TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jan-Erik Kalmus, IADB186008

KODUTÖÖ 6

Aines Algoritmid ja andmestruktuurid (ICD0001)

Juhendaja: Jaanus Pöial

Tallinn 2020

Autorideklaratsioon

Kinnitan, et olen koostanud antud kodutöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Jan-Erik Kalmus

20.11.2020

Sisukord

1 Ülesandepüstitus

Autori poolt valitud kodutöö nr. 6 ülesandepüstitus on:

„*On antud sidus lihtgraaf, mille iga serva jaoks on teada selle läbilaskevõime (toru jämedus, mittenegatiivne). Kootada meetod, mis leiab kahe etteantud tipu vahelise maksimaalse läbilaskevõime Ford-Fulkersoni meetodil.*“

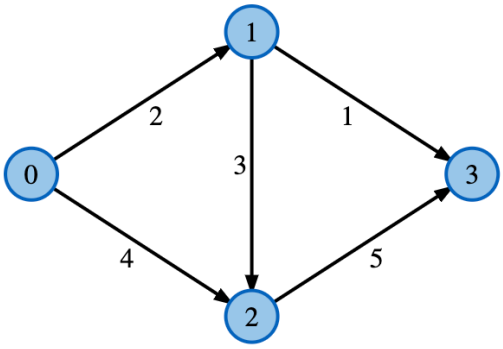
2 Graafiteooria sissejuhatus

Käesoleva ülesande lahenduse mõistmiseks peaks olema vähemalt lühiülevaade graafiteooriast. Järgnevalt tutvustab autor graafiteooria põhialuseid ülesande mõistmiseks mineva vajalikkuse piires.

2.1 Graaf

Lihtgraaf G on kahe hulga kombinatsioon: tema tippude hulk *V* ja tema servade hulk *E*. (Buldas, Laud, & Willemson, 2008) Serva võib vaadelda kui teed või ühendust kahe serve vahel, mistõttu on ka ühe serva esituseks hulgas E kahe tipu kombinatsioon, näiteks *e1 = {v1, v2}*. (Buldas, Laud, & Willemson, 2008)

Hulgad V ja E saavad oma tähised vastavalt terminite inglisekeelsetele tähistustele *Vertex* ja *Edge* (mitmuses *Vertices* ja *Edges*).



Joonis 1. Näide siduvast lihtgraafist, mille servadel on tähistatud ka läbilaskevõimed.

Servi võime omakorda määratleda kahte liiki: suunamata ja suunatud servad. Suunatud serva iseloomustab serva kindel suund, ehk kahe tipu vahel on võimalik liikuda vaid üht pidi, nagu näiteks eelnevalt toodud serva *e1 = {v1, v2}* puhul.

Suunamata serva puhul on liikumine võimaik aga mõlemas suunas. Seetõttu on suunamata serva esitus teistsugune. *e1 = {(v1, v2), (v2, v1)}*. Suunamata serva võime nimetada ka kaareks. (Buldas, Laud, & Willemson, 2008)

Ülesandepüstituse täielikuks mõistmiseks on vaja ka mõista terminit *sidus graaf*. Graaf G on sidus, kui iga tema kahe erineva punkti vahel leidub vähemalt üks neid tippe omavahel ühendav ahel (nimetatakse ka teeks).

3 Ford-Fulkersoni algoritm

Graafiteooria mõistes üldistatult võimaldab Ford-Fulkersoni algoritm arvutada kahe punkti vahelist maksimaalselt läbilaskevõimet, võttes arvesse kõikide servade individuaalselt läbilaskevõime piiri.

Autorite poolt on sõnastatud algoritmi aluseks olev teoreem:

Võrgu maksimaalsete voogude väärtused on võrdsed selle võrgu minimaalsete lõigete läbilaskevõimetega. (Fulkerson & Ford, 1955) (Buldas, Laud, & Willemson, 2008)

3.1 Terminoloogia

- Voog (*flow*) – kahe punkti vahel oleval serval liikuva info või massi arvuline väärtus. (Buldas, Laud, & Willemson, 2008)
- Lähe *(source*) – tipp, millest voo uurimist alustatakse (Buldas, Laud, & Willemson, 2008)
- Suue (*sink*) – tipp, mis on uuritava voo lõpp-punktiks (Buldas, Laud, & Willemson, 2008)
- Sild (*augmented path)* – ahel, mille puhul ei esine tsükleid ja kõikide ahela lülide läbilaskevõimed on algoritmile vastavad (Buldas, Laud, & Willemson, 2008)
- Läbilaskevõime (*capacity*) – maksimaalne voog, mis servas võib esineda (Buldas, Laud, & Willemson, 2008)

3.2 Piirangud

Algoritmi rakendamiseks on seatud järgnevad piirangud:

- Servas liikuv vool ei tohi ületada selle maksimaalset läbilaskevõimest (Fischer, 2015-2017)

- Alg ja lõpp punktide vahel olevate punktide siseneva ja väljuva voogude vaheline suhe peab olema 0 (Fischer, 2015-2017)

- A -> B: punktist A väljuv voog peab olema võrdne punkti B sisendiks oleva vooga (Fischer, 2015-2017)

3.3 Algoritm

Algoritmi üldine tööpõhimõte lahti seletatuna:

- Määrata algseks maksimaalseks läbilaskevõimeks 0

- Arvutada maksimaalne läbilaskevõime iga võimaliku sobiva alguspunkti ja lõpp-punkti ühendava piirangutele vastava tee puhul

- Lisada tulemus summale

- Eelmise kahe sammu kohta tsükkel kuni enam ühtegi sobivat teed ei leita

- Tagastada summa

3.4 Rakendused

Ford-Fulkersoni algoritm loodi L.R. Ford ja D.R. Fulkersoni poolt 1955. aastal. (Buldas, Laud, & Willemson, 2008) Algoritm leidis esimest korda suuremat rakendust vahetult peale selle avaldamist, kui Ühendriikide sõjaväeringkondadel tekkis vajadus arvutada Ida-Euroopa raudteevõrgustiku maksimaalset võimekust läbilaskevõimet saatmaks Nõukogude Liidu varustust rindejoonele võimaliku konflikti korral. (Buldas, Laud, & Willemson, 2008)

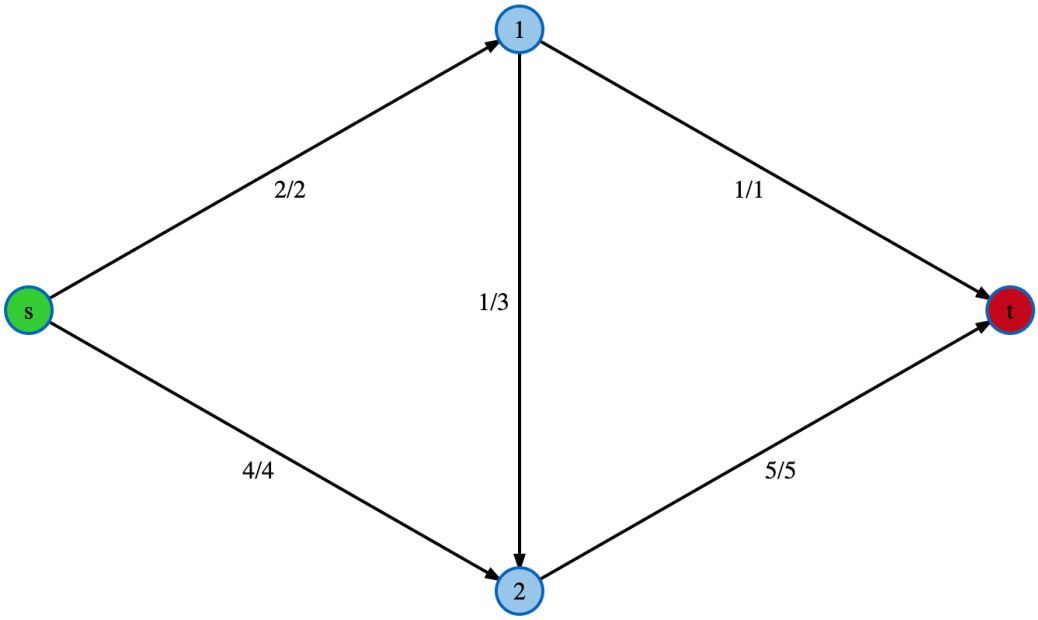
3.5 Lihtsutatud näide

Järgneva graafi G tippude hulk on V = {v1, v2, v3, v4} ning servade hulk E = {(v3, v2), (v3, v1), (v1, v2), (v1, v4), (v2, v4)}, kus igale servale on määratud kindel läbilaskevõime. Ford-Fulkersoni algoritmi rakendades leiame kolm võimaliku teed:

