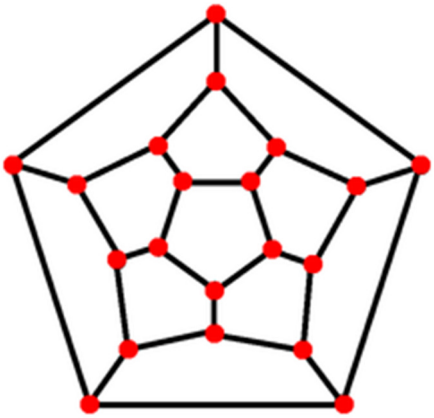
10 GRAAFITEOORIA



Raamatu esimeses osas oli juttu põhilistest graafiteooria algoritmidest, kuid see andis valdkonnast ainult väikese maitse suhu. Tegelikult moodustab graafiteooria nii arvutiteadusest üldiselt kui ka olümpiaadidel esinevates ülesannetes märksa suurema osa. Viimastel aastatel on sageli tulnud ette võistlusi, kus graafiülesanded moodustavad ülesandekomplektist enam kui poole. Seega tuleme nüüd teema juurde tõsisemalt tagasi.

Kõrvaltoodud joonis illustreerib veel üht graafiteooria uurimisteemat – planaarseid graafe. Graafi nimetatakse planaarseks, kui selle tippe on võimalik joonistada nii, et ükski servade paar ei tarvitse omavahel lõikuda. Iga hulktahukas on teisendatav planaarseks graafiks ja joonisel on toodud korrapärase dodekaeedri selline teisendus nn Schlegeli diagrammi kujul.

10.1 FLOYD-WARSHALLI ALGORITM

Kuuendast peatükist tuttava Dijkstra algoritmiga saab leida lühima tee kahe tipu vahel graafis (kui selles ei esine negatiivse kaaluga tsükleid ega servi). Dijkstra algoritmi kasutades kogu graafi läbi käies saame lühimad teed mingist konkreetsest tipust A selle graafi igasse teise tippu. Mõnikord on aga vaja leida graafi iga tipupaari jaoks lühim tee nende tippude vahel, näiteks linnadevaheliste kauguste tabeli loomiseks.

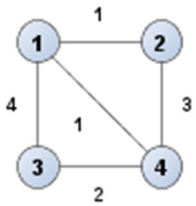
Üheks võimaluseks on muidugi kasutada Dijkstra algoritmi, valides alguspunktiks kõik graafi tipud. Dijkstra algoritmi keerukus on $O(T + S * logS)$,kus T on tippude ja S servade arv. Läbides seda algoritmi T korda, on keerukuseks $O(T^2 + T * S * logS)$. Kui graafis on vähe servi, on selline lahendus tõhus.

Teiseks võimaluseks on kasutada Floyd-Warshalli algoritmi. 1962. aastal avaldas tollal 26 aastane Illinois' Tehnoloogiainstituudis töötanud Robert Floyd algoritmi, mis kasutab dünaamilist planeerimist graafis kõigi lühimate teede leidmiseks. Sarnase algoritmini oli samal aastal jõudnud ka Stephen Warshall. Nagu ka Dijkstra algoritmis, ei tohi Floyd-Warshalli algoritmi puhul graafis olla negatiivseid tsükleid, negatiivsed servad on lubatud.

10.1.1 Ülesanne: tuleb hiljem

NB: Ülesanne lisandub hiljem

Vaatame, kuidas töötab Floyd-Warshalli algoritm järgmise graafi näitel:



10.1.2 Rekursioonivalem

Nagu mainitud, kasutab Floyd-Warshalli algoritm dünaamilist planeerimist. Üldine idee seisneb selles, et vaatame teed tipust i tippu j. Valime suvalise tipu k ning proovime seda lisada teesse tipus i tippu j. Kui teepikkus tipust i tippu j väheneb, siis oleme leidnud lühema tee tipust i tippu j läbi tipu k.

Kõigi teede leidmiseks tuleb seda teha iga tipupaari (i,j) ja iga tipu k jaoks. Saame järgneva rekursioonivalemi:

$$T(i,j,k) = \begin{cases} P(i,j), kui\ k = 0 \\ \min(T(i,j,k-1), T(i,k,k-1) + T(k,j,k-1)) \end{cases}$$

Siit on näha, et kõigepealt tuleks leida kõik teepikkused $k = 0$ korral, seejärel $k = 1$ korral jne, kuni $k = n$, kus n on tippude arv graafis.

10.1.3 Tabeli täitmine

Nagu ikka – kus on dünaamiline planeerimine, seal on tabel. Graafide juures on juba tuttavaks mõisteks kauguste maatriks kui graafi üks võimalikke esitusviise. Sellisel juhul on kauguste maatriksis vaid otseteed iga tipupaari (i,j) vahel. Sama tabelit pisut laiendades saamegi minimaalsete kauguste tabeli.

Esimesel sammul, kui $k = 0$, ongi valemi kohaselt tabelis ainult servade pikkused. Kui serv kahe tipu vahel puudub, on kauguseks lõpmatus:

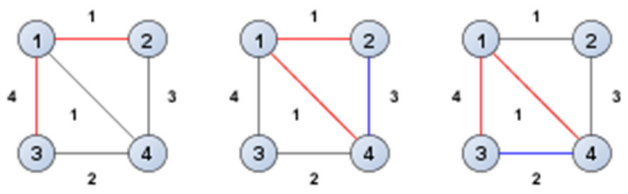
K=0	1	2	3	4
1	0	1	4	1
2		0	∞	3
3			0	2
4				0

Järgmisel sammul $k = 1$ proovime minna igast tipust igasse tippu läbi tipu 1.

$$T(2,3,1) = min\big(T(2,3,0), T\ (2,1,0) + T(1,3,0)\big) = min(\infty, 1+4) = \mathbf{5}$$

$$T(2,4,1) = min(T(2,4,0), T(2,1,0) + T(1,4,0)) = min(3, 1+1) = \mathbf{2}$$

$$T(3,4,1) = min(T(3,4,0), T(3,1,0) + T(1,4,0)) = min(2,4+1) = 2$$



Sinisega on märgitud otsetee tippude vahel (k=0), punasega tee läbi tipu 1 (k=1).

K=1	1	2	3	4
1	0	1	4	1
2		0	**5**	**2**
3			0	2
4				0

Läbi tipu 2:

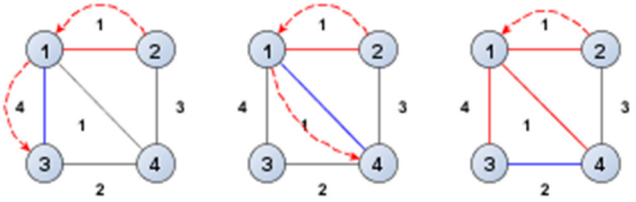
$$T(1,3,2) = min(T(1,3,1), T(1,2,1) + T(2,3,1)) = min(4, 1+5) = 4$$

$$T(1,4,2) = min(T(1,4,1), T(1,2,1) + T(2,4,1)) = \min(1, 1+2) = 1$$

$$T(3,4,2) = \min(T(3,4,1), T(3,2,1) + (2,4,1) = \min(2, 5+2) = 2$$

Siin on oluline silmas pidada, et kõige lühemad teed läbi tipu 1 ehk juhul, kui k=1 on meil juba leitud ja eelnevas tabelis kirjas! See, kas need tegelikult tippu 1 kasutasid või mitte, pole minimaalse tee kaalu leidmise juures enam oluline.

K=2	1	2	3	4
1	0	1	4	1
2		0	5	2
3			0	2
4				0



Sinisega on seni leitud parim tee ühest tipust teise. Kuna eelmisel etapil (k=1) ei parandanud ühtki teed (1,3), (1,4) ega (3,4) siis on need kõik juhtumisi üheservalised. Punasega on märgitud lühim tee läbi tipu 2, mis praeguseks teada. Näiteks tipust 3 tippu 4 minemiseks tuleb minna kõigepealt tippu 2 läbi tipu 1 ja seejärel tipust 2 tippu 4 (taas läbi tipu 1, kuna eelmisel sammul leidsime, et see on lühem kui otsetee tippude 2 ja 4 vahel).

Läbi tipu 3:

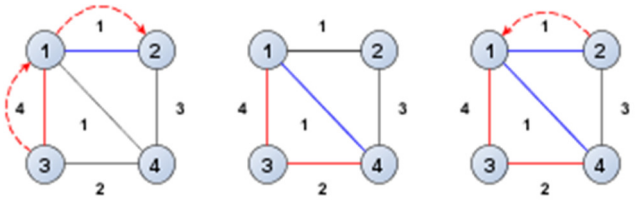
$$T(1,2,3) = \min\big(T(1,2,2), T(1,3,2) + T(3,2,2)\big) = \min(1, 4+5) = 1$$

$$T(1,4,3) = \min(T(1,4,2), T(1,3,2) + T(3,4,2)) = \min(1,\ 4+2) = 1$$

$$T(2,4,3) = \min\big(T(2,4,2), T(2,3,2) + T(3,4,2)\big) = \min(2,\ 5+2) = 2$$

Seekordki midagi ei parandata ja tabel jääb muutumatuks:

K=3	1	2	3	4
1	0	1	4	1
2		0	5	2
3			0	2
4				0



Kaugused tipu 3 kaudu. Kolmandal graafil on näha, et kasutame (üle) eelmisel sammul (k=1) leitud teed tipust 2 tippu 4 saamiseks.

Läbi tipu 4:

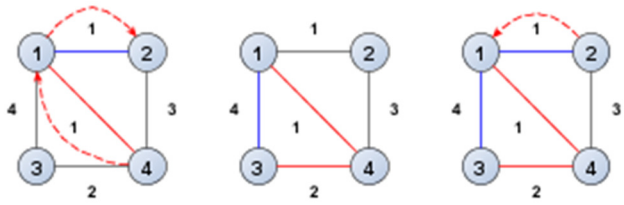
$$T(1,2,4) = \min(T(1,2,3), T(1,4,3) + T(4,2,3)) = \min(1, 1+2) = 1$$

$$T(1,3,4) = \min(T(1,3,3), T(1,4,3) + T(4,3,3)) = \min(4, 1+2) = \mathbf{3}$$

$$T(2,3,4) = \min(T(2,3,3), T(2,4,3) + T(4,3,3)) = \min(5, 2+2) = \mathbf{4}$$

Leidsime lühemad teed tippude 1 ja 3 ning 2 ja 3 vahel:

K=4	1	2	3	4
1	0	1	**3**	1
2		0	**4**	2
3			0	2
4				0



Funktsioon, mis seda teeb, on tõesti väga lihtne ja lühike:

```
int** maatriks;

void LeiaKaugused(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                if (maatriks[j][i] + maatriks[i][k] < maatriks[j][k]) {
                    maatriks[j][k] = maatriks[j][i] + maatriks[i][k];
                }
            }
        }
    }
}
```

Nagu siit näha, on üks oluline eelis Dijkstra algoritmi mitmekordse rakendamise ees just realisatsiooni kompaktsus.

10.1.4 Floyd-Warshalli algoritmi kasutusi

Väikese tipuhulgaga graafide korral on mõnikord kiirem kasutada Floyd-Warshalli algoritmi Dijkstra algoritmi asemel lühima tee leidmiseks kahe konkreetse tipu vahel. Keerukus on oluliselt suurem, kuid aeg koodi kirjutamiseks lühem ning kood ise lihtsam.

Küllaltki lihtsalt saab Floyd-Warshalli algoritmi muuta nii, et minimaalse tee asemel leiame maksimaalse. Praktikas enamasti just pikimat kaugust ei otsita, kuid kaaluks võib olla hoopis nt läbilaskevõime ja tahetakse leida suurima läbilaskevõimega teid/kanaleid. Sellisel juhul (teede pikkuse) summa asemel leida miinimum (kõige kehvema läbilaskevõimega koht ehk pudelikael otsustab, kui suur on kogu tee läbilaskevõime).

Lisaks saab selle algoritmiga tuvastada negatiivseid tsükleid ja leida kõige odavama (lühema) tsükli graafis. Selleks tuleks diagonaalil asuvad väärtused `[i,i]` seada väga suureks. Kui pärast algoritmi jooksutamist ei ole mõne `i` väärtuse puhul diagonaalil asuv `[i,i]` väärtus enam lõpmatu, siis leidub

graafis tsükkel, mis läbib tippu `i`. Kõige väiksema väärtusega selline tsükkel ongi minimaalne tsükkel graafis. Kui mõni väärtus `[i,i]` on negatiivne, siis on graafis negatiivne tsükkel.

Floyd-Warshalli algoritmi abil saab leida ka graafi **diameetri** ehk suurima võimaliku minimaalse kauguse kahe graafi tipu vahel. Selleks on loomulikult leitud minimaalsete kauguste tabelis maksimaalne väärtus. Eelpool vaadeldud graafi diameeter on 4.

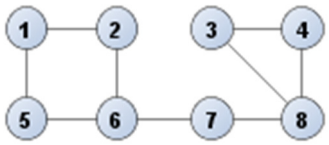
Lahendiks olevast kauguste tabelist saab lihtsalt näha, kas kahe punkti vahel üldse leidub tee või mitte. Kui ainult seda infot vaja on, võimegi tabelis meeles pidada vaid tee leidumist (või mitteleidumist). Taas on see praktiline ainult siis, kui tippude hulk on väike.

Sel moel aga saab tuvastada ka tugevalt sidusad komponendid suunatud graafis. Pärast algoritmi jooksutamist kontrollime tulemuseks saadud tabelit. Näiteks mingi tipuga i kuuluvad samasse sidususkomponenti need tipud `j`, kus `[i,j] = [j,i].`

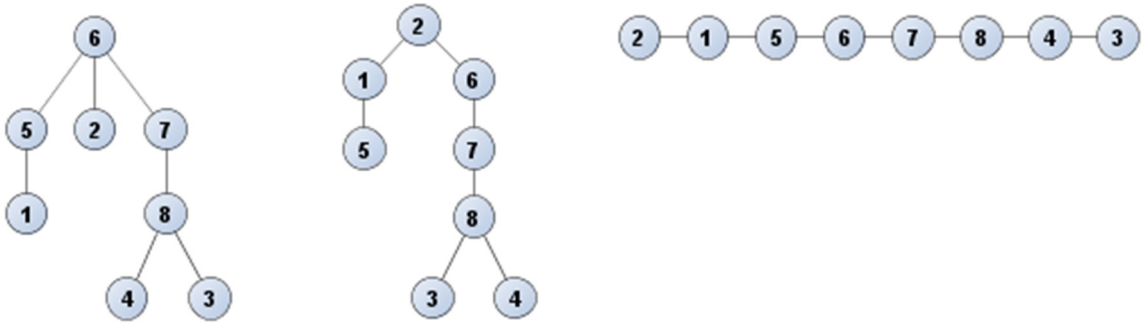
10.2 TOESE MÕISTE

Tuletame meelde, et puud saab vaadelda kui graafi erijuhtu. Graaf G on puu, kui G on sidus ja tsükliteta.

Sidusa graafi G **toeseks** ehk **aluspuuks** nimetatakse tema alamgraafi T, mis on puu ja mis sisaldab graafi G kõiki tippe. Ühel graafil võib olla mitu erinevat toesepuud. Näiteks graafi



toesepuudeks võivad olla järgmised graafid:



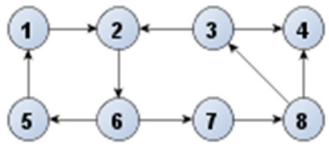
10.2.1 Servade liigitamine

Graafi sügavuti läbides tekib nn sügavuti otsimise puu, mis on graafi aluspuu. Milline see puu täpselt on, sõltub sellest, millisest tipust sügavuti läbimist alustatakse ning mis järjekorras naabreid valitakse. Sügavuti otsimisel saab suunatud graafis eristada nelja tüüpi servi:

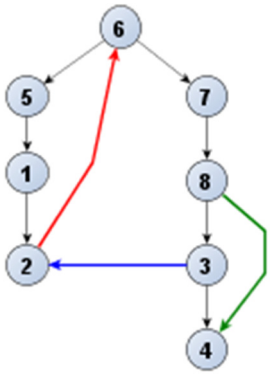
- Puu servad (*tree edges*) ehk servad, mis antud valiku juures moodustavad graafi toesepuu.
- Tagasiviivad servad (*back edges*) ehk servad, mis ühendavad tippu mõne ülema tipuga.
- Edasiviivad servad (*forward edges*) ehk servad, mis ühendavad tippu mõne alama tipuga, kuid ei ole puu servad.

- Ristuvad servad (*cross edges*) ehk servad, mis ei kuulu ühtegi eelnevasse liiki.

Alustades järgmise suunatud graafi läbimist tipust numbriga 6 ja töödeldes alati enne väiksema numbriga naabri



saame järgneva toesepuu koos lisaservadega:



Joonisel mustad servad on puu servad, punane serv on tagasiviiv serv, roheline edasiviiv serv ning sinine on ristuv serv.

Suunamata graafis räägime harilikult kahest esimesest tüübist, kuna muud tüüpi servad tegelikult langevad kokku.

10.2.2 DFS koos servade liigitamisega

Võtame tavalisel sügavuti läbimisel abiks veel ühe tipu oleku, eristades vaatlemata, leitud ja töödeldud tippe. Peame meeles ka tipu vanema toesepuus ehk tipu, mille kaudu antud tipp esmalt leiti. Nii saame lihtsa algoritmi, kuidas eristada loodavas toesepuus puu servad, tagasiviivad servad ja edasiviivad/ristuvad servad. Algoritm ise on järgmine:

1. Vaata graafi tippu u, märgi see leituks.
2. Vaatle tipu u naabrit v
   a. Kui seda naabrit pole veel leitud, pane u selle vanemaks ja jätka punktist 1 tipu v kohta. Serv (u, v) on puu serv.
   b. Kui naaber on varem leitud ja u ei ole v vanem, siis on serv (u, v) tagasiviiv serv.
   c. Kui naaber on juba uuritud, siis on serv (u, v) edasiviiv või ristuv serv.
3. Kui kõik u naabrid on töödeldud, märgi tipp u töödelduks.

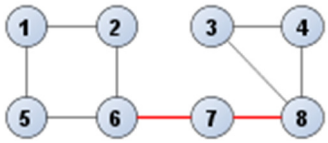
Funktsioon, mis liigitab servad:

```
void liigitaServ(int u) {
    tipud[u] = LEITUD;
    for (int j = 0; j < (int)graaf[u].size(); j++) {
        int v = graaf[u][j];
        if (tipud[v] == UUS) {
            vanem[v] = u;
            cout << "Puu serv " << u+1 << " " << v+1 << endl;
            liigitaServ(v);
        }
        else if (tipud[v] == LEITUD) {
            if (v != vanem[u]) {
                cout << "Tagasiviiv serv " << u+1 << " " << v+1 << endl;
            }
        }
        else if (tipud[v] == UURITUD) {
            cout << "Edasiviiv või ristuv serv " << u+1 << " " << v+1 << endl;
        }
    }
    tipud[u] = UURITUD;
}
```

10.3 Sillad ja Artikulatsioonipunktid

Sidusa graafi $G = (T, S)$ serva $s \in S$ nimetatakse **sillaks**, kui selle serva eemaldamisel graafis G muutub graaf mittesidusaks. Võib öelda ka, et sild on graafi serv, mille eemaldamisel kasvab graafi sidususkomponentide arv.

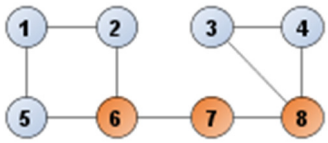
Näiteks graafis:



on sildadeks servad (6,7) ja (7,8) (joonisel märgitud punasega).

Sidusa graafi $G = (T, S)$ tippu $t \in T$ nimetatakse **artikulatsioonipunktiks**, kui selle tipu eemaldamisel graafist *koos sellesse suunduvate ja sellest väljuvate servadega* muutub graaf mittesidusaks. Võib öelda ka, et artikulatsioonipunkt on graafi tipp, mille eemaldamisel kasvab graafi sidususkomponentide arv.

Näiteks graafis



on tipud 6, 7 ja 8 (joonisel punased) artikulatsioonipunktid.

Sillad ja artikulatsioonipunktid on omavahel üsna tihedalt seotud. Silla otsatipud on artikulatsioonipunktid, välja arvatud juhul, kui sild on ainus otspunkti sisenev või sealt väljuv serv. Samas iga kahe artikulatsioonipunkti vahel asuv serv ei pruugi olla sild.

10.3.1 Ülesanne: Muinastulede öö

Juba viikingiajal oli kombeks meie esiisadel ohust märku anda süüdatud lõketega. Siis toimis rannikul laevateede läheduses juba ühtne valvesüsteem, mis ulatus ka sisemaale. Sellesse pidi iga kogukond andma lõkke süütamisel ja hoidmisel oma kindlaksmääratud panuse ja kandis küllalt ranget vastutust teate mitteedastamise korral. Süsteem oli ise lihtne – kui lõket nähti, tuli süüdata ise teine, et see paistaks järgmiste elupaikadeni jne. On arusaadav, et mõned lõkked on sõnumi levimiseks igasse maanurka olulisemad kui teised. Sinu ülesanne on leida, milliseid lõkkeid ei tohi kindlasti süütamata jätta. Ülesandes eeldatakse, et kui lõke kohas A paistab kohta B, siis lõke kohas B paistab kohta A.

Sisendi esimesel real on üks arv n – lõkkekohtade arv. Järgneval n real on igaühel 0 – n-1 arvu – n.dast lõkkest paistvate lõkete järjekorranumbrid. Väljastada nende lõkkekohtade järjekorranumbrid, mida ei tohi kindlasti jätta süütamata. Järjekorraumbrid väljastada kasvavas järjestuses.

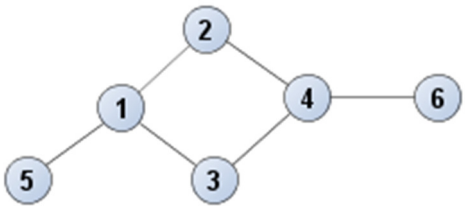
NÄIDE:

```
6
2 3 5
4
4
6
```

Vastus:

```
1 4
```

Selles ülesandes on lõkked graafi tippudeks ja kahe lõkke vahel on serv, kui ühest lõkkekohast paistab teine lõke. Kui graaf loodud, tuleb leida need lõkked, mis ei tohi kindlasti süütamata jääda ehk mille väljalülitamise korral ei ole moodustunud graaf enam sidus, st need lõkked, mis on artikulatsioonipunktideks.



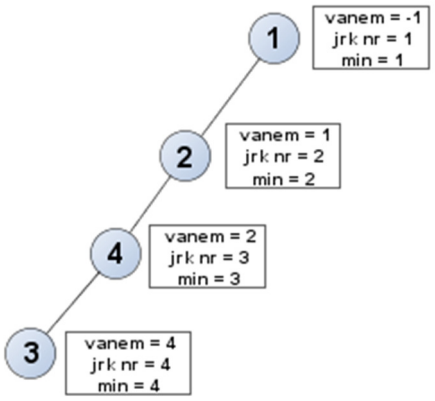
Ülesande näiteandmetele vastav graaf

Vahetu lähenemine oleks selline, et eemaldame graafist iga tipu ning kontrollime seejärel sidususkomponentide arvu. Kui sidususkomponentide arv suureneb, oleme leidnud artikulatsioonipunkti, kui mitte, siis antud punkt ei ole artikulatsioonipunkt. Selle keerukuseks oleks $O(V * (V + E))$.

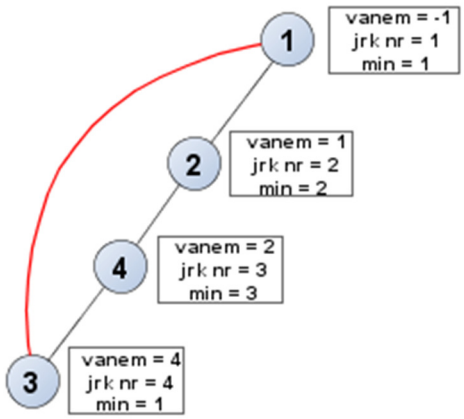
Saab ka paremini. Kasutades sügavuti läbimist, nimetame tipu u tipu v eellaseks, kui tippu v jõudmiseks läbisime tipu u. Tipp u on artikulatsioonipunkt kui

- u on juur toeses ja tal on vähemalt 2 alluvat.
- u ei ole juur ja tal on selline alluv v, millest algavas alampuus ei ole ühtki serva mõne u eellasega.

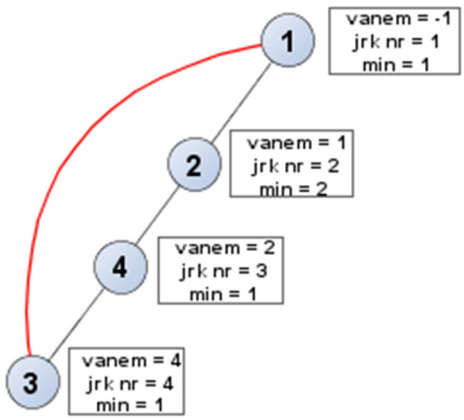
Nii on artikulatsioonipunkti leidmise algoritmi aluseks toesepuu. Hakkame seda looma, aga iga tipu juures peame meeles lisaks kahte numbrit – tipu avastamise järjekorranumbrit ehk leidmise aega ja üht teist numbrit, mida nimetame minimaalseks leidmise ajaks – see on vähima järjekorranumbriga tipu, millesse tema alampuust jõuda võib, number. Lisaks peame meeles, kes on tipu vanem. Alustades tipust 1 ja võttes tippe järjekorras, saame esialgu järgmise toesepuu alguse:



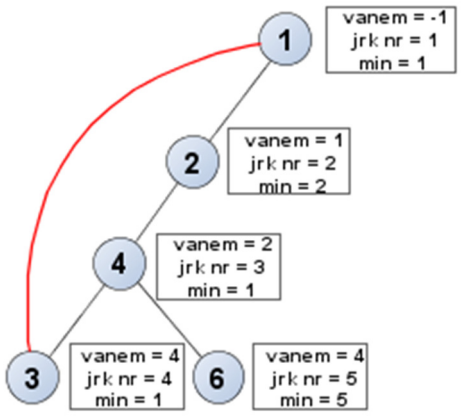
Nüüd tipust 3 läheb tagasiviiv serv tippu 1. Uuendame tipu 3 minimaalset leidmise aega milleks on kas tipu 3 senine minimaalne leidmise aeg või tipu 3 naabri 1 järjekorranumber – see, mis on neist kahest väiksem. Saame:



Kuna tipu 3 naabrid on nüüd kõik töödeldud, läheme tagasi tipu 4 juurde ja uuendame selle minimaalset aega, milleks saab tema alluva, tipu 3 minimaalne aeg:

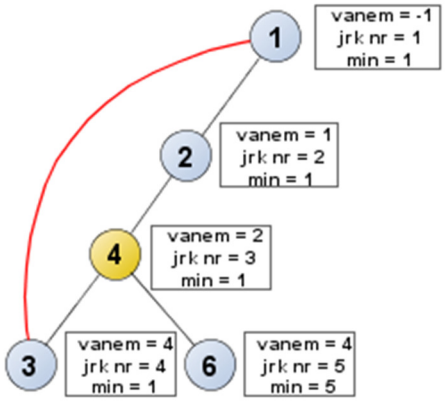


Hetkel tipp 4 ei kvalifitseeru artikulatsioonipunktiks, kuid sel tipul on veel naabreid, nimelt tipp 6:

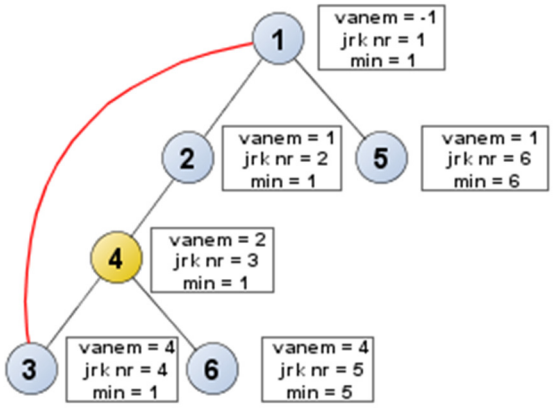


Kuna tipul 6 naabreid ei ole, siis sellega midagi teha ei ole vaja. Selline tipp ei saa olla ka artikulatsioonipunktiks. Küll aga on artikulatsioonipunktiks tipp 4, sest see on ainus tipp, mille kaudu saab tippu 6. Märgime tipu 4 artikulatsioonipunktiks (joonisel edaspidi kollane).

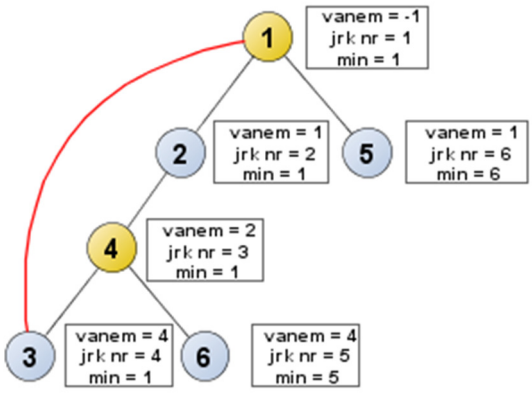
Tipu 4 naabrid on nüüd kõik üle vaadatud ja läheme tagasi tipu 2 juurde. Siin saame taas parandada madalaimat aega tipu 4 andmete põhjal ning otsustada, et 2 hetkel ei ole artikulatsioonipunktiks:



Ka tipul 2 ei ole rohkem alluvaid ja liigume tagasi tipu 1 juurde. Veel ei muutu seal midagi, jätkame tipu 1 naabriga, tipp 5-ga. Lisame selle puusse:



Kuna tipul 5 naabreid pole peale tema vanema, siis analoogselt tipuga 6 see mingit töötlemist ei vaja. Nüüd aga, kuna tipp 1 on puu juur ja tal on kaks alluvat, on ka tipp 1 artikulatsioonipuntiks.



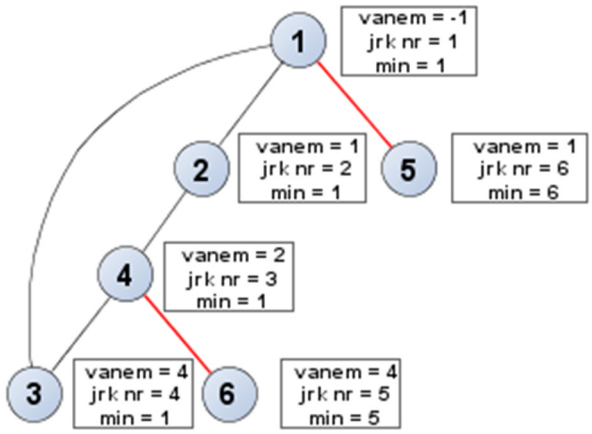
Selle algoritmi autoriteks on John Hopcroft Cornelli Ülikoolist ja Robert Tarjan Princetoni Ülikoolist ning see pärineb aastast 1973. See algoritm on lineaarse keerukusega - $O(V + E)$.

```
int n // tippude arv
int jrk_nr = 0; // mitmendana tipp leitakse
bool ap[n] = { 0 }; // leitud artikulatsioonipunktid
bool kylastatud[n] = {0}; // kas tipp on külastatud
int aeg[n] = {0}; // külastamise järjekorranumber ehk „aeg“
int madalaim[n] = {0}; // vähim sellise tipu külastusaeg, millel on antud tipuga serv.
int vanemad[n]; // hoiame iga tipu jaoks meeles tema vanema

void leiaAPd(int u)
{
    int alluvaid = 0; // loendame alluvaid alampuus
    kylastatud[u] = true; // siin me oleme
    aeg[u] = madalaim[u] = ++jrk_nr; // alguses on madalaim avastamise aeg

    vector<int>::iterator i;
    for (i = graaf[u].begin(); i != graaf[u].end(); ++i) {
        int v = *i;  // v on u naaber

        if (!kylastatud[v]) { //Kui v ei ole veel külastatud, siis on see u alluv puus
            alluvaid++; // loeme selle u alluvaks
            vanemad[v] = u; // ja paneme u v vanemaks
            leiaAPd(v); // jätkame v-ga

            madalaim[u] = min(madalaim[u], madalaim[v]);

            // u on artikulatsioonipunkt, kui:
            if (vanemad[u] == -1 && alluvaid > 1) // u on juur ja tal on mitu alluvat
                ap[u] = true;

            if (vanemad[u] != -1 && madalaim[v] >= aeg[u]) // u alt ei ole tagasiserva
                ap[u] = true;
        }

        else if (v != vanemad[u]) // kui v ei ole u vanem
            madalaim[u] = min(madalaim[u], aeg[v]);
    }
}
```

10.3.2 Sildade leidmine

Sildade leidmise algoritm on sarnane artikulatsioonipunktide leidmisele. Tipp u on artikulatsioonipunkt siis, kui `madalaim[v] >= aeg[u]`, serv (u, v) on sild siis, kui `madalaim[v] > aeg[u]`. Kuna otsime servi, siis mingid erandid juurtipu jaoks ei ole sildade leidmisel vajalikud.

Eelmises ülesandes saadud andmete põhjal on sildadeks servad (1, 5) ja (4, 6):

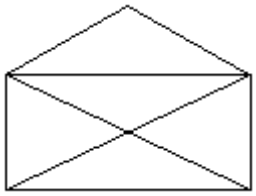


Siin on protseduur, mis leiab sillad:

```
void leiaSild(int u)
{
    kylastatud[u] = true; // siin me oleme
    aeg[u] = madalaim[u] = ++jrk_nr; // leidmise jrk

    // vaatame kõiki naabreid
    vector<int>::iterator i;
    for (i = graaf[u].begin(); i != graaf[u].end(); ++i) {
        int v = *i; // v  on u naaber
        if (!kylastatud[v]) { //kui ei ole enne leitud
            vanemad[v] = u; // paneme u vanemaks
            leiaSild(v); // ja töötleme graafi sealt edasi

            madalaim[u] = min(madalaim[u], madalaim[v]);

            //Kas v-l on ühendus mõne u eellasega? Kui mitte, siis on u-v sild
            if (madalaim[v] > aeg[u])
                cout << u << " " << v << endl;
        }

        // kui oleme enne külastanud, muudame madalamat aega
        else if (v != vanemad[u])
            madalaim[u] = min(madalaim[u], aeg[v]);
    }
}
```

10.4 EULERI GRAAF

Kunagi oli populaarne ajaviitemäng, kus tuli joonistada paberile lahtine kirjaümbrik ilma pliiatsit paberilt tõstmata ja ühtki joont topelt tõmbamata. Selleks on rohkem kui üks võimalus, aga neid kõiki ühendab see, et alustada tuleb ühest alumistest punktidest ning lõpetada teises alumises punktis. Graafiteoorias nimetatakse graafis sellist teed ehk ahelat, mis läbib iga selle graafi serva täpselt üks kord, **Euleri ahelaks.**



Euleri tsükliks nimetatakse sellist Euleri ahelat, mis algab ja lõppeb samas tipus. Sellist graafi, milles leidub Euleri tsükkel, nimetatakse **Euleri graafiks**, graafi, milles leidub Euleri ahel aga **Euleri semigraafiks.**

Selle kindlaks tegemine, kas graaf on Euleri graaf, on väga lihtne. Graaf peab olema sidus ja selles peavad olema kõik tipud sellised, et sinna viivate teede arv on paarisarv. See tingimus on nii tarvilik (kehtib igal Euleri graafil) kui ka piisav (kehtib ainult Euleri graafidel). Euleri semigraafis tohib olla täpselt kaks tippu, mille korral see ei kehti. Sel juhul algab Euleri ahel ühes nendest tippudest ja lõppeb teises, nagu ka kirjaümbriku näites. Kontrolli, kas graaf on Euleri graaf või semigraaf, saab teha üsna lihtsalt ajaga $O(V + E)$ ning seda tehakse sageli juba graafi sisselugemisel.

10.4.1 Euleri tsükli leidmine

Keerulisem on aga leida tegelik tee või tsükkel. Selleks on kaks tuntud algoritmi: Fleury algoritm aastast 1883 ja Hierholzeri algoritm aastast 1873.

Fleury põhiideeks on „ära põleta sildu!“:

1. Kui graafis on 2 paaritut tippu, alusta ühest neist. Kui selliseid tippe pole, alusta, kust tahad.
2. Vaata tipust väljuvaid servi: kui nende seas on selline, mis ei ole sild, vali see esimesena.

Fleury algoritm on küll väga elegantne, kuid kuna selle jaoks on vaja leida graafist sillad, on selle keerukuseks $O(S^2)$, kus S on servade arv graafis. Samuti ei saa Fleury algoritmi kasutada suunatud graafidel.

Hierholzeri idee põhineb tsüklite eemaldamisel – kui eemaldada Euleri graafist kõik mingi tsükli servad, siis allesjäänud alamgraafid on samuti Euleri graafid. Algoritm ise on järgmine:

1. Alusta suvalisest tipust u.
2. Mine suvaliselt mööda servasid, kuni oled tagasi tipus u (kui graafis on Euleri tsükkel, pole kinnijäämine võimalik). Tulemuseks on tsükkel.
3. Niikaua, kui meie leitud tsüklis on mõni tipp v, millest väljub mõni serv, mis pole leitud tsükli osa:
    a. Koosta uus tsükkel alustades tipust v.
    b. Lisa uus tsükkel esialgsele.

Kui hoida meeles, kus on ülejäävaid servi, töötleme iga serva vaid ühe korra ja sellisel juhul on kogu algoritmi keerukus $O(E)$.

Hierholzeri algoritmi saab kasutada ka suunatud graafidel.

Vaatame järgmist ülesannet:

10.4.2 Loomanimed

Taas üks tuntud laste ajaviitemäng on selline, et öeldakse järjestikku kordamööda loomanimesid, aga nii, et iga järgmise looma nimi peab algama selle tähega, millega eelmine lõppes. Näiteks ütleb esimene mängija „rebane“, teine peab ütlema e-tähega algava looma, nt „eesel“, kolmas selle peale l-iga algava looma, nt „lõvi“ ning seejärel esimene i-ga algava, nt „ilves“ jne.
Sulle on antud hulk tähestiku järjekorras loomanimesid ja sa pead leidma, kas neist saab moodustuda terve mäng ja kui saab, siis leidma vähemalt ühe sobiva järjestuse.
Sisendi esimesel real on loomade arv `n` ja järgneval `n` real igaühel üks loomanimi. Loomanimed koosnevad kõik väikestest ladina tähtedest.
Väljastada loomanimed selles järjekorras, mis moodustaks mängu, iga nimi eraldi real. Kui kõikidest nimedest mängu moodustamine pole võimalik, väljastada ühele reale „`ei saa`“.

NÄIDE 1:

```
8
eesel
lehm
lammas
mutt
rebane
siil
tigu
tiiger
```

Vastus:

```
tiiger
rebane
eesel
lammas
siil
lehm
mutt
tigu
```

NÄIDE 2:

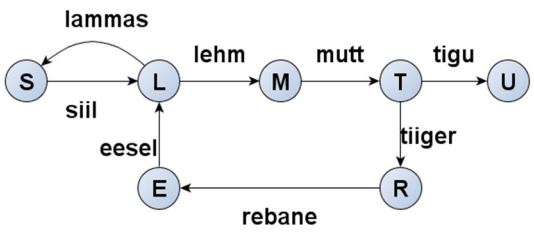
```
4
karu
koer
rott
makaak
```

Vastus:

```
ei saa
```

Pildil skulptuur Breemeni linna moosekantidest.
Kui pole veel lugenud, siis loe nende lugu!

Selle ülesande lahendamiseks tuleb kõigepealt analüüsida, milline graaf tekib. Üks võimalus ongi esitada graaf kujul, kus tippudeks on kõikvõimalikud erinevad algus ja lõpptähed ning loomanimed on suunatud servad nende tippude vahel:

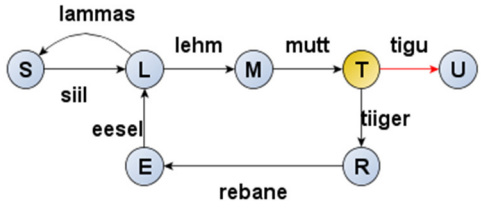


Lahendiks on Euleri tee selles graafis, kui selline leidub. Seega teiseks ülesandeks on teha kindlaks, kas see tee leidub ning ühes sellega, milline on tee alguspunkt. Seda teeb järgnev funktsioon:

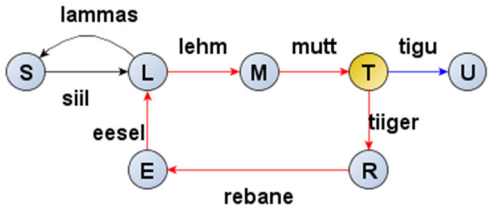
```
bool onTee()
{
    int alguseid = 0;
    int loppe = 0;
    for (int i = 0; i < 26; i++) {
        int valja = graaf[i].size();
        if (sisse[i] > valja + 1 || sisse[i] + 1 < valja)
            return false; //sisse ja väljatulevate vahe on rohkem kui 1
        if (sisse[i] == valja + 1) {
            loppe++; // võimalik lõpp
        }
        if (sisse[i] + 1 == valja) {
            alguseid++; // võimalik algus
            algus = i;
        }
        if (valja == 0 && algus == i)
            algus++; // tsükli jaoks esimese tipp, kust on servi;
    }
    if (alguseid < 2 && alguseid == loppe) { // üks lõpp ja üks algus või ei kumbagi
        return true;
    }
    return false;
}
```

Kui on kindlaks tehtud, et Euleri tee eksisteerib ning ühes sellega selle algustipp (tsükli korral sobib milline tahes, alustame tähestikus esimesest tähest, kust servi väljub). Antud näites väljub tipust T 2 serva, siseneb aga 1, seega peab tipp T olema Euleri tee alguseks. Tee lõpuks on tipp U, millest väljuvaid servi pole.

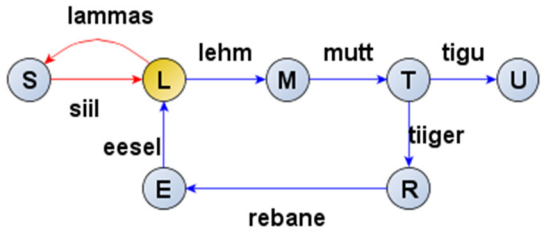
Alustame tipust T ja võtame suvalise serva, näiteks „tigu“:



Eemaldades selle serva, proovime edasi liikuda tipust U. Kuna aga sealt väljuvaid servi (rohkem) pole, siis oleme jõudnud tee lõppu. Praegu on meil kogu teeks „tigu“, aga graafis on veel servi. Läheme tagasi algusserva T juurde ja valime sealt teise väljuva serva „tiiger“. Sellest omakorda väljub „rebane“ jne:



Nendest kahest käigust saame kokku tee `tiiger-rebane-eesel-lehm-mutt-tigu`. T-st rohkem väljuvaid servi ei ole, M-ist ka mitte, aga L-ist on. Alustame järgmist teejuppi sealt:



Saame lisada olemasolevale teele veel kaks looma ja tee on nüüd `tiiger-rebane-eesel-lammas-siil-lehm-mutt-tigu`. Nüüd on ka kasutatud kõik loomanimed (servad) ja vastus on käes.

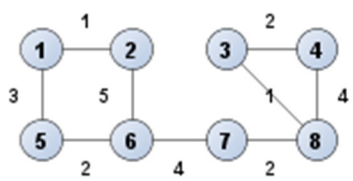
Funktsioon, mis leiab Euleri tee ja väljastab selle:

```
void leiaTee()
{
    tipud.push(algus);
    int u = algus; // vaadeldav tipp

    while (!tipud.empty()) {
        if (!graaf[u].empty()) { // on veel väljuvaid servi
            tipud.push(u);
            int v = graaf[u].top();
            graaf[u].pop();
            u = v;
        }
        else { // jõudsime tippu, kus pole väljuvaid servi
            tee.push_back(u); // u on viimane tipp, mis pole veel tees
            u = tipud.top(); // võtame järgmise (eelmise) tipu
            tipud.pop();
        }
    }

    // tee väljastamine
    for (int i = tee.size() - 1; i > 0; i--) {
        cout << servad[tee[i]][tee[i - 1]].top() << endl;
        servad[tee[i]][tee[i - 1]].pop();
    }
}
```

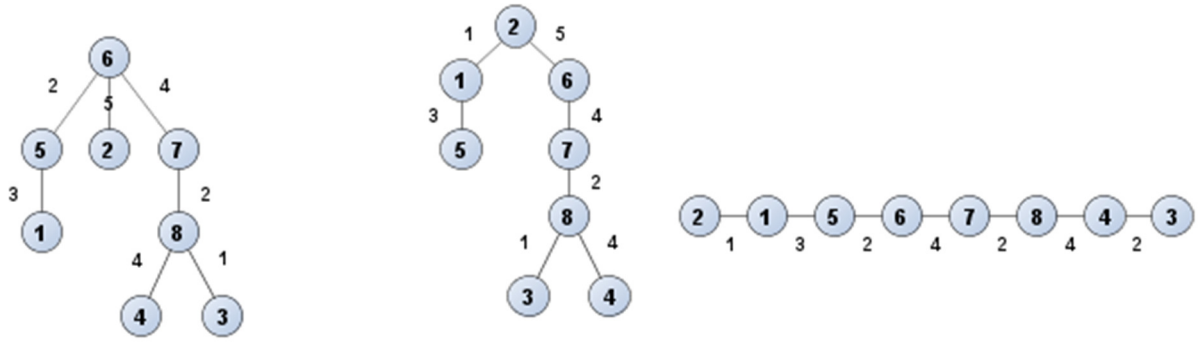
10.5 Minimaalne Toes

Minimaalsest toesest saab rääkida siis, kui tegemist on kaalutud graafiga. Nimetame toesepuu kaaluks kõigi tema servade kaalude summat. Erinevate toesepuude kaalud võivad olla erinevad.

Näiteks graafis

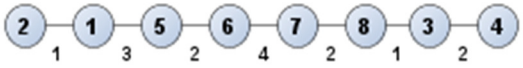


on eelpool toodud toesepuude kaalud



esimesel puul 21, teisel 20 ja kolmandal 18.

Probleemide lahendamisel tuleb ette, et on vaja leida just graafi kõige väiksema ehk minimaalse kaaluga toes. Antud näitepuudest pole ükski minimaalse kaaluga, selleks on hoopis puu kogukaaluga 15:



10.5.1 Minimaalse toese leidmine

Minimaalse toese leidmiseks on kaks tuntud algoritmi. Esimene neist on Kruskali algoritm, mille idee on järgmine:

1. Alusta tühjast puust.
2. Kuni lisatud on vähem kui $n - 1$ serva, lisa lühim serv nende hulgast, mis ei tekita eelnevatega koos tsüklit.
3. Abiks on, kui sorteerime alguses kõik servad kaalude järgi
4. Meil tekib hulk sidususkomponente – nende ühendamiseks saab kasutada ühisosata hulki (*disjoint sets, union-find*)

Primi algoritmi töötab aga järgmiselt:

1. Vali suvaline tipp toesepuu alguseks.
2. Kuni leidub veel tippe, mis puusse pole lisatud, leia lühim serv, mis seob mõnda puu tippu seal mitteolevaga,
3. Lisa see serv puusse.

10.5.2 Ülesanne: Elektrikatkestus

Meie linnakeses oli suur torm ja seetõttu langes rivist välja hulk elektriliine ning praktiliselt kogu linn on elektrita. Kuna varasem elektrivõrk oli üles ehitatud eri aegadel tõmmatud liinidest, siis ei pruukinud iga tarbija olla ühendatud võrguga just kõige lühemat teed pidi ning kokku sarnanes kogu süsteem nagunii rohkem taldrikutäie spagettidega. Igatahes tuleb nüüd pea kogu elektrisüsteem üles ehitada nii kiiresti ja odavalt kui võimalik. Sulle on antud tarbijate nimekiri ja nendevahelised kaugused, kus ühendus tõmmata võimalik on. Väljasta uute liinide minimaalne kogupikkus, mis varustab kõik tarbijad elektriga.

Sisendi esimesel real on kaks täisarvu: tarbijate arv m ja ühenduste arv n. Järgneval n real on igaühel 3 täisarvu x, y ja k, kus x ja y on tarbijate numbrid, kelle vahele ühenduse tekitada saaks ja k on nende tarbijate vaheline kaugus.

Väljastada üks täisarv minimaalne liinide kogupikkus, millega saab kõik tarbijad omavahel ühendada.

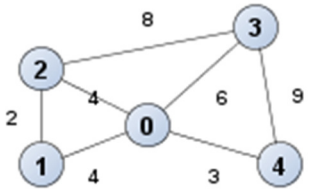
NÄIDE:

```
5 7
1 0 4
1 2 2
0 2 4
0 4 3
0 3 6
2 3 8
3 4 9
```

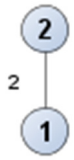
Vastus:

```
15
```

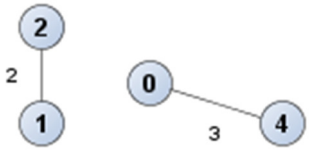
Esiteks, kuna tegu on graafiülesandega, tuleb luua graaf. Visuaalselt näeb näiteandmetel põhinev graaf välja selline:



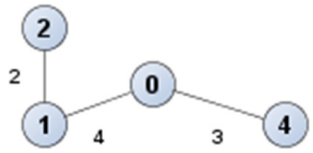
See on küllaltki lihtne ja läbipaistev minimaalse toese ülesanne. Kasutame ülesande lahendamiseks Kruskali algoritmi. Hakkame looma graafi aluspuud ja kõigepealt valime sinna kõige kasulikuma ehk kõige väiksema kaaluga serva, milleks on (1,2):



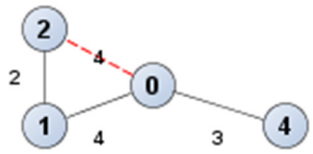
Järgmiseks vaatame kaalult järgmist serva (0,4). See ei tekita eelmise servaga tsüklit, lisame selle:



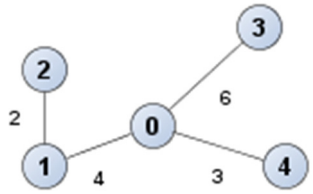
Nüüd on meil valida kahe serva vahel, mis on võrdse kaaluga: (0,1) ja (0,2). Proovime lisada neist esimese, mis ka õnnestub, kuna tsüklit ei teki:



Siin tasub tähele panna, et uus serv ühendas kaks enne iseseisvat komponenti. Seejärel proovime lisada serva (0,2). Seda aga toesesse lisada ei saa, kuna tipud 0 ja 2 on juba samas komponendis ja nii tekiks tsükkel:

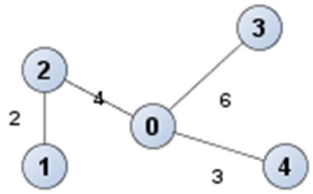


Seega serv (0,2) ei kuulu minimaalsesse toesesse, jätame selle kõrvale ja jätkame servaga (0,3). Selle saab taas lisada olemasolevale komponendile:



Kindlasti ei tasu ka ära unustada, et toesepuus, nagu igas puus, on täpselt $n-1$ serva, kus n on tippude arv. Kuna graafis on 5 tippu ja oleme lisanud juba 4 serva, siis ülejäänud servi enam vaatama ei pea – minimaalne toesepuu on leitud.

Kui oleksime valinud serva (0,2) enne serva (0,1), saanuksime järgmise minimaalse toesepuu:



Seega ka minimaalseid toesepuid võib olla mitu.

Konkreetse ülesande teeb lihtsamaks asjaolu, et tegelikku toest leida pole vajagi – piisab selle kaalust. Seega toesesse kuuluvate servade meelespidamise asemel piisab, kui toesesse kuuluva serva leidmisel lisame selle serva kaalu summale. Kuna tegelikke minimaalseid toesepuid võib olla mitu, siis tegelikult selline kogukaalu leidmine on küllaltki tavaline.

Siin on funktsioon, mis leiab minimaalse liinide pikkuse, kasutades Kruskali algoritmi:

```
int leiaPikkus()
{
    int servi_puus = 0;
    int jrgm_serv = 0;
    int vastus = 0;

    sort(servad.begin(), servad.end()); // sorteeritud kauguse (kaalu) järgi

    UF hulgad = UF(m); // iga tipp on eraldi hulk
    while (servi_puus < m - 1) // puus on m-1 serva
    {
        int kaugus = servad[jrgm_serv].first;
        pair<int, int> otsad = servad[jrgm_serv].second;

        //kui ei teki tsüklit, lisame serva puusse
        if (!hulgad.connected(otsad.first, otsad.second)) {
            vastus += kaugus;
            servi_puus++;
            hulgad.merge(otsad.first, otsad.second);
        }
        jrgm_serv++;
    }

    return vastus;
}
```

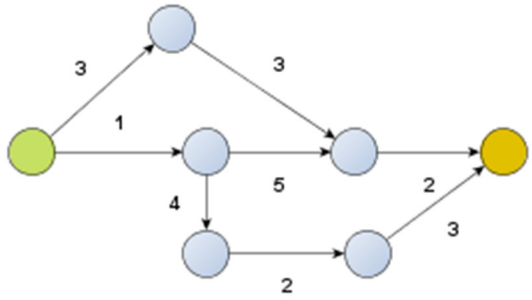
10.6 MAKSIMAALNE VOOG JA MINIMAALNE LÕIGE

Probleem sellest, kui suur on mingi võrgustiku (milleks võib olla veevärk, teedevõrk, kommunikatsioonistruktuur vms) summaarne läbilaskevõime, oli aktuaalne juba enne arvutite laialdast kasutuselevõttu. Probleemiga seonduvad kaks peamist mõistet on maksimaalne voog ja minimaalne lõige. 1956. aastal panid Ameerika matemaatikud Lester Ford ja Delbert Fulkerson tähele, et need mõisted on omavahel tihedalt seotud ning leiutasid sellele tähelepanekule toetudes Ford-Fulkersoni algoritmi. Enne algoritmi kirjelduseni jõudmist tutvume veel mõningate graafiteooria mõistete formaalsemate definitsioonidega.

10.6.1 Võrk

Võrguks nimetatakse suunatud sidusat graafi, mille igale kaarele on vastavusse seatud mittenegatiivne reaalarv ehk kaare **läbilaskevõime (***i.k. capacity*).

Võrku, millel on täpselt üks tipp, kust kaared ainult väljuvad ja täpselt üks tipp, kuhu kaared ainult sisenevad, nimetatakse **transpordivõrguks** (i.k. *flow network*, *transportation network*). Tippu, kust kaared väljuvad, nimetatakse **lähteks** (i.k. *source*) ja tippu, kuhu kaared suunduvad, **suudmeks** (i.k. *sink*). Kuna edaspidi käsitleme peamiselt transpordivõrke, siis edaspidi mõeldakse sõna „võrk“ tähenduses justnimelt transpordivõrku.

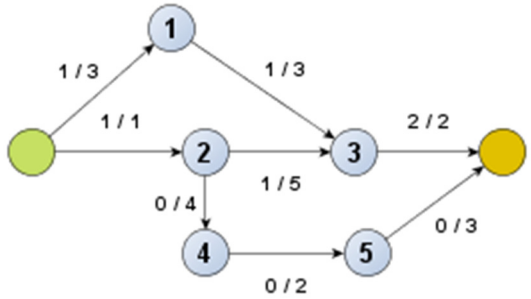


Transpordivõrk. Roheline tipp on lähe ja kollane tipp suue. Kaartel on läbilaskevõimed.

10.6.2 Voog

Transpordivõrgu kaartele seatakse sageli vastavusse veel üks arv: kaare **voog** (i.k *flow*). Läbilaskevõimest ja voost arusaamiseks on hea ette kujutada näiteks teedevõrgustikku. Teedel on läbilaskevõime ehk mitu autot ajaühikus seda maksimaalselt läbida saab. Samas on arusaadav, et alati nii palju autosid seal ei sõida. See, kui palju autosid tegelikult ajaühikus teelõiku läbib, ongi voog. Voog ei saa olla suurem kui läbilaskevõime.

Vool on veel selline omadus, et tippu sisenevate kaarte voogude summa on võrdne sellest tipust väljuvate voogude summaga. Erandiks on lähe ja suue. Samas lähtest väljuvate kaarte voogude summa on võrdne suudmesse jõudvate voogude summaga. Ideaalses maailmas ei lähe midagi teel kaduma, kõik, mis lähtest väljub, jõuab suudmesse.

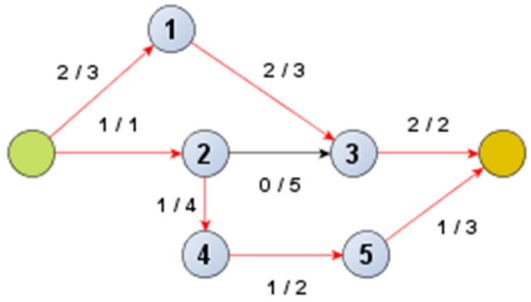


Voogudega võrk. Esimene arv kaare juures on voog ja teine läbilaskevõime.

Võime ette kujutada, et joonisel alustab teed lähtest kaks sõidukit: üks liigub lähtest suudmesse läbi tippude 1 ja 3 ning teine läbi tippude 2 ja 3. Kaart tipus 3 suudmesse läbivad mõlemad sõidukid.

10.6.3 Maksimaalne voog

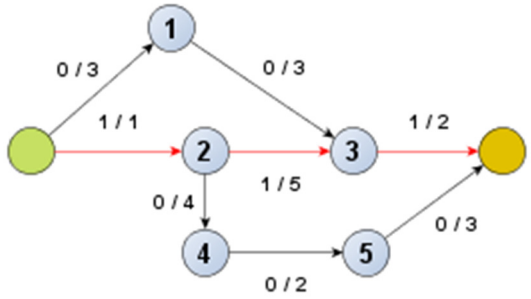
Võrkude puhul on enamasti kõige huvitavamaks küsimuseks, kui palju maksimaalselt saab midagi võrku läbides jõuda lähtest suudmesse ehk leida võrgu **maksimaalne voog.**



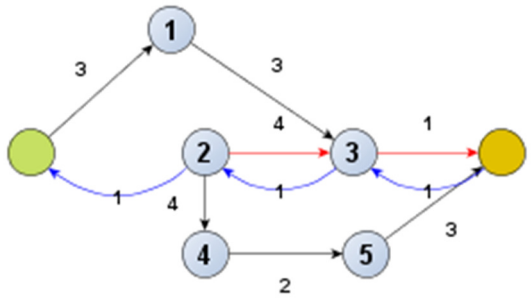
Võrgu maksimaalne voog. Maksimaalