E, C, H. Topoloogiline järjestus lubab meil garanteerida seda, et iga järgnev tipp, mis selles järjestuses võetakse tuleb pärast

eelnevaid. See tingimus on vajalik, et tagada järgnevas sammus iga tipu stardipunktist kauguse mõõtmise korrektsus, kui liidame sihtpunkti enda kauguse lisaks kaare enda kaalule.[8]

2.5 Kõigi tippude kõige kaugema teekonna leidmine.

Alustades alguspunktist hakkame arvutama iga järgneva tipu kauguse alguspunktist. Selleks on appi võetud kaks abilist. Esiteks on igal tipul muutuja, mis hoiab endas vastava tipu topoloogilist järjekorranumbrit (*topOrderIndex*). Teiseks on võetud kasutusele topoloogilise järjekorra pikkune tühi loend, kuhu vastavalt järjekorranumbrile salvestada tipu kaugus alguspunktist (*distancesFromStart*). Võttes esimesena alguspunkti, vaatleme kõik temast väljuvaid kaari ning salvestame nende kaarte kaalud, mis on ühtlasi teekonna pikkuse mõõduks, kaare sihttipu järjekorranumbri alusel kauguste loendisse. Kui nõnda oleme talitanud kõikide kaartega, siis on meie loendis nüüd järgmised väärtused:

A	B	F	D	E	G	C	H
0	1		3			1	

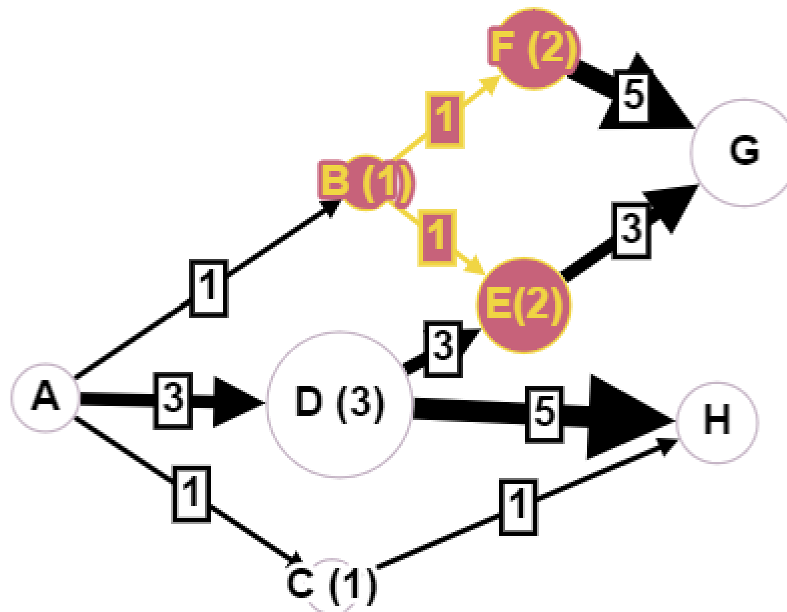


Joonis 9: Punktist A väljuvate kaart pikkuste mõõtmine.

Järgnevalt liigume topoloogilises järjekorras järgmise tipuni, milleks on B. Leiame temast väljuvad kaared ning uuendame iga kaare sihttipu kaugust alguspunktist vastavalt kaare

kaalule. Ent nüüd liidame juurde ka kaare algustipu enda kaalu, antud juhul B. Uus seis on järelikult:

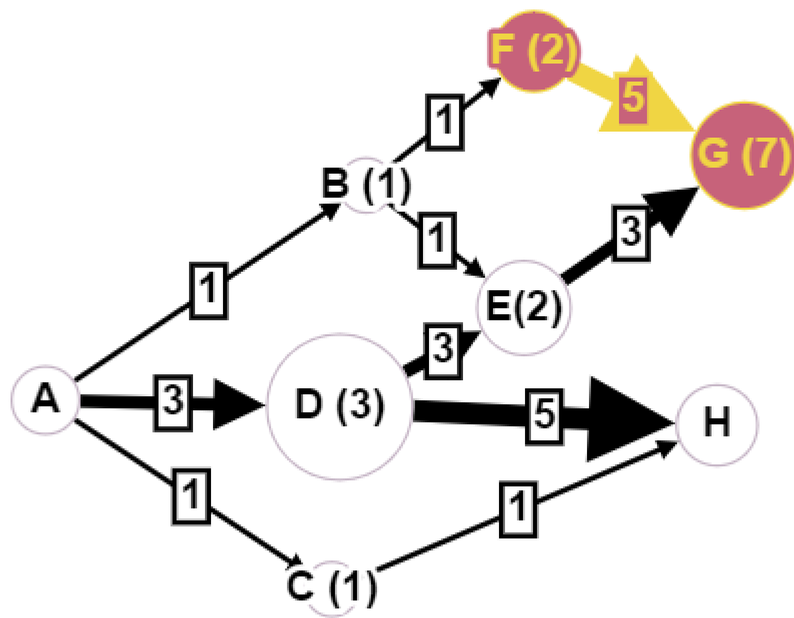
A	B	F	D	E	G	C	H
0	1	2	3	2		1	



Joonis 10: Järgnevate tippude kaugustele liidetakse eelneva tipu kaugus ning kaare kaal.

Järgmisena tuleb F, millega saame esimese hinnangu meie otsitavale lõpp-punkti kaugusele:

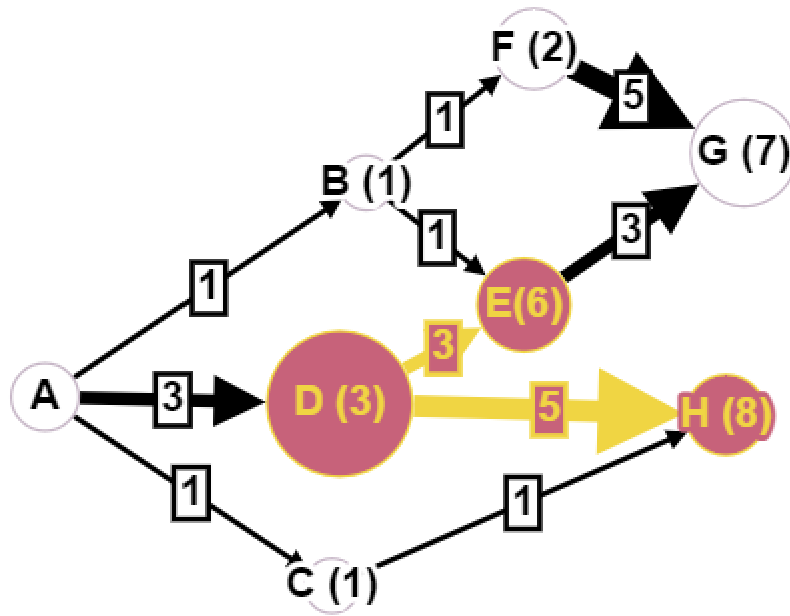
A	B	F	D	E	G	C	H
0	1	2	3	2	7	1	



Joonis 11: Esimene hinnang lõpp-punkti kaugusele.

Seejärel D: kuna läbi tipu D on E kaugus suurem kui see eelnevalt oli, siis uuendame kauguste loendis väärtust.

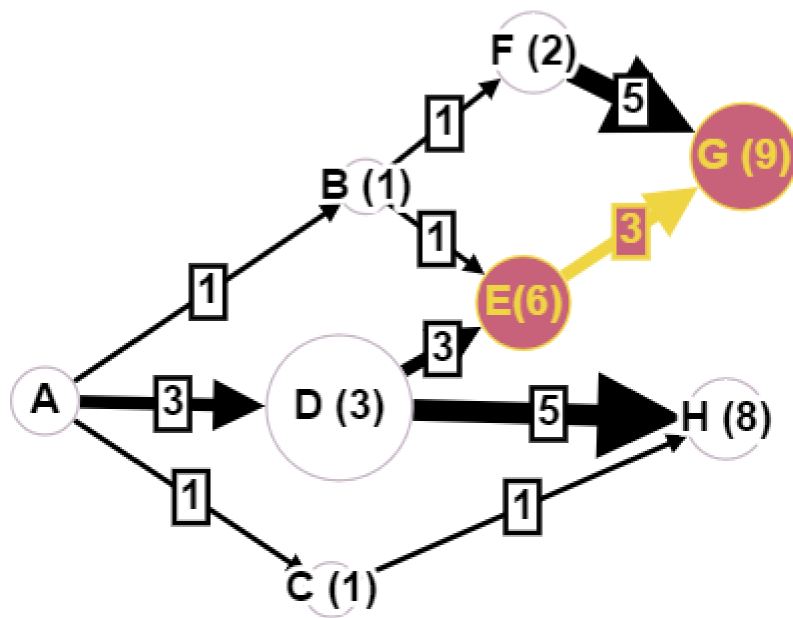
A	B	F	D	E	G	C	H
0	1	2	3	6	7	1	8



Joonis 12: Tipu E kaugus hinnang uueneb.

Seejärel E: taaskord saame pikema teekonna ja uuendame meie lõpp-punkti kauguse hinnangut.

A	B	F	D	E	G	C	H
0	1	2	3	6	9	1	8



Joonis 13: Tipu G kauguse hinnangu uueneb.

Järgnev on G, mis on ühtlasi meie lõpp-punkt ja vastavalt topoloogilisele järjestusele, ei suundu ükski järgnev punkt tagasi eelnevatesse punktidesse, sh ka meie lõpp-punkti ja võime lõpetada otsingu.

2.5.1 Koodinäide: Peamine meetod pikima tee leidmiseks.

```
/**
 * Find the longest path from start vertex to end vertex from the provided reverse topological order.
 * Start vertex is the last vertex in the provided reverse topological order.
 * Arc weights are used as distances between any two vertices.
 * In case of equal distances, differences are not made and the first to match is taken.
 *
 * @param topOrder reverse topological order with start vertex
 * @param end      target vertex to which the longest path is looked for
 * @return object with start and end vertices, longest path length and list of arcs that form the path.
 */
private LongestPath findLongestPath(List<Vertex> topOrder, Vertex end) {
    Integer[] distancesFromStart = new Integer[topOrder.size()];
    Vertex[] priorVertices = new Vertex[topOrder.size()];
    for (int i = topOrder.size() - 1; i > 0; i--) {
        Vertex source = topOrder.get(i);
        if (source == end) break;
        if (distancesFromStart[source.topOrderIndex] == null) distancesFromStart[source.topOrderIndex] = 0;
        updateDistancesFromSource(distancesFromStart, priorVertices, source);
    }
    Vertex start = topOrder.get(0);
    if (end.topOrderIndex == -1) return new LongestPath(start, end, pathLength: -1);
    List<Arc> longestPath = getLongestPathArcs(start, end, priorVertices);
    int longestPathLength = distancesFromStart[end.topOrderIndex];
    return new LongestPath(topOrder.get(topOrder.size() - 1), end, longestPathLength, longestPath);
}
```

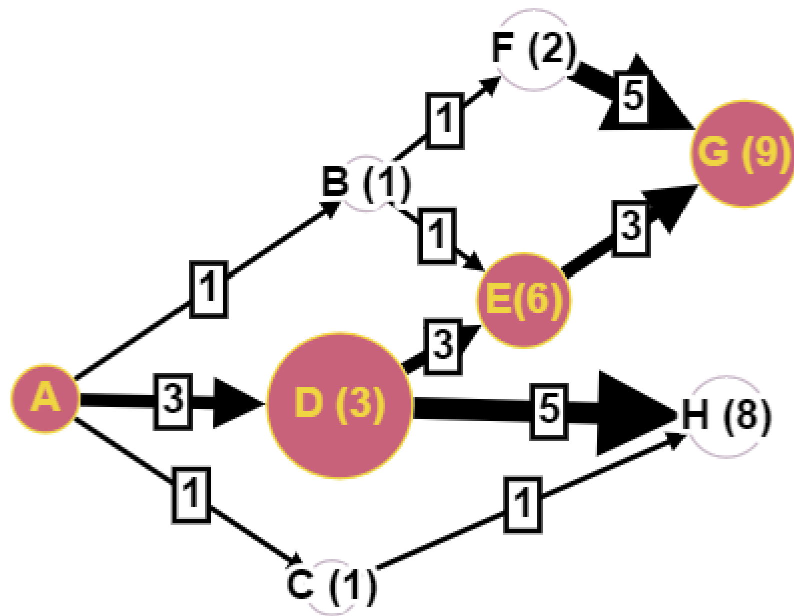
2.5.2 Koodinäide: Lähtetipust väljasuunduvate kaarte tippude kauguste uuendamine ning eellastipu salvestamine.

```
/**
 * Update the distances from source vertex to all other vertices its arcs are pointing to.
 *
 * @param distancesFromStart Container for updated distances.
 * @param priorVertices      Container for reconstructing the longest path.
 * @param source             Vertex whose children distances from start will be updated.
 */
private void updateDistancesFromSource(Integer[] distancesFromStart, Vertex[] priorVertices,
                                       Vertex source) {
    Arc nextArc = source.first;
    Vertex nextTarget;
    int arcWeight;
    while (nextArc != null) {
        nextTarget = nextArc.target;
        arcWeight = nextArc.weight;
        if (nextTarget.topOrderIndex == -1) {
            nextArc = nextArc.next;
            continue;
        }
        if (distancesFromStart[nextTarget.topOrderIndex] == null) {
            distancesFromStart[nextTarget.topOrderIndex] = 0;
        }
        int newDistance = distancesFromStart[source.topOrderIndex] + arcWeight;
        if (distancesFromStart[nextTarget.topOrderIndex] < newDistance) {
            distancesFromStart[nextTarget.topOrderIndex] = newDistance;
            priorVertices[nextTarget.topOrderIndex] = source;
        }
        nextArc = nextArc.next;
    }
}
```

2.6 Pikima teekonna leidmine

Eelmises punktis saime teada, milline on pikim teepikkus (9), ent nüüd on vaja ka teada läbi milliste kaart see tee kulgeb. Selleks on meil kasutusel teine loend, mis samamoodi hoiab tipu järjekorranumbri positsioonil viimatise tipu infot, kust tulnud kaar uuendas kaugushinnangut (*priorVertices*). Nõnda saab lõpp-punktist tagasi kerides leida viimatise tipu, kust väljus pikim kaar lõpp-punkti suunas ning samuti selle pikima kaare. Tagasisammumist tehakse seni, kuni ollakse jõutud alguspunktini. Seejärel salvestatakse pikima teekonna info objekti *LongestPath*, millelt saab küsida nii algus – kui lõpp-tipu infot, kui ka teekonna pikkust ja mis kaari teekond läbib.

A	B	F	**D**	**E**	**G**	C	H
0	1	2	3	6	9	1	8
NA	A	B	**A**	**D**	**E**	A	D



Joonis 14: Pikima teekonna tipud.

2.6.1 Koodinäide: Programmi töö tulemusel loodav objekt.

```java
/**
 * Container for the results of the longest path search.
 * Contains longest path length between start and end vertices, with also
 * list of arcs in topological order that form the path.
 */
class LongestPath {


    private final Vertex start;
    private final Vertex end;
    private final int pathLength;
    private final List<Arc> pathArcs;

    LongestPath(Vertex start, Vertex end, int pathLength, List<Arc> pathVertices) {
        this.start = start;
        this.end = end;
        this.pathLength = pathLength;
        this.pathArcs = pathVertices;
    }

    LongestPath(Vertex start, Vertex end, int pathLength) {
        this(start, end, pathLength,
                pathVertices: null);
    }

    public Vertex getStart() { return start; }

    public Vertex getEnd() { return end; }

    public int getPathLength() { return pathLength; }

    public List<Arc> getPathArcs() { return pathArcs; }

    @Override
    public String toString() {
        if (start == null && end == null) return "Path does not exist";
        if (pathLength == -1) return "Path does not exist from " + start + " to " + end;
        return "Longest path from " + start + " to " + end +
                " is with length " + pathLength + " and follows arcs:\n" + pathArcs;
    }
}
```

2.6.2 Koodinäide: Kaarte loendi leidmise meetodid.

```
/**
 * Get arcs list for the longest path from start vertex to end vertex.
 * Relies on the priorVertices array, where at the vertex topSortIndex position its
 * parent vertex is positioned.
 *
 * @param start          first vertex of the path
 * @param end            last vertex of the path
 * @param priorVertices Vertex array with prior vertices at topSortIndex position.
 * @return List of arcs in topological order that form the longest path from start to
 * end vertex.
 */
private List<Arc> getLongestPathArcs(Vertex start, Vertex end, Vertex[] priorVertices) {
    List<Arc> longestPath = new LinkedList<>();
    Vertex target = end;
    Vertex source = priorVertices[target.topOrderIndex];
    ;
    while (source != null) {
        Arc longestArc = getLongestArc(source, target);
        longestPath.add(longestArc);
        target = source;
        source = priorVertices[source.topOrderIndex];
        if (source == start) {
            break;
        }
    }
    Collections.reverse(longestPath);
    return longestPath;
}
```

```java
/**
 * Find the longest arc from source to target vertex.
 *
 * @param target vertex the arc has to point to
 * @param source vertex from where the arc points from
 * @return the Arc with the longest distance (weight)
 */
private Arc getLongestArc(Vertex source, Vertex target) {
    Arc nextArc = source.first;
    int longestDistance = 0;
    Arc longestArc = null;
    while (nextArc != null) {
        if (nextArc.target == target && nextArc.weight > longestDistance) {
            longestDistance = nextArc.weight;
            longestArc = nextArc;
        }
        nextArc = nextArc.next;
    }
    return longestArc;
}
```

3 Programmi kasutamisjuhend

Programmi käivitatakse kõigepealt luues uus graaf. Graafil on endal juhuslike või lihtsamate graafide genereerimise meetodid või tuleks defineerida end huvitav graaf eelnevalt ise. Seejärel anda graafi meetodile *findLongestPath()* alguspunkt ja lõpp-punkt ette. Saadud tulemuse võib väljastada konsooli ning see vormistatakse kujule:

Longest path from <alguspunkt> to <lõpp-punkt> is with length <pikkus> and follows arcs:
[<kaare nimi>(<kaal>),<kaare nimi>(<kaal>),<kaare nimi>(<kaal>)]

Näiteks:
Longest path from a to g is with length 9 and follows arcs:
[ad(3), de(3), eg(3)]

Alternatiivselt võib meetodi väljundina salvestada *LongestPath* objekti, millelt saab küsida üksikult vastavaid infot (nt ainult teekonna pikkust).

Näitegraafide genereerimiseks on mitmeid meetodeid:

1) Ahelate loomiseks:
   g.createChainDag(<tippude arv>)
2) Antud näidetes käsitletud graaf:
   g.createDemoDag()
3) Lihtne näide võrdsete teepikkustega graafist:
   g.createSimpleEqualDistancesDag()
4) Massiivse juhusliku graafi loomiseks:
   g.createRandomDag(<tippude arv>, <kaarte arv>);

4 Testimiskava

4.1 Lihtne ahel.

Koostame lihtsa graafi 5 tipuga, mis on ühes ahelas.

Joonis 15: Lihtne ahel.

Antud juhul saame väljundiks:

Longest path from v1 to v5 is with length 18 and follows arcs:
[av1_v2(7), av2_v3(5), av3_v4(2), av4_v5(4)]

Vahetades alguspunkti ja lõpp-punkti ära, saame väljundiks:

Path does not exist from v5 to v1

4.2 Massiivne ahel.

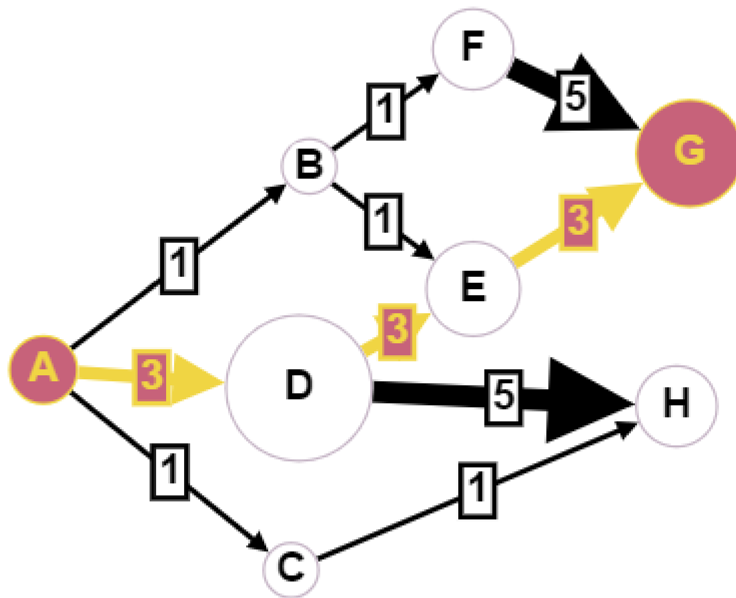
Koostame ahelgraafi 5000 tipuga, kus kaalud on kõikidel kaartel võrdne ühega.

Longest path from v1 to v5000 is with length 4999 and follows arcs:
[av1_v2(1), av2_v3(1), av3_v4(1), av4_v5(1), ... , av4998_v4999(1), av4999_v5000(1)]

Time elapsed for calculation: 161 ms

4.3 Lihtne tsüklita orienteeritud graaf.

Vaatleme eespool käsitletud lihtsamat juhtu: pikim tee tipust A tippu G.



Antud juhul on väljundiks:

Longest path from a to g is with length 9 and follows arcs:
[ad(3), de(3), eg(3)]

Time elapsed for calculation: 56 ms

Järgnevalt küsime sama graafi kohta pikimat teed A ja H vahel.

Longest path from a to h is with length 8 and follows arcs:
[ad(3), dh(5)]

Time elapsed for calculation: 53 ms

4.4 Massiivne juhuslikult genereeritud graaf.

Teeme juhusliku massiivse graafi, milles on 2000 tippu ja 40000 kaart. Antud juhul on eelduseks, et kui kahe suvalise tipu vahel eksisteerib pikim tee, siis vastupidist teed ei tohi eksisteerida, muidu oleks tegu tsüklilise graafiga.
Esimene juhuslike tippude väljund:

Longest path: Longest path from v565 to v1501 is with length 85 and follows arcs: [av565_v1622(10), av1622_v652(6), av652_v419(9), av419_v818(2), av818_v765(10), av765_v1349(1), av1349_v1259(3), av1259_v1112(7), av1112_v570(10), av570_v777(7), av777_v71(4), av71_v481(4), av481_v76(10), av76_v103(5), av103_v1501(3)]

Time elapsed for calculation: 45 ms

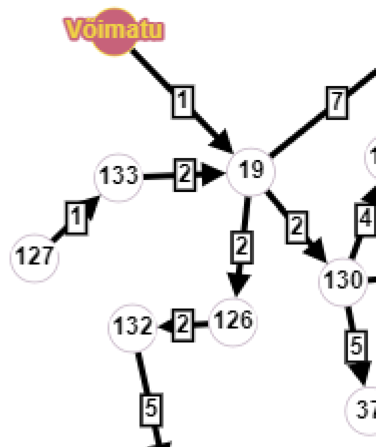
Nüüd vahetame alguse ja lõpu tipud ära:

Longest path: Path does not exist from v1501 to v565

Time elapsed for calculation: 2 ms

4.5 Massiivne juhuslikult genereeritud graaf, milles on võimatu tipp.

Genereerime taas juhusliku graafi 2000 tipu ja 40000 kaarega. Seekord lisame juurde ühe tipu, mis näitab kaarega mõne juhusliku teise graafi tipu poole, st ei ole ühtegi kaart, mis suubuks lisatud tippu.



Joonis 16: Tipp, millesse ei leidu teed.

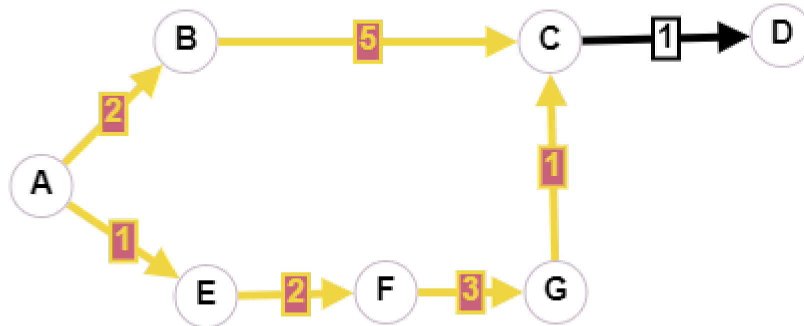
Programmi väljundiks:

Longest path: Path does not exist from v239 to vImpossible

Time elapsed for calculation: 3 ms

4.6 Võrdsete distantside juht.

Vaatleme lihtsat graafi, kus on kaks teed, mis on sama pikad. Sellisel juhul valib programm selle tee, mille puhul esimesena punkti C uuendati pikimat kaugust ning see sõltub sellest, kas esimesena valitakse tipust A väljub kaar kaaluga 2 või kaaluga 1.



Joonis 17: Sama pikk tee lõpp-punkti.

Antud juhul valitakse alati esimene ja see sõltub graafi kokkupanemisel tipu kaarte viidastruktuurist.

Väljund:

Longest path: Longest path from a to d is with length 8 and follows arcs:
[ae(1), ef(2), fg(3), gc(1), cd(1)]

Time elapsed for calculation: 49 ms

5 Kasutatud allikad:

1. Goodrich, M. T. & Tamassia, R. *Data Structures and Algorithms in Java, 4th ed.* (John Wiley & Sons, Inc, 2005).
2. Buldas, A., Laud, P. & Willemson, J. *Graafid*. (Tartu Ülikool, Matemaatika-Informaatikateaduskond, Arvutiteaduse Instituut, 2008).
3. Põial, J. Graaf, eestikeelne sõnavara, https://enos.itcollege.ee/~japoia/algoritmid/graafid.html.
4. Detect cycle in a graph, https://www.geeksforgeeks.org/detect-cycle-in-a-graph/.
5. Directed acyclic path, ideed atsükliliste graafide rakendustest, https://en.wikipedia.org/wiki/Directed_acyclic_graph#Applications.
6. Graph Online, tarkvara graafide kujutamiseks, http://graphonline.ru/en/?graph=OsLdkIIkdENfKUXnZZcst.
7. Topological Sort, William Fiset, https://www.youtube.com/watch?v=eL-KzMXSXXI&t=0s&ab_channel=WilliamFiset.
8. Shortest/Longest path on a Directed Acyclic Graph (DAG) | Graph Theory, William Fiset, https://www.youtube.com/watch?v=TXkDpqjDMHA&list=PLDV1Zeh2NRsDGO4--qE8yH72HFL1Km93P&index=18&ab_channel=WilliamFiset.

6 Lisad

6.1 Programmi täielik tekst.

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

/**
 * 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("G");

        //  test 1: simple determined outcome
    /*    g.createChainDag(5000);
        Vertex start = g.getVertices().get(0);
        Vertex end = g.getVertices().get(g.getVertices().size() - 1);
        System.out.println(g);
        System.out.println("**");
        var startTime = System.currentTimeMillis();
        System.out.println(g.findLongestPathToEnd(start, end));
        var endTime = System.currentTimeMillis();
        System.out.println("Time elapsed for calculation: " + (endTime - startTime) + "
ms");
        System.out.println("**");*/
        // immediate followup again to test if visited is reset


        //  test 1: simple determined outcome
        /*g.createDemoDag();
        Vertex a = g.getVertexById(g.first, "a");
        Vertex e = g.getVertexById(g.first, "h");
        System.out.println(g);

        System.out.println("**");
```

```
        var startTime = System.currentTimeMillis();
        System.out.println(g.findLongestPathToEnd(a, e));
        var endTime = System.currentTimeMillis();
        System.out.println("Time elapsed for calculation: " + (endTime - startTime) + "
ms");
        System.out.println("**");*/
        // immediate followup again to test if visited is reset

        //  test 2: equal distances
        g.createSimpleEqualDistancesDag();
        System.out.println(g);
        Vertex start = g.getVertexById(g.first, "a");
        System.out.println("Random start: " + start);
        // get all accessible vertices from start to pick a legal target
        List<Vertex> topOrderForStart = g.getReverseTopOrderForAVertex(start);
        System.out.println("Reverse topological order: " + topOrderForStart);
        Vertex end = g.getVertexById(g.first, "d");
        System.out.println("End target: " + end);

        System.out.println("**");
        var startTime = System.currentTimeMillis();
        System.out.println("Longest path: " + g.findLongestPathToEnd(start, end));
        var endTime = System.currentTimeMillis();
        System.out.println("Time elapsed for calculation: " + (endTime - startTime) + "
ms");
        System.out.println("**");

        // test 3: (huge) random generator test
        // generation takes time
/*
        g.createRandomDag(2000, 40000);
        System.out.println(g);
        Vertex start = g.vertices.get(g.getRandomInt(g.vertices.size() - 1));
        System.out.println("Random start: " + start);
        // get all accessible vertices from start to pick a legal target
        List<Vertex> topOrderForStart = g.getReverseTopOrderForAVertex(start);
        System.out.println("Reverse topological order: " + topOrderForStart);
        Vertex end = topOrderForStart.get(0);
        System.out.println("End target: " + end);
        System.out.println("**");
        var startTime = System.currentTimeMillis();
        System.out.println("Longest path: " + g.findLongestPathToEnd(start, end));
        var endTime = System.currentTimeMillis();
        System.out.println("Time elapsed for calculation: " + (endTime - startTime) + "
ms");
        System.out.println("**");
        startTime = System.currentTimeMillis();
        System.out.println("Longest path: " + g.findLongestPathToEnd(end, start));
        endTime = System.currentTimeMillis();
        System.out.println("Time elapsed for calculation: " + (endTime - startTime) + "
ms");
```

```
        System.out.println("**");
*/


        // test 4: (huge) random generator test with impossible connection
        // generation takes time
/*        g.createRandomDag(2000, 40000);
        System.out.println(g);
        // add inaccessible vertex
        System.out.println("Adding new inaccessible vertex to random random vertex:");
        System.out.println("Vertices before: " + g.vertices.size());
        Vertex end = g.createVertex("vImpossible");
        System.out.println("Created vertex: " + end);
        System.out.println("Vertices after: " + g.vertices.size());
        Vertex randomVertex = g.vertices.get(g.getRandomInt(g.vertices.size() - 1));
        Arc arcNaToRandom = g.createArc("NA out", end, randomVertex);
        System.out.println("Created arc from new vertex (" + end + ") to random vertex
(" + randomVertex + "): " + arcNaToRandom);
        // find distance
        System.out.println("\nFind out distance between vertices:");
        Vertex start = g.vertices.get(g.getRandomInt(g.vertices.size() - 1));
        System.out.println("Random start: " + start);
        List<Vertex> topOrderForStart = g.getReverseTopOrderForAVertex(start);
        System.out.println("Reverse topological order: " + topOrderForStart);
        System.out.println("End target: " + end);
        System.out.println("**");
        var startTime = System.currentTimeMillis();
        System.out.println("Longest path: " + g.findLongestPathToEnd(start, end));
        var endTime = System.currentTimeMillis();
        System.out.println("Time elapsed for calculation: " + (endTime - startTime) + "
ms");
        System.out.println("**");*/

        // test 5: chain test
/*        g.createChainDag(5000);
        System.out.println(g);
        Vertex start = g.vertices.get(g.vertices.size() - 1);
        System.out.println("Reverse topological order: \n" +
g.getReverseTopOrderForAVertex(start));
        Vertex end = g.vertices.get(0);
        System.out.println("Longest path: " + g.findLongestPathToEnd(start, end));*/
    }


    /**
     * Vertex represent one intersection or endpoint in graph.
     * Vertex has helper attributes "topological order index" and "visited" status for
use in Graph functions.
     */
    class Vertex {
```

```java
        private final String id;
        private Vertex next;
        private Arc first;
        private int topOrderIndex = -1;
        private boolean visited = false;

        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.
     * Arcs can have weights to represent distances between two vertices (default weight
is 1).
     */
    class Arc {

        private final String id;
        private Vertex target;
        private Arc next;
        private final int weight;

        Arc(String s, Vertex v, Arc a) {
            id = s;
            target = v;
            next = a;
            this.weight = 1;
        }

        Arc(String s, int weight) {
            id = s;
            target = null;
            next = null;
            this.weight = weight;
        }

        Arc(String s) {
```

```java
        this(s, null, null);
    }

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

}


/**
 * Container for the results of the longest path search.
 * Contains longest path length between start and end vertices, with also
 * list of arcs in topological order that form the path.
 */
class LongestPath {


    private final Vertex start;
    private final Vertex end;
    private final int pathLength;
    private final List<Arc> pathArcs;

    LongestPath(Vertex start, Vertex end, int pathLength, List<Arc> pathVertices) {
        this.start = start;
        this.end = end;
        this.pathLength = pathLength;
        this.pathArcs = pathVertices;
    }

    LongestPath(Vertex start, Vertex end, int pathLength) {
        this(start, end, pathLength, null);
    }

    public Vertex getStart() {
        return start;
    }

    public Vertex getEnd() {
        return end;
    }

    public int getPathLength() {
        return pathLength;
    }

    public List<Arc> getPathArcs() {
        return pathArcs;
    }
```

```
        @Override
        public String toString() {
            if (start == null && end == null) return "Path does not exist";
            if (pathLength == -1) return "Path does not exist from " + start + " to " +
end;
            return "Longest path from " + start + " to " + end +
                    " is with length " + pathLength + " and follows arcs:\n" + pathArcs;
        }
    }

    /**
     * The problem to solve: Design an efficient algorithm for finding a longest
directed path from a vertex s
     * to a vertex t of an acyclic weighted digraph (DAG). Providing both the length of
the path and the list of
     * vertices the path consists of.
     * <p>
     * Wrote a DAG generation for huge graphs which took 3x more time than the path
algorithm itself.
     * <p>
     * Sources for solutions ideas, none offered complete solutions:
     * general theory: TalTech Algorithms lectures
     * and Coursera lectures at https://www.coursera.org/learn/algorithms-
part2/home/welcome
     * Topological sort example for DAG:
     * https://www.youtube.com/watch?v=eL-KzMXSXXI&list=PLDV1Zeh2NRsDGO4--
qE8yH72HFL1Km93P&index=15&ab_channel=WilliamFiset
     * Finding shortest and longest paths example for DAG:
     * https://www.youtube.com/watch?v=TXkDpqjDMHA&list=PLDV1Zeh2NRsDGO4--
qE8yH72HFL1Km93P&index=17&ab_channel=WilliamFiset
     * cycle check for DAG:
     * https://www.geeksforgeeks.org/detect-cycle-in-a-graph/
     */
    class Graph {

        private final String id;
        private Vertex first;
        private List<Vertex> vertices;
        private final int MAXIMUM_WEIGHT = 10;
        private final int MINIMUM_WEIGHT = 1;

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

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

```java
@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();
}

/**
 * Adds a new vertex to vertices chain, not indicating connection between them.
 * Also updates the vertices list of the graph.
 **/
public Vertex createVertex(String vid) {
    Vertex vertex = new Vertex(vid);
    vertex.next = first;
    first = vertex;
    vertices = getVertices();
    return vertex;
}

/**
 * Adds a new connection between vertices with weight 1.
 * Also updates the vertices list of the graph.
 **/
public Arc createArc(String aid, Vertex from, Vertex to) {
    Arc arc = new Arc(aid, 1);
    arc.next = from.first;
    from.first = arc;
    arc.target = to;
    vertices = getVertices();
    return arc;
}
```

```java
    /**
     * Adds a new connection between vertices with specified weight.
     * Also updates the vertices list of the graph.
     **/
    public Arc createArc(String aid, Vertex from, Vertex to, int weight) {
        Arc arc = new Arc(aid, weight);
        arc.next = from.first;
        from.first = arc;
        arc.target = to;
        vertices = getVertices();
        return arc;
    }

    /**
     * Create a random directed acyclic graph (DAG).
     * Without multiple arcs, with provided number of vertices and arcs.
     *
     * @param nrOfVertices number of vertices
     * @param nrOfArcs     number of arcs
     */
    public void createRandomDag(int nrOfVertices, int nrOfArcs) {
        if (nrOfVertices <= 0)
            return;
        if (nrOfVertices > 2500)
            throw new IllegalArgumentException("Too many vertices: " +
nrOfVertices);
        if (nrOfArcs < nrOfVertices - 1 || nrOfArcs > nrOfVertices * (nrOfVertices -
1) / 2)
            throw new IllegalArgumentException
                    ("Impossible number of edges: " + nrOfArcs);
        first = null;
        createRandomTreeForDag(nrOfVertices);       // n-1 arcs created here
        vertices = getVertices();
        int arcsLeftToDo = nrOfArcs - nrOfVertices + 1;  // remaining arcs
        while (arcsLeftToDo > 0) {
            int i = getRandomInt(nrOfVertices);
            int j = getRandomInt(nrOfVertices);
            if (i == j) {
                continue;  // no loops
            }
            Vertex fromVertex = vertices.get(i);
            Vertex toVertex = vertices.get(j);
            boolean arcDoesNotExists = arcDoesNotExist(toVertex, fromVertex)
                    && arcDoesNotExist(fromVertex, toVertex);
            boolean noCycles = !graphHasCycle(toVertex, fromVertex);
            if (arcDoesNotExists && noCycles) {
                createArc("a" + fromVertex.toString() + "_" + toVertex.toString(),
                        fromVertex, toVertex, getRandomInt(MAXIMUM_WEIGHT) +
MINIMUM_WEIGHT);
                arcsLeftToDo--;
```

```java
            }
        }
    }

    /**
     * Generate a random integer value form 0 to provided upperbound.
     *
     * @param upperBound upper bound of the generated value
     * @return random integer between 0-upper bound
     */
    private int getRandomInt(int upperBound) {
        return (int) (Math.random() * upperBound);
    }

    /**
     * Find out if arc pointing to vertex already exists.
     *
     * @param to   vertex to look for
     * @param from vertex whose arcs to check for
     * @return true, if arc does not exist, otherwise false
     */
    private boolean arcDoesNotExist(Vertex to, Vertex from) {
        Arc nextArc = from.first;
        while (nextArc != null) {
            if (nextArc.target == to) {
                return false;
            }
            nextArc = nextArc.next;
        }
        return true;
    }

    /**
     * Create graph with provided number of vertices and one arc.
     * Arc will be pointing to a random vertex that was created before it, thus no
cycles.
     *
     * @param nrOfVertices number of vertices to create in the graph
     */
    private void createRandomTreeForDag(int nrOfVertices) {
        if (nrOfVertices <= 0)
            throw new RuntimeException("Can´t create graph with 0 vertices.");
        Vertex[] vertices = new Vertex[nrOfVertices];
        for (int i = 0; i < nrOfVertices; i++) {
            vertices[i] = createVertex("v" + (nrOfVertices - i));
            if (i == 0) {
                continue;
            }
            int sourceIndex = (int) (Math.random() * i - 1);
            createArc("a" + vertices[sourceIndex].toString() + "_"
                            + vertices[i].toString(), vertices[sourceIndex],
```

```
vertices[i],
                        getRandomInt(MAXIMUM_WEIGHT) + MINIMUM_WEIGHT);
            }
        }


        /**
         * Return a list of all vertices in the graph.
         *
         * @return list of vertices.
         */
        private List<Vertex> getVertices() {
            List<Vertex> vertices = new LinkedList<>();
            Vertex start = this.first;
            while (start != null) {
                vertices.add(start);
                start = start.next;
            }
            return vertices;
        }

        /**
         * Creates a simple DAG where all vertices are connected by one arc only.
         *
         * @param nrOfVertices number of vertices in the graph
         */
        public void createChainDag(int nrOfVertices) {
            if (nrOfVertices <= 0)
                throw new RuntimeException("Can´t create graph with less than 1
vertices.");
            if (nrOfVertices > 5000)
                throw new IllegalArgumentException("Too many vertices: " +
nrOfVertices);
            first = null;
            Vertex[] vertices = new Vertex[nrOfVertices];
            for (int i = 0; i < nrOfVertices; i++) {
                vertices[i] = createVertex("v" + (nrOfVertices - i));
                if (i == 0) {
                    continue;
                }
                int sourceIndex = i - 1;
                createArc("a" + vertices[i].toString() + "_"
                                + vertices[sourceIndex].toString(), vertices[i],
vertices[sourceIndex],
                        getRandomInt(MAXIMUM_WEIGHT) + MINIMUM_WEIGHT);
            }
            this.vertices = getVertices();
        }

        /**
         * Recursively finds vertex by id.
```

```java
     * Returns the first match.
     *
     * @param vertex vertex where to start looking from, usually first vertex of
graph
     * @param id     id of the vertex to look for
     * @return vertex, if found, otherwise null
     */
    public Vertex getVertexById(Vertex vertex, String id) {
        if (vertex.id.equals(id)) return vertex;
        if (vertex.next == null) return null;
        return getVertexById(vertex.next, id);
    }


    /**
     * Creates a simple DAG where longest path from "a" to "e" is 19.
     */
    public void createSimpleDag() {
        Vertex a = createVertex("a");
        Vertex b = createVertex("b");
        Vertex c = createVertex("c");
        Vertex d = createVertex("d");
        Vertex e = createVertex("e");
        Vertex f = createVertex("f");
        Vertex g = createVertex("g");
        Vertex h = createVertex("h");
        vertices = getVertices();
        createArc("ab", a, b);
        createArc("ac", a, c, 10);
        createArc("bd", b, d, 12);
        createArc("ce", c, e, 3);
        createArc("de", d, e, 2);
        createArc("ef", e, f);
        createArc("cg", c, g);
        createArc("gh", g, h);
        createArc("dc", d, c, 3);
    }

    /**
     * Creates a simple DAG where longest path from "a" to "e" is 19.
     */
    public void createDemoDag() {
        Vertex a = createVertex("a");
        Vertex b = createVertex("b");
        Vertex c = createVertex("c");
        Vertex d = createVertex("d");
        Vertex e = createVertex("e");
        Vertex f = createVertex("f");
        Vertex g = createVertex("g");
        Vertex h = createVertex("h");
        vertices = getVertices();
```

```java
        createArc("ab", a, b, 1);
        createArc("ad", a, d, 3);
        createArc("ac", a, c, 1);
        createArc("bf", b, f, 1);
        createArc("be", b, e, 1);
        createArc("fg", f, g, 5);
        createArc("eg", e, g, 3);
        createArc("dh", d, h, 5);
        createArc("de", d, e, 3);
        createArc("ch", c, h, 1);
    }


    /**
     * Creates a simple DAG where longest path from "a" to "e" is 19.
     */
    public void createSimpleEqualDistancesDag() {
        Vertex a = createVertex("a");
        Vertex b = createVertex("b");
        Vertex c = createVertex("c");
        Vertex d = createVertex("d");
        Vertex e = createVertex("e");
        Vertex f = createVertex("f");
        Vertex g = createVertex("g");
        vertices = getVertices();
        createArc("ae", a, e, 1);
        createArc("ab", a, b, 2);
        createArc("bc", b, c, 5);
        createArc("cd", c, d, 1);
        createArc("ef", e, f, 2);
        createArc("fg", f, g, 3);
        createArc("gc", g, c, 1);
    }

    /**
     * Helper function to reset vertices visited value.
     * Needed for topological sorting.
     */
    private void setVerticesNotVisited() {
        for (Vertex vertex : vertices) {
            vertex.visited = false;
        }
    }

    /**
     * Helper function to reset vertices top order index value.
     * Needed for finding longest path between vertices.
     */
    private void resetVerticesTopOrderIndex() {
        for (Vertex vertex : vertices) {
            vertex.topOrderIndex = -1;
```

```java
    }
}


/**
 * Helper function to reset vertices visited and top order index values.
 * Needed for topological sorting and finding longest path between vertices.
 */
private void resetVertices() {
    for (Vertex vertex : vertices) {
        vertex.topOrderIndex = -1;
        vertex.visited = false;
    }
}


/**
 * Finds if there is a path from one vertex another vertex.
 * <p>
 * Side effect: resets vertices visited values
 *
 * @param to   vertex to start looking from
 * @param from vertex what to look for for a cycle to exists
 * @return true, if it is possible to get to start vertex from vertex
 */
private boolean graphHasCycle(Vertex to, Vertex from) {
    setVerticesNotVisited();
    boolean hasCycle = hasCycle(to, from);
    setVerticesNotVisited();
    return hasCycle;
}

/**
 * Recursively find out if there is a path from one vertex to another vertex.
 * <p>
 * Side effect: resets vertices visited values
 *
 * @param to   vertex whose children to compare against the other vertex
 * @param from vertex we are looking for
 * @return true, if it is possible to get to the other vertex
 */
private boolean hasCycle(Vertex to, Vertex from) {
    to.visited = true;
    if (to == from) {
        return true;
    }
    if (to.first == null) return false;
    Arc arc = to.first;
    while (arc != null) {
        Vertex nextTo = arc.target;
        if (!nextTo.visited) {
```

```
                if (hasCycle(nextTo, from)) {
                    return true;
                }
            }
            arc = arc.next;
        }
        return false;
    }


    /**
     * Find the longest directed path from start vertex to end vertex.
     * Weights of the arcs are taken as the distance between any two vertices.
     * <p>
     * Side effect: resets vertex visited and topOrderIndex values.
     *
     * @param start vertex to start looking from
     * @param end   vertex to look for
     * @return object with longest path start, end, length and arcs info
     */
    public LongestPath findLongestPathToEnd(Vertex start, Vertex end) {
        if (start == null || end == null) {
            throw new RuntimeException("Start vertex or end vertex was empty.");
        }
        resetVertices();
        List<Vertex> topOrder = new LinkedList<>();
        getReverseTopOrderToTarget(start, end, topOrder);
        if (topOrder.size() < 2) return new LongestPath(start, end, -1);
        LongestPath longestPath = findLongestPath(topOrder, end);
        resetVertices();
        return longestPath;
    }


    /**
     * Find the reverse topological order to the target vertex.
     * Initial provided vertex is the start vertex.
     * Recursively look for all accessible paths from start vertex to target.
     * If meeting target, search will not look behind it.
     * All accessible paths not ending with target are also included.
     * <p>
     * Side effect: using vertices topOrderIndex to provide later distance
measurement
     *
     * @param vertex    Initial start vertex to start looking from.
     * @param target    Target vertex to look for.
     * @param topOrder Reverse topological order container where the found vertices
are added recursively.
     */
    private void getReverseTopOrderToTarget(Vertex vertex, Vertex target,
List<Vertex> topOrder) {
```

```
            vertex.visited = true;
            if (vertex != target) {
                boolean firstVertexNotVisited = vertex.first != null &&
vertex.first.target != null
                        && !vertex.first.target.visited;
                if (firstVertexNotVisited) {
                    getReverseTopOrderToTarget(vertex.first.target, target, topOrder);
                    boolean neighbourVertexNotVisited = vertex.first.next != null &&
vertex.first.next.target != null
                            && !vertex.first.next.target.visited;
                    if (neighbourVertexNotVisited) {
                        getReverseTopOrderToTarget(vertex.first.next.target, target,
topOrder);
                    }
                }
            }
            vertex.topOrderIndex = topOrder.size();
            topOrder.add(vertex);
        }


        /**
         * Generate reverse topological order for the provided vertex.
         * The provided vertex is the last one.
         * <p>
         * Side effect: resets vertices visited status.
         *
         * @param vertex vertex to get the reverse topological order for.
         * @return list of vertices in reverse topological order
         */
        public List<Vertex> getReverseTopOrderForAVertex(Vertex vertex) {
            setVerticesNotVisited();
            resetVerticesTopOrderIndex();
            List<Vertex> topOrderForStart = new LinkedList<>();
            getReverseTopOrderForAVertex(vertex, topOrderForStart);
            setVerticesNotVisited();
            return topOrderForStart;
        }


        /**
         * Find the reverse topological order for the provided vertex.
         * Initial provided vertex is the start vertex.
         * Recursively look for all accessible paths from start vertex.
         * <p>
         * Side effect: using vertices visited value
         *
         * @param vertex   vertex from where to look for all accessible vertices.
         * @param topOrder reverse topological order container that is filled
recursively
         */
```

```
        private void getReverseTopOrderForAVertex(Vertex vertex, List<Vertex> topOrder)
{
            vertex.visited = true;
            boolean firstVertexNotVisited = vertex.first != null && vertex.first.target
!= null
                    && !vertex.first.target.visited;
            if (firstVertexNotVisited) {
                getReverseTopOrderForAVertex(vertex.first.target, topOrder);
                boolean neighbourVertexNotVisited = vertex.first.next != null &&
vertex.first.next.target != null
                        && !vertex.first.next.target.visited;
                if (neighbourVertexNotVisited) {
                    getReverseTopOrderForAVertex(vertex.first.next.target, topOrder);
                }
            }
            vertex.topOrderIndex = topOrder.size();
            topOrder.add(vertex);
        }

        /**
         * Find the longest path from start vertex to end vertex from the provided
reverse topological order.
         * Start vertex is the last vertex in the provided reverse topological order.
         * Arc weights are used as distances between any two vertices.
         * In case of equal distances, differences are not made and the first to match
is taken.
         *
         * @param topOrder reverse topological order with start vertex
         * @param end      target vertex to which the longest path is looked for
         * @return object with start and end vertices, longest path length and list of
arcs that form the path.
         */
        private LongestPath findLongestPath(List<Vertex> topOrder, Vertex end) {
            Integer[] distancesFromStart = new Integer[topOrder.size()];
            Vertex[] priorVertices = new Vertex[topOrder.size()];
            for (int i = topOrder.size() - 1; i > 0; i--) {
                Vertex source = topOrder.get(i);
                if (source == end) break;
                if (distancesFromStart[source.topOrderIndex] == null)
distancesFromStart[source.topOrderIndex] = 0;
                updateDistancesFromSource(distancesFromStart, priorVertices, source);
            }
            Vertex start = topOrder.get(0);
            if (end.topOrderIndex == -1) return new LongestPath(start, end, -1);
            List<Arc> longestPath = getLongestPathArcs(start, end, priorVertices);
            int longestPathLength = distancesFromStart[end.topOrderIndex];
            return new LongestPath(topOrder.get(topOrder.size() - 1), end,
longestPathLength, longestPath);
        }

        /**
```

```
         * Update the distances from source vertex to all other vertices its arcs are
pointing to.
         *
         * @param distancesFromStart Container for updated distances.
         * @param priorVertices      Container for reconstructing the longest path.
         * @param source             Vertex whose children distances from start will be
updated.
         */
        private void updateDistancesFromSource(Integer[] distancesFromStart, Vertex[]
priorVertices, Vertex source) {
            Arc nextArc = source.first;
            Vertex nextTarget;
            int arcWeight;
            while (nextArc != null) {
                nextTarget = nextArc.target;
                arcWeight = nextArc.weight;
                if (nextTarget.topOrderIndex == -1) {
                    nextArc = nextArc.next;
                    continue;
                }
                if (distancesFromStart[nextTarget.topOrderIndex] == null) {
                    distancesFromStart[nextTarget.topOrderIndex] = 0;
                }
                int newDistance = distancesFromStart[source.topOrderIndex] + arcWeight;
                if (distancesFromStart[nextTarget.topOrderIndex] < newDistance) {
                    distancesFromStart[nextTarget.topOrderIndex] = newDistance;
                    priorVertices[nextTarget.topOrderIndex] = source;
                }
                nextArc = nextArc.next;
            }
        }

        /**
         * Get arcs list for the longest path from start vertex to end vertex.
         * Relies on the priorVertices array, where at the vertex topSortIndex position
its parent vertex is positioned.
         *
         * @param start         first vertex of the path
         * @param end           last vertex of the path
         * @param priorVertices Vertex array with prior vertices at topSortIndex
position.
         * @return List of arcs in topological order that form the longest path from
start to end vertex.
         */
        private List<Arc> getLongestPathArcs(Vertex start, Vertex end, Vertex[]
priorVertices) {
            List<Arc> longestPath = new LinkedList<>();
            Vertex target = end;
            Vertex source = priorVertices[target.topOrderIndex];
            ;
            while (source != null) {
```

```java
                Arc longestArc = getLongestArc(source, target);
                longestPath.add(longestArc);
                target = source;
                source = priorVertices[source.topOrderIndex];
                if (source == start) {
                    break;
                }
            }
            Collections.reverse(longestPath);
            return longestPath;
        }

        /**
         * Find the longest arc from source to target vertex.
         *
         * @param target vertex the arc has to point to
         * @param source vertex from where the arc points from
         * @return the Arc with the longest distance (weight)
         */
        private Arc getLongestArc(Vertex source, Vertex target) {
            Arc nextArc = source.first;
            int longestDistance = 0;
            Arc longestArc = null;
            while (nextArc != null) {
                if (nextArc.target == target && nextArc.weight > longestDistance) {
                    longestDistance = nextArc.weight;
                    longestArc = nextArc;
                }
                nextArc = nextArc.next;
            }
            return longestArc;
        }
    }
}
```