- V3 -> V1 -> V4, kus (v3, v1) läbilaskevõime suhteks jääb 1/2 ja (v1, v4) suhteks 1/1. Kokku on selle tee maksimaalne läbilaskevõime **1**.

- V3 -> V1 -> V2 -> V4, kus (v3, v1) läbilaskevõime suhteks jääb 2/2, (v1, v2) suhteks 1/3 ja (v2, v4) puhul on suhteks 1/5. Kokku on selle tee maksimaalne läbilaskevõime **1**.

- V3 -> V2 -> V4, kus (v3, v2) läbilaskevõime suhteks jääb 4/4 ja (v2, v4) suhteks 5/5. Kokku on selle tee maksimaalne läbilaskevõime **4**.



Joonis 2. Näitena toodud sidus lihtgraaf, kus *s* on *v3* ja *t* on *v4*. Iga serva juures on näha kasutatud läbilaskevõime hulka. Näiteks (v1, v2) puhul 1/3, kus 1 on hetke voog servas ning 3 on serva maksimaalne läbilaske võime.

Oluline on jällegi silmas pidada, et iga tee maksimaalse läbilaskevõime määrab selle tee kõike väiksema läbilaskevõimega lüli. Seetõttu kutsutaksegi kõige väiksema läbilaskevõimega tee lüli pudelikaelaks.

Liites iga tee maksimaalse läbilaskevõime kokku, saame tulemuseks **6** (1+1+4).

4 Kasutatud andmestruktuurid

Lahenduses kasutati ülesandeks ettenähtud andmestruktuuride põhja, mis koosnes põhiliselt kolmest komponendist: graafi andmestruktuur (*Graph*), servade andmestruktuur (*Arc)* ja tippude andmestruktuur *(Vertex)*. Lahendusesiseselt loodi lisaks vaid leitud teega seotud andmete hoidmiseks eraldi andmestruktuur nimega Path.

Järgnevalt toob autor välja iga kasutatud andmestruktuuri põhiparameetrid ning ülesande põhjale lisatud omapoolsed muudatused.

4.1 Graph

Ülesande põhjas olnud andmestruktuur. Andmestruktuur on seotud graafiga.

Andmestruktuuri väljad:

- Id - Graafi nimetus. Andmetüüp: sõne *(String)*
- First – Viide graafi esimesele tipule. Andmetüüp: *Vertex*.

Antud andmestruktuuri puhul autor muudatusi ei teinud.

4.2 Vertex

Ülesande põhjas olnud andmestruktuur. Andmestruktuur esindab ühte tippu graafil.

Andmestruktuuri väljad:

- Id - Tipu nimetus. Andmetüüp: sõne *(String)*
- Next – Viide järgmisele tipule. Andmetüüp: *Vertex.*
- First – Viide tipu esimesele servale. Andmetüüp*: Arc*.

Antud andmestruktuuri puhul autor muudatusi ei teinud.

4.3 Arc

Ülesande põhjas olnud andmestruktuur. Andmestruktuur esindab graafil oleva omavahel ühendatud kahe tipu vahelist serva.

Andmestruktuuri väljad:

- *Id* - Kaare nimetus. Andmetüüp: sõne *(String)*
- *Next* – Viide serva lähtepunktis oleva tipu järgmisele kaarele. Andmetüüp: *Arc*.
- *Target* – Viide serva otspunktis oleva tipule. Andmetüüp: *Vertex*.
- *Capacity* – Kaare maksimaalne läbilaskevõime. Andmetüüp: täisarv *(Integer)*
- *Flow* – Voog servas. Andmetüüp: täisarv *(Integer)*

Andmestruktuurile lisas autor väljad *capacity* ja *flow* hoidmaks informatsiooni serva maksimaalse läbilaskevõime ja voo kohta.

4.4 Path

Autori poolt loodud andmestruktuur. Andmestruktuur on esitus ühest loodud sobivast teest graafil oleva kahe tipu vahel.

Andmestruktuuri väljad:

- *arcs* - Nimekiri, mis hoiab viiteid kõikidele tee liikmeteks olevatele servadele. Andmetüüp: List<*Arc*>
- *maximumCapacity* – Hoiab tee maksimaalset läbilaskevõimet.

5 Lahendus

Järgnevas osas väljendab autor Ford-Fulkersoni algoritmi lihtsasti hoomatavas pseudokoodis, millele tuginedes luuakse ka algoritmi realisatsioon ise.

```
int ford_fulkerson(Graph graph, Vertex start, Vertex end) {
	// algne maksimaalne läbilaske võime on 0
	var max = 0

	// peame nimekirja läbitud teedest, et ei korduks
	var nimekiri

	// tsükkel kestab niikaua, kuni leiame sobivaid teid mida läbida
	while (path_fn(start, end, nimekiri))

		// võta viimane lisatud väärtus nimekirjast (path_fn tulemus)
		var path

		// leia bottleneck (määrab valitud tee maksimaalse
		läbilaskevõime)
		var bottleneck = path.getMaxCapacity()

		// lisa bottleneck maksimaalsele läbilaskevõimele
		max += bottleneck

	// tagastame maksimaalse läbilaske võime
	return max;
}
```

Algoritmi töö toetamiseks on vaja luua abimeetod, mille ülesandeks on graafis kahe antud punkti vahel sobiva tee leidmine. Tee leidmiseks rakendatakse selles lahenduses *breadth-first search* algoritmi, mille käigus otsitakse lühim tee.

Bteadth-first search kujutab endast algoritmi, millega otsitakse läbi graafi iga tipu kõik naabrid ja selle tulemusel leitakse ülesse sihiks olnud tipp. Algoritmi töö tulemuse alusel saame võimalusel rekonstrueerida tee tipust tipuni.

Loodud tee hoidmiseks ja sellega seotud arvutused pannakse eraldi klassi sisse. Selle andmestruktuuri sisse kapseldatakse ka erinev töötlusega seotud loogika, näiteks erinevate voogude muutmine, millega parandatakse programmikoodi loetavust.

Kogu algoritmiga seonduv loogika seotakse ühes ülemklassis, mille sisse antakse vajalikud andmestruktuurid parameetritena. Eesmärgiks on kasutajamugavus, mille puhul peab kasutaja vaid klassile andma sisse vajalikud andmed parameetritena ning meetodit *Execute* käivitama.

Samuti paraneb ülemklassi loomise tulemusena koodi disain ja loetavus, sest erinevad algoritmiga seotud meetodid, mis jagavad sarnaseid alginfo parameetreid, ei pea neid üksteisele pidevalt sisendparameetritena saatma.

5.1 Käivitamine

Lahenduse käivitamiseks on vaja initsialiseerida lahenduse klass *FordFulkerson* tema ainsa konstruktori abil, mille sisendparameetriteks on:

- Sidus lihtgraaf - *graph*
- Lähtetipp - *source*
- Lõpptipp ehk suue – *sink*

Kõikide sisendparameetrite kirjeldused, funktsioonide töökirjeldus on esitatud ka vastavalt nõuetele programmi *JavaDoc* dokumentatsioonis.

Peale algoritmi lahendava klassi initsialiseerimist tuleks loodud isendil kutsuda välja meetod *Execute(),* mille tulemusena programm alustab tööd ja väljastab tehtud arvutuste tulemused teksti vormis terminali.

5.2 Väljund

Programm toob töö käigus välja tagasisidena kasutajale kõik olulise informatsiooni. Enne töö alustamist esitatakse kasutajale tema valitud lähe ja suue. Algoritmi põhitööni jõudes tuuakse kasutajale välja iga läbitud tee visuaalne esitus, misjärel näidatakse ka

läbitud tee maksimaalset läbilaskevõimet. Väljundi lõpetab algoritmi töö tulemus, ehk kahe valitud tipu vahel olev maksimaalne läbilaskevõime.

```
Source node: v2
Destination node: v6
Traversed path: av2_v6(C: 3, F: 3) Path capacity: 3
Traversed path: av2_v4(C: 3, F: 3) -> av4_v6(C: 8, F: 3) Path capacity: 3
Traversed path: av2_v1(C: 9, F: 5) -> av1_v6(C: 5, F: 5) Path capacity: 5
Traversed path: av2_v1(C: 9, F: 9) -> av1_v4(C: 4, F: 4) -> av4_v6(C: 8, F:
7) Path capacity: 4
Maximum capacity: 15
```

Näide: Programmi väljund, kus lähtepuntiks oli tipp *v2*, suudmeks määratud tipp *v6.* Algoritm läbis kokku 4 teed ning maksimaalseks läbilaskevõimeks leiti 15.

6 Testimisplaan

Algoritmi testimisel on oluline veenduda algoritmi tulemuse korrektsuseks. Ülesande lahenduspõhjal eksisteerib eelnev suvaliste lihtgraafide genereerimise meetod, mida autor plaanib testimisel rakendada. Muudatusena lihtgraafide genereerimise algoritmis loob autor orienteeritud graafe.

Esiteks luuakse viis erineva mahuga suvalist graafi, millega algoritm teeb oma arvutused. Seejärel rekonstrueerib autor need graafid käsitsi usaldusväärses veebirakenduses, mis sarnaselt arvutab vajaliku tulemuse. Tulemuste klappimisel loetakse test õnnestunuks ning ebakõlade puhul ebaõnnestunuks.

Tulemuste verifitseerimiseks kasutas autor järgnevat veebirakendust: *https://www-m9.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_en.html*

6.1 Testandmed

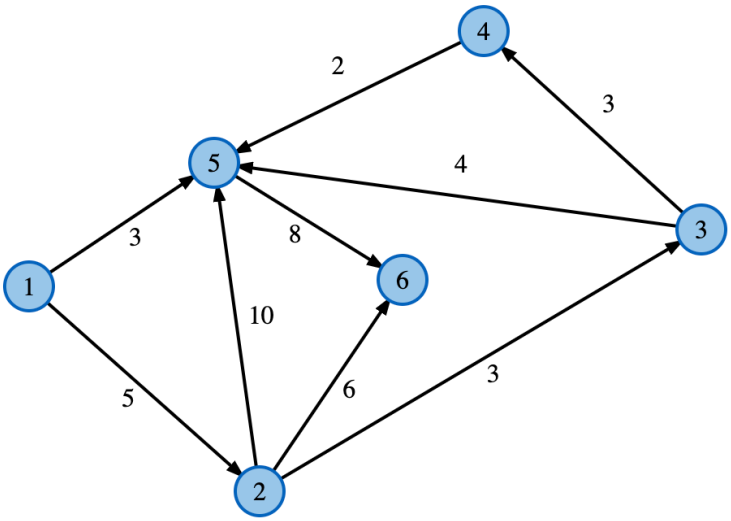
Testandmete ja nende tulemuste analüüsimisel tuleb silmas pidada, et kuna autori tulemused on kontrollitud teises veebirakenduses, siis rakenduses kasutatav algoritm ei pruugi valida ilmtingimata samu ahelaid, mis autori algoritm. Lõpptulemusena leitav maksimaalne läbilaskevõime peab aga mõlemal juhul olema sama.

Testandmeid luuakse juhuslikult ning testgraafi esitus on iga testjuhtumi puhul toodud välja joonisena. Igale servale antakse juhuslik maksimaalne läbilaskevõime väärtus 1-10-ni.

Juhuslikkuse tõttu ei pruugi graafide ühendused olla nende numbriliste nimede järgi väga hästi kooskõlas, kuid see nüanss esindab reaalseid juhtumeid ilmselt paremini.

Iga testgraafi juures on näha tema graafiline esitus ning programmi väljund antud graafi puhul. Peale testjuhtumite tutvustamist on igale testjuhtumile vastav kontrolltulemus veebirakenduses ning tulemuste võrdlus, millele tuginedes otsustatakse, kas test on õnnestunud või ebaõnnestunud.

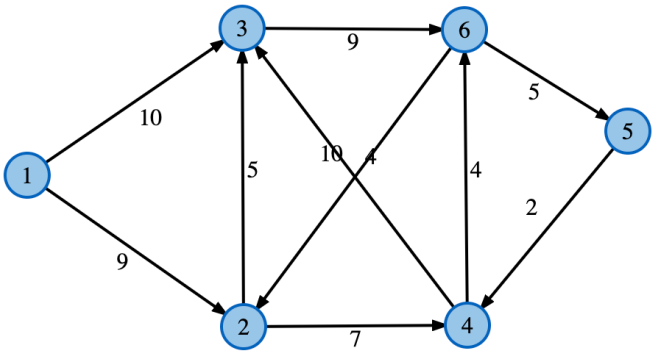
6.1.1 Esimene testgraaf



Väljund:

```
Source node: v1
Destination node: v6
Traversed path: av1_v5(C: 3, F: 3) -> av5_v6(C: 8, F: 3) Path capacity: 3
Traversed path: av1_v2(C: 5, F: 5) -> av2_v6(C: 6, F: 5) Path capacity: 5
Maximum capacity: 8
```

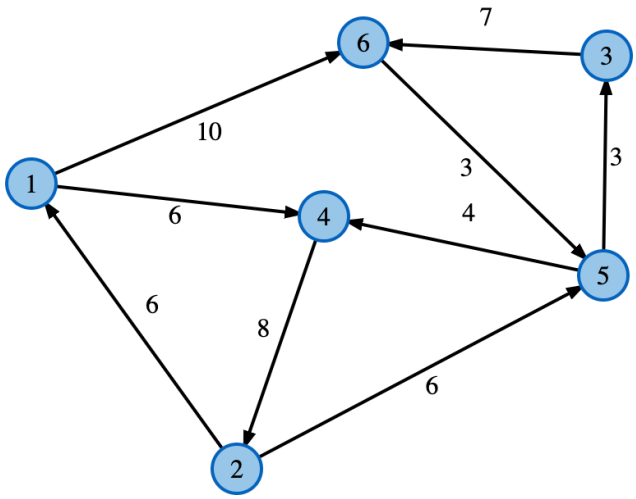
6.1.2 Teine testgraaf



Väljund:

```
Source node: v1
Destination node: v6
Traversed path: av1_v3(C: 10, F: 9) -> av3_v6(C: 9, F: 9) Path capacity: 9
Traversed path: av1_v2(C: 9, F: 4) -> av2_v4(C: 7, F: 4) -> av4_v6(C: 4, F:
4) Path capacity: 4
Maximum capacity: 13
```

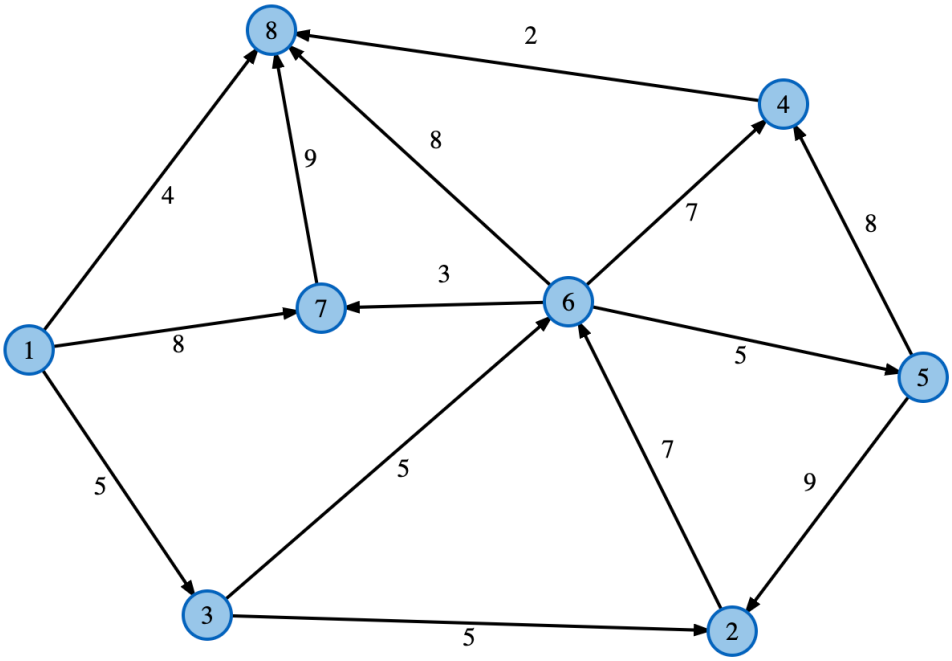
6.1.3 Kolmas testgraaf



Väljund:

```
Source node: v1
Destination node: v6
Traversed path: av1_v6(C: 10, F: 10) Path capacity: 10
Traversed path: av1_v4(C: 6, F: 3) -> av4_v2(C: 8, F: 3) -> av2_v5(C: 6, F:
3) -> av5_v3(C: 3, F: 3) -> av3_v6(C: 7, F: 3) Path capacity: 3
Maximum capacity: 13
```

6.1.4 Neljas testgraaf



Väljund:

```
Source node: v1
Destination node: v8
Traversed path: av1_v8(C: 4, F: 4) Path capacity: 4
Traversed path: av1_v7(C: 8, F: 8) -> av7_v8(C: 9, F: 8) Path capacity: 8
Traversed path: av1_v3(C: 5, F: 5) -> av3_v6(C: 5, F: 5) -> av6_v8(C: 5, F:
5) Path capacity: 5
Maximum capacity: 17
```

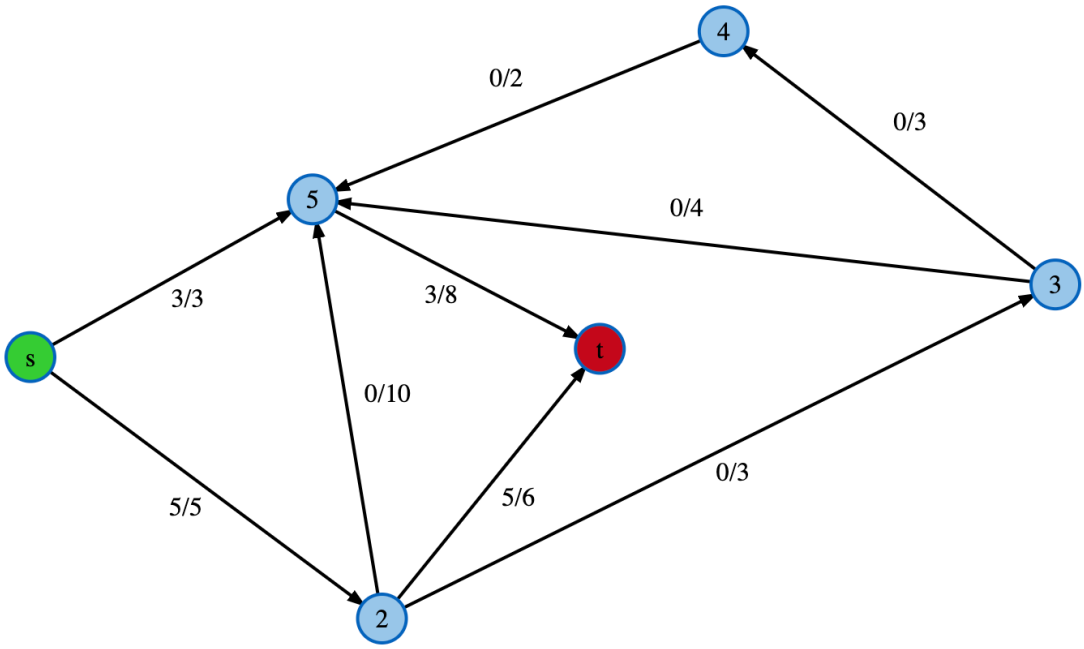
6.1.5 Viies testgraaf

Viienda testgraafi puhul luuakse suvaline 2000 tipuga graaf, ning valitakse kaks suvalist tippu. Seejärel kontrollitakse algoritmi töö pikkust. Algoritmi tulemuse usaldusväärsus tugineb eelnevale neljale testgraafile, kuna suvalist graafi luues tulemust kontrollida on skontekstis ebaproportsionaalselt aeganõudev.

6.2 Testimise tulemused

6.2.1 Esimene testgraaf

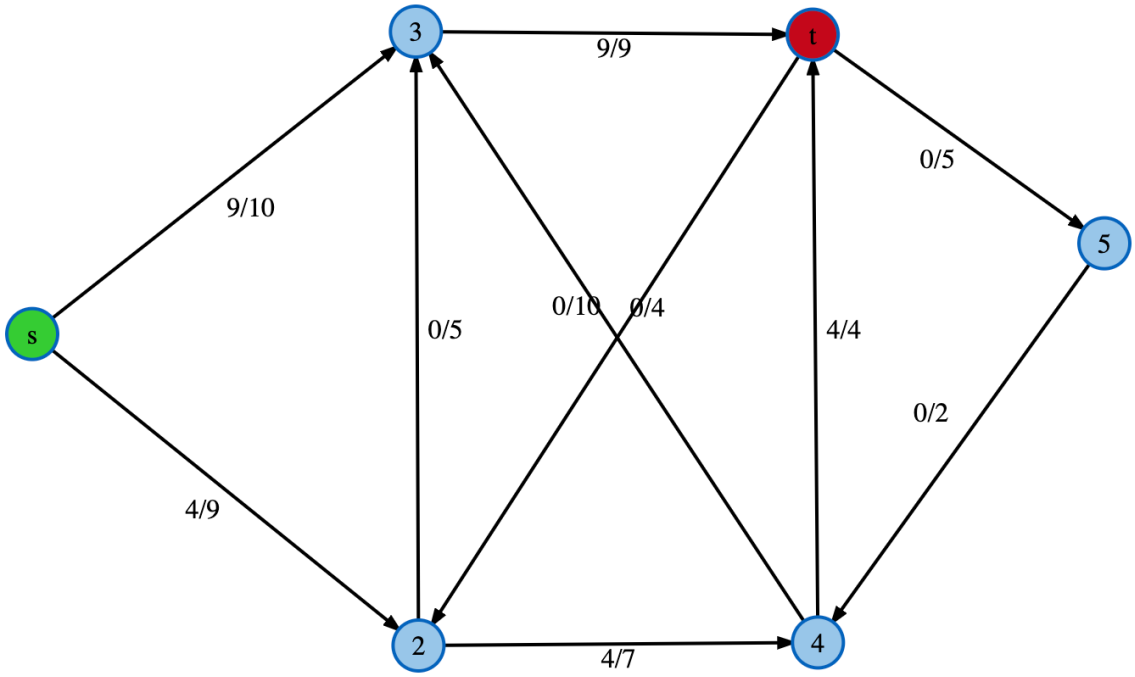
Esimese testgraafi puhul leidis veebirakendus maksimaalse läbilaskevõime väärtuseks 8. Autori loodud lahendus sai samuti tulemuseks 8. Test loetakse õnnestunuks.



Joonis 2. Esimese testgraafi kontroll, kus *s* on tipp *v1* ja *t* on tipp *v6.* (Fischer, 2015-2017)

6.2.2 Teine testgraaf

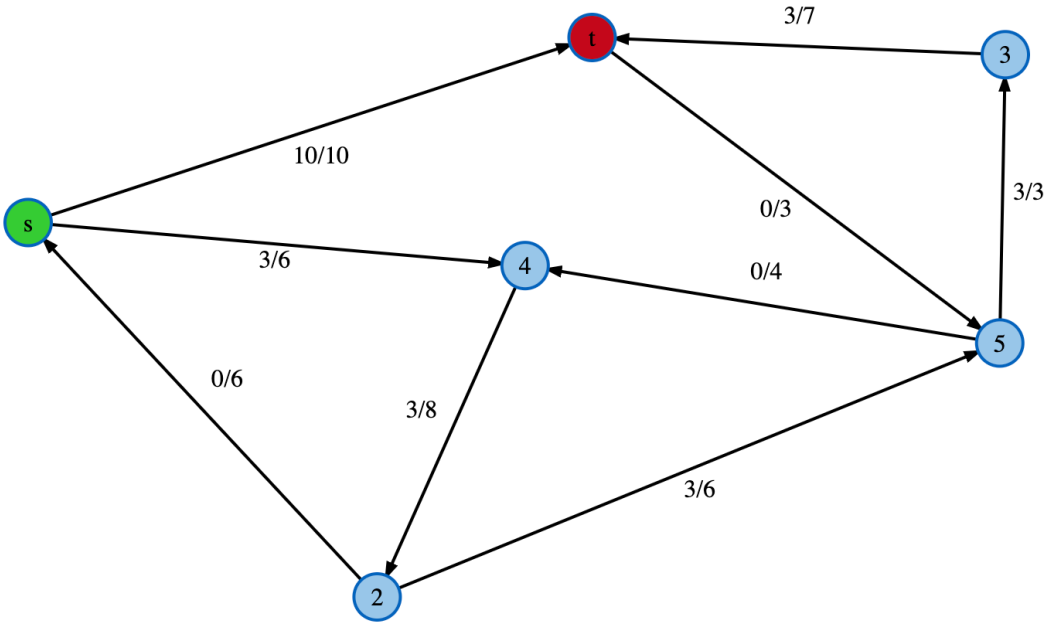
Teise testgraafi puhul leidis veebirakendus maksimaalse läbilaskevõime väärtuseks 13. Autori loodud lahendus sai samuti tulemuseks 13. Test loetakse õnnestunuks.



Joonis 3. Teise testgraafi kontroll, kus *s* on tipp *v1* ja *t* on tipp *v6.* (Fischer, 2015-2017)

6.2.3 Kolmas testgraaf

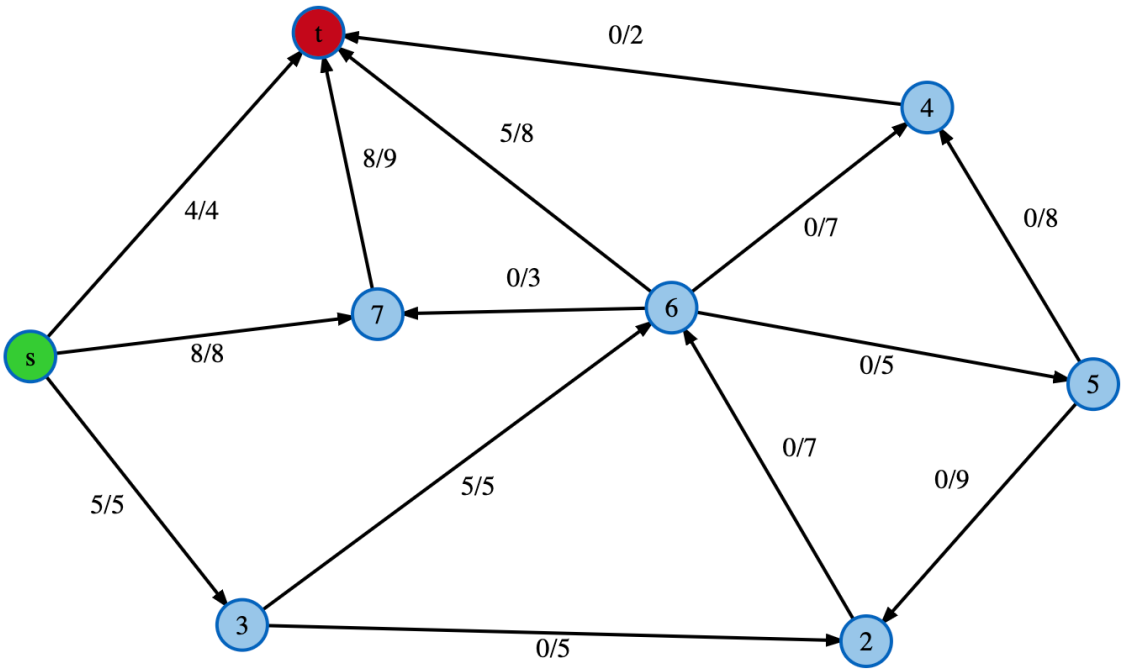
Kolmanda testgraafi puhul leidis veebirakendus maksimaalse läbilaskevõime väärtuseks 13. Autori loodud lahendus sai samuti tulemuseks 13. Test loetakse õnnestunuks.



Joonis 4. Teise testgraafi kontroll, kus *s* on tipp *v1* ja *t* on tipp *v6.* (Fischer, 2015-2017)

6.2.4 Neljas testgraaf

Neljanda testgraafi puhul leidis veebirakendus maksimaalse läbilaskevõime väärtuseks 17. Autori loodud lahendus sai samuti tulemuseks 17. Test loetakse õnnestunuks.



Joonis 4. Teise testgraafi kontroll, kus *s* on tipp *v1* ja *t* on tipp *v8.* (Fischer, 2015-2017)

6.2.5 Viies testgraaf

Viiendaks testgraafiks on suvaline graaf, millel on 2000 tippu. Abimeetodit kasutades määrame mingi suvalise tipu selles graafis, mis oleks võimalikult kaugel. Antud testi tulemuse usaldusväärsuse baseerume eelneva nelja testi tulemuse najale, seega sel juhul mõõdame ainult algoritmi lahendusaega. **Väljund:**

```
Source node: v1

Destination node: v2000

Traversed path: av1_v904(C: 8, F: 2) -> av904_v1256(C: 10, F: 2) -> av1256_v2000(C: 2, F: 2) Path capacity: 2

Traversed path: av1_v1912(C: 2, F: 2) -> av1912_v1948(C: 6, F: 2) -> av1948_v2000(C: 10, F: 2) Path capacity: 2

Traversed path: av1_v1949(C: 2, F: 2) -> av1949_v1942(C: 4, F: 2) -> av1942_v2000(C: 4, F: 2) Path capacity: 2

Traversed path: av1_v221(C: 3, F: 3) -> av221_v923(C: 9, F: 3) -> av923_v1368(C: 9, F: 3) -> av1368_v2000(C: 4, F: 3) Path capacity: 3

Traversed path: av1_v1472(C: 3, F: 3) -> av1472_v1223(C: 10, F: 3) -> av1223_v1197(C: 5, F: 3) -> av1197_v2000(C: 3, F: 3) Path capacity: 3

Traversed path: av1_v1818(C: 3, F: 2) -> av1818_v627(C: 2, F: 2) -> av627_v1026(C: 5, F: 2) -> av1026_v2000(C: 2, F: 2) Path capacity: 2

Traversed path: av1_v1818(C: 3, F: 3) -> av1818_v1145(C: 4, F: 1) -> av1145_v1330(C: 8, F: 1) -> av1330_v2000(C: 2, F: 1) Path capacity: 1

Traversed path: av1_v904(C: 8, F: 7) -> av904_v1256(C: 10, F: 7) -> av1256_v1540(C: 5, F: 5) -> av1540_v2000(C: 8, F: 5) Path capacity: 5

Traversed path: av1_v1302(C: 10, F: 2) -> av1302_v172(C: 10, F: 2) -> av172_v532(C: 3, F: 2) -> av532_v2000(C: 2, F: 2) Path capacity: 2

Traversed path: av1_v1302(C: 10, F: 4) -> av1302_v243(C: 2, F: 2) -> av243_v1749(C: 3, F: 2) -> av1749_v2000(C: 5, F: 2) Path capacity: 2

Traversed path: av1_v1302(C: 10, F: 5) -> av1302_v1145(C: 8, F: 1) -> av1145_v1330(C: 8, F: 2) -> av1330_v2000(C: 2, F: 2) Path capacity: 1
```

Traversed path: av1_v1302(C: 10, F: 10) -> av1302_v220(C: 8, F: 5) -> av220_v1790(C: 10, F: 5) -> av1790_v2000(C: 10, F: 5) Path capacity: 5

Traversed path: av1_v546(C: 2, F: 2) -> av546_v396(C: 8, F: 2) -> av396_v1969(C: 5, F: 2) -> av1969_v2000(C: 4, F: 2) Path capacity: 2

Traversed path: av1_v1750(C: 10, F: 2) -> av1750_v52(C: 6, F: 2) -> av52_v1749(C: 2, F: 2) -> av1749_v2000(C: 5, F: 4) Path capacity: 2

Traversed path: av1_v1750(C: 10, F: 7) -> av1750_v1061(C: 6, F: 5) -> av1061_v977(C: 6, F: 5) -> av977_v2000(C: 5, F: 5) Path capacity: 5

Traversed path: av1_v1750(C: 10, F: 10) -> av1750_v1684(C: 6, F: 3) -> av1684_v194(C: 9, F: 3) -> av194_v2000(C: 6, F: 3) Path capacity: 3

Traversed path: av1_v1403(C: 6, F: 3) -> av1403_v876(C: 4, F: 3) -> av876_v1399(C: 3, F: 3) -> av1399_v2000(C: 5, F: 3) Path capacity: 3

Traversed path: av1_v1403(C: 6, F: 5) -> av1403_v1941(C: 2, F: 2) -> av1941_v820(C: 5, F: 2) -> av820_v2000(C: 2, F: 2) Path capacity: 2

Traversed path: av1_v1403(C: 6, F: 6) -> av1403_v871(C: 7, F: 1) -> av871_v341(C: 8, F: 1) -> av341_v2000(C: 6, F: 1) Path capacity: 1

Traversed path: av1_v1646(C: 9, F: 2) -> av1646_v966(C: 8, F: 2) -> av966_v1998(C: 2, F: 2) -> av1998_v2000(C: 10, F: 2) Path capacity: 2

Traversed path: av1_v1646(C: 9, F: 3) -> av1646_v1553(C: 4, F: 1) -> av1553_v1749(C: 6, F: 1) -> av1749_v2000(C: 5, F: 5) Path capacity: 1

Traversed path: av1_v1646(C: 9, F: 9) -> av1646_v1562(C: 8, F: 6) -> av1562_v1999(C: 7, F: 6) -> av1999_v2000(C: 8, F: 6) Path capacity: 6

Traversed path: av1_v1362(C: 7, F: 1) -> av1362_v508(C: 9, F: 1) -> av508_v1368(C: 4, F: 1) -> av1368_v2000(C: 4, F: 4) Path capacity: 1

Traversed path: av1_v1362(C: 7, F: 6) -> av1362_v755(C: 8, F: 5) -> av755_v24(C: 5, F: 5) -> av24_v2000(C: 8, F: 5) Path capacity: 5

Traversed path: av1_v1362(C: 7, F: 7) -> av1362_v797(C: 5, F: 1) -> av797_v1942(C: 8, F: 1) -> av1942_v2000(C: 4, F: 3) Path capacity: 1

Traversed path: av1_v937(C: 8, F: 6) -> av937_v1531(C: 6, F: 6) -> av1531_v1948(C: 9, F: 6) -> av1948_v2000(C: 10, F: 8) Path capacity: 6

Traversed path: av1_v676(C: 8, F: 5) -> av676_v871(C: 8, F: 5) -> av871_v341(C: 8, F: 6) -> av341_v2000(C: 6, F: 6) Path capacity: 5

```
Traversed path: av1_v676(C: 8, F: 8) -> av676_v220(C: 5, F: 3) -> av220_v1790(C: 10, F: 8) -> av1790_v2000(C: 10, F: 8) Path capacity: 3

Traversed path: av1_v1866(C: 6, F: 3) -> av1866_v1765(C: 7, F: 3) -> av1765_v24(C: 8, F: 3) -> av24_v2000(C: 8, F: 8) Path capacity: 3

Traversed path: av1_v903(C: 3, F: 3) -> av903_v1920(C: 3, F: 3) -> av1920_v1998(C: 7, F: 3) -> av1998_v2000(C: 10, F: 5) Path capacity: 3

Traversed path: av1_v1134(C: 6, F: 3) -> av1134_v241(C: 9, F: 3) -> av241_v1450(C: 6, F: 3) -> av1450_v1540(C: 5, F: 3) -> av1540_v2000(C: 8, F: 8) Path capacity: 3

Traversed path: av1_v1134(C: 6, F: 5) -> av1134_v241(C: 9, F: 5) -> av241_v1093(C: 10, F: 2) -> av1093_v1399(C: 2, F: 2) -> av1399_v2000(C: 5, F: 5) Path capacity: 2

Traversed path: av1_v1134(C: 6, F: 6) -> av1134_v241(C: 9, F: 6) -> av241_v421(C: 3, F: 1) -> av421_v1198(C: 10, F: 1) -> av1198_v2000(C: 3, F: 1) Path capacity: 1

Traversed path: av1_v904(C: 8, F: 8) -> av904_v1280(C: 5, F: 1) -> av1280_v94(C: 6, F: 1) -> av94_v836(C: 6, F: 1) -> av836_v2000(C: 3, F: 1) Path capacity: 1

Traversed path: av1_v937(C: 8, F: 8) -> av937_v744(C: 8, F: 2) -> av744_v1870(C: 3, F: 2) -> av1870_v544(C: 7, F: 2) -> av544_v2000(C: 10, F: 2) Path capacity: 2

Traversed path: av1_v1866(C: 6, F: 6) -> av1866_v1323(C: 8, F: 3) -> av1323_v869(C: 8, F: 3) -> av869_v1996(C: 3, F: 3) -> av1996_v2000(C: 8, F: 3) Path capacity: 3

Maximum capacity: 96

Execution time (ms): 299
```

Programmi tööajaks 2000 tipu ja 8000 serva puhul kujunes 299 ms ehk. 0,299 sekundit.

7 Kokkuvõte

Autor lahendas kodutöö nr. 6 valitud lähteülesande vastavalt etteantud Ford-Fulkersoni algoritmile. Algoritmi siseselt kasutas autor toetava algoritmina *breath-first search* algoritmi, mille eesmärgiks on leida võimalikult graafil lühike tee ühest sisendiks antud lähtetipust kuni valitud lõpp-punktini.

Lahenduse testimisel kasutas autor veebirakendusena tehtud realisatsiooni algoritmist, mis võimaldab tulemuse graafilist esitust testandmete visualiseerimiseks ja lahtimõtestamiseks. Autori esitatud lahendus läbis edukalt kõik ettevalmistatud testpunktid, ning võib järeldada, et lahendus rahuldab algset ülesandepüstitust.

Lõppanalüüsis jõudis autor teadmiseni, et kasutatud lahenduste kombinatsiooni (Ford-Fulkerson ja *breadth-first search)* nimetatakse ka Edmonds-Karp algoritmiks. (Buldas, Laud, & Willemson, 2008)

Raskuspunktina toob autor välja töös sobiva tee otsimise algoritmi realiseerimise kasutuses olevatele andmestruktuuridele tuginedes. Samuti oli autori jaoks töö hea meeldetuletus, et enne koodi kirjutamist tasub alati lõppeesmärk ja protsess endale täielikult lahti mõtestada, et töö edeneks sujuvamalt, kiiremalt ning arusaadavamalt.

Kasutatud kirjandus

Buldas, A., Laud, P., & Willemson, J. (2008). *Graafid.* Tartu: Tartu Ülikooli Kirjastus.

Fischer, Q. (2015-2017). *Ford-Fulkerson-Algorithm*. Allikas: Ford-Fulkerson-Algorithm: https://www-m9.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_en.html

Fulkerson, D. R., & Ford, L. R. (1955). *Maximal flow through a network .* Allikas: Yale University: http://www.cs.yale.edu/homes/lans/readings/routing/ford-max_flow-1956.pdf

Lisa 1 – Programmi lähtekood

```
import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

/** 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");
    g.createRandomSimpleGraph (2000, 6000);
    Vertex selected = g.first;

    while(selected.next != null) {
      selected = selected.next;
    }
    FordFulkerson alg = new FordFulkerson(g, g.first, selected);
    alg.Execute();
  }

  /** Ford-Fulkerson algorithm
   *  Objective: Find the maximum flow capacity between two nodes in a graph
   *  Task in estonian:
   *  "On antud sidus lihtgraaf, mille iga serva jaoks on teada selle läbilaskevõime ("toru jämedus",
mittenegatiivne)."
   *  "Koostada meetod, mis leiab kahe etteantud tipu vahelise maksimaalse läbilaskevõime Ford-
Fulkersoni meetodil."
   *  @author Jan-Erik Kalmus
   */

  class Vertex {

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

    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;
  }

  /** Determines all arcs of Vertex at hand and returns them
   * @return List of all arcs of subject Vertex
   */
  private List<Arc> getArcsInVertex() {
    List<Arc> collection = new ArrayList<>();
    Arc subject = this.first;

    do {
      collection.add(subject);
      subject = subject.next;
    } while(subject != null);

    return collection;
  }

  /** Determines whether a specific vertex is connected to this vertex
   * @param target - Vertex expected to be connected with vertex at hand
   * @return Is target vertex connected to this vertex
   */
  private boolean vertexInArcs(Vertex target) {
    List<Arc> arcs = this.getArcsInVertex();

    for (Arc arc : arcs) {
      if(arc.target.equals(target) && arc.getCapacity() != 0)
        return true;
    }

    return false;
  }
}


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

  private String id;
  private Vertex target;
  private Arc next;
  private final int info = 0;
  private final int capacity;
  private int flow = 0;

  Arc (String s, Vertex v, Arc a, int c) {
    id = s;
    target = v;
    next = a;
    capacity = c;
  }
```

```
  /** Calculates available capacity of the arc
   * @return Current maximum capacity
   */
  public int getCapacity() {
    return capacity - flow;
  }

  Arc (String s) {
    this (s, null, null, ThreadLocalRandom.current().nextInt(2, 10 + 1));
  }

  @Override
  public String toString() {
    return id + String.format("(C: %d, F: %d)", capacity, flow);
  }
}


class Graph {

  private String id;
  private Vertex first;
  private int info = 0;

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

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

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

```
}

public Vertex createVertex (String vid) {
  Vertex res = new Vertex (vid);
  res.next = first;
  first = res;
  return res;
}

public Arc createArc (String aid, Vertex from, Vertex to) {
  Arc res = new Arc (aid);
  res.next = from.first;
  from.first = res;
  res.target = to;
  return res;
}

/**
 * Create a connected undirected random tree with n vertices.
 * Each new vertex is connected to some random existing vertex.
 * @param n number of vertices added to this graph
 */
public void createRandomTree (int n) {
  if (n <= 0)
    return;
  Vertex[] varray = new Vertex [n];
  for (int i = 0; i < n; i++) {
    varray [i] = createVertex ("v" + String.valueOf(n-i));
    if (i > 0) {
      int vnr = (int)(Math.random()*i);
      createArc ("a" + varray [vnr].toString() + "_"
        + varray [i].toString(), varray [vnr], varray [i]);
      createArc ("a" + varray [i].toString() + "_"
        + varray [vnr].toString(), varray [i], varray [vnr]);
    } else {}
  }
}

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

```
      }
      v = v.next;
    }
    return res;
  }

  /**
   * Create a connected simple (undirected, no loops, no multiple
   * arcs) random graph with n vertices and m edges.
   * @param n number of vertices
   * @param m number of edges
   */
  public void createRandomSimpleGraph (int n, int m) {
    if (n <= 0)
      return;
    if (n > 2500)
      throw new IllegalArgumentException ("Too many vertices: " + n);
    if (m < n-1 || m > n*(n-1)/2)
      throw new IllegalArgumentException
        ("Impossible number of edges: " + m);
    first = null;
    createRandomTree (n);      // n-1 edges created here
    Vertex[] vert = new Vertex [n];
    Vertex v = first;
    int c = 0;
    while (v != null) {
      vert[c++] = v;
      v = v.next;
    }
    int[][] connected = createAdjMatrix();
    int edgeCount = m - n + 1;  // remaining edges
    while (edgeCount > 0) {
      int i = (int)(Math.random()*n);  // random source
      int j = (int)(Math.random()*n);  // random target
      if (i==j)
        continue;  // no loops
      if (connected [i][j] != 0 || connected [j][i] != 0)
        continue;  // no multiple edges
      Vertex vi = vert [i];
      Vertex vj = vert [j];
      createArc ("a" + vi.toString() + "_" + vj.toString(), vi, vj);
      connected [i][j] = 1;
      createArc ("a" + vj.toString() + "_" + vi.toString(), vj, vi);
      connected [j][i] = 1;
      edgeCount--;  // a new edge happily created
    }
  }
}

/** Ford-Fulkerson algorithm processing class
 *  Contains alrogithm rep
 */
class FordFulkerson {

  private final Vertex start;
  private final Vertex sink;
  private final Graph graph;
  private Path currentPath;
```

```
    /** Constructor for Ford-Fulkerson algorithm processor
     * @param graph - Subject graph for algorithm
     * @param source - Source node of Ford-Fulkerson algorithm
     * @param sink - Sink node of Ford-Fulkerson algorithm
     */

    public FordFulkerson(Graph graph, Vertex source, Vertex sink) {
      this.start = source;
      this.sink = sink;
      this.graph = graph;
    }


    private class Path {
      private final LinkedList<Arc> arcs;
      private int maximumCapacity;

      /**
       * Supporting class to designate an augmented path in graph
       */

      public Path () {
        arcs = new LinkedList<>();
        maximumCapacity = Integer.MAX_VALUE;
      }

      /**
       * Add an arc to the current path
       */
      public void pushArc (Arc addition) {
        arcs.add(addition);
        recalculateArcCapacity();
      }

      /**
       *  Recalculates maximum capacity of the path based on maximum capacities of
       *  all of the arcs present in the path.
       */
      private void recalculateArcCapacity() {
        for(var arc : arcs) {
          if(arc.getCapacity() < this.maximumCapacity) {
            this.maximumCapacity = arc.getCapacity();
          }
        }
      }

      /**
       *  Updates the flows of all the arcs present in the path according to the bottleneck value
       * @throws IllegalStateException - An arc does not have the required capacity for the
predetermined paths bottleneck
       */

      private void updateFlows() {
        for(var arc : arcs) {
          if(arc.getCapacity() - this.maximumCapacity < 0) {
            String errorMessage = String.format("Error in updating capacity for arc: %s. Arc
capacity: %d. Path capacity: %d", arc.toString(), arc.getCapacity(), this.maximumCapacity);
            throw new IllegalStateException(errorMessage);
```

```
      }

      arc.flow += this.maximumCapacity;
    }
  }

  /**
   *  Getter method for maximum capacity of path
   *  @return Maximum capacity of the path
   */

  public int getMaximumCapacity () {
    return maximumCapacity;
  }

  /**
   * Builds string representation of the path from starting node to the final node
   * @return String representation of path
   */

  public String toString() {
    StringBuilder builder = new StringBuilder();

    var iterator =  arcs.iterator();

    while(iterator.hasNext()) {
      builder.append(iterator.next().toString());

      if(iterator.hasNext()) {
        builder.append(" -> ");
      }
    }

    builder.append(String.format(" Path capacity: %d", maximumCapacity));

    return builder.toString();
  }
}

/**
 * Calculate maximum flow between two random points using Ford-Fulkerson algorithm
 * Ford-Fulkerson algorithm:
 * - Determine whether an legitimate path to destination exists
 * - Fetch maximum flow of this path - ultimately determined by its bottleneck capacity
 * - Form a sum of said flows
 * @return maximum flow capacity between designated source and sink
 */
public int Execute() {
  // Initialize maximum flow
  int maximumFlow = 0;

  System.out.println("Source node: " + this.start.toString());
  System.out.println("Destination node: " + this.sink.toString());

  // Loop while we have legitimate paths
  while(findPath()) {

    // Update capacities of all arcs in the path according to its bottleneck
    currentPath.updateFlows();
```

```
    System.out.println("Traversed path: " + currentPath.toString());

    // Append maxmimum flow of current path
    maximumFlow += currentPath.getMaximumCapacity();
  }

  System.out.println("Maximum capacity: " + maximumFlow);
  // Return result
  return maximumFlow;
}


/**
 * Supporting class for BFS queue
 * Keeps reference of subject Vertex and the previous Vertex associated to it
 */
private class QueueItem {
  public Vertex previous;
  public Vertex subject;

  public QueueItem(Vertex p, Vertex s) {
    previous = p;
    subject = s;
  }
}

/**
 * Find and process an untraversed path. Determine bottleneck of this path in process
 * @return path found
 */
private boolean findPath () {

  // Initialize queues and traversed vertices list
  LinkedList<QueueItem> queue = new LinkedList<QueueItem>();
  List<Vertex> traversedVertices = new ArrayList<Vertex>();
  List<Vertex> previous = new ArrayList<Vertex>();

  // Start from the beginning
  queue.add(new QueueItem(this.start, this.start));

  // BFS
  while(!queue.isEmpty()) {
    QueueItem qItem = queue.poll();
    Vertex subject = qItem.subject;
    boolean connectedToDestination = subject.vertexInArcs(this.sink);

    // Continue if we have explored this vertex already
    if(traversedVertices.contains(subject)) {
      continue;
    }

    traversedVertices.add(subject);
    previous.add(qItem.previous);

    var arcs = subject.getArcsInVertex();

    // Check the neighbors of current vertex
    for (Arc arc : arcs) {
```

```
        // Skip if arc lacks capacity -> Can not go there
        if(arc.getCapacity() == 0) {
          continue;
        }

        // This vertex has a valid path to the sink node
        if(connectedToDestination && !arc.target.equals(sink)) {
          continue;
        }

        // If we have found a path to the subject, finish
        if(arc.target.equals(sink)) {
          traversedVertices.add(arc.target);
          previous.add(subject);
          return path_found(traversedVertices, previous);
        }

        // If the target of this arc has not been discovered yet, add to queue
        if(!traversedVertices.contains(arc.target)) {
          queue.add(new QueueItem(subject, arc.target));
        }
      }
    }

    return path_found(traversedVertices, previous);
  }

  /** Attempts to reconstruct road based on BFS result, ultimately determines whether connection
exists
   * @param v - list of traversed vertices
   * @param p - list of previous vertices associated with traversed vertices at equal list index
   * @return Evaluation of legitimate path from source to sink being found
   */
  public boolean path_found(List<Vertex> v, List<Vertex> p) {
    // Initialize currentPath
    currentPath = new Path();

    // To keep track of path while backtracking from end
    List<Arc> reversePath = new ArrayList<>();

    // If the relation between the amount of vertices and previous vertices is not correct, path was
not found
    if(v.size() != p.size()) {
      return false;
    }

    // The loop did not break at the final vertex
    if (!v.get(v.size() - 1).equals(this.sink)) {
      return false;
    }

    // Final vertex --> Expectedly the sink
    Vertex current = v.get(v.size() - 1);

    // Start backtracking the path
    for ( var i = v.size() - 1 ; i >= 0; i--) {
      Vertex previous = p.get(i);
```

```
        // Can only happen with the first one
        if(previous.equals(current)) {
          break;
        }

        if(!previous.vertexInArcs(current)) {
          return false;
        }

        boolean found = false;

        // Find our suitable arc and add it
        for (var arc : previous.getArcsInVertex()) {
          if(arc.target.equals(current)) {
            reversePath.add(arc);
            found = true;
            current = previous;

            i = v.indexOf(current) + 1; // Point index at correct location
          }
        }

        // If a suitable arc was not found,
        if(!found) {
          return false;
        }
      }

      // Reverse the reversed paths list
      Collections.reverse(reversePath);

      // Construct path
      for(Arc arc : reversePath) {
        currentPath.pushArc(arc);
      }

      return true;
    }

  }

}
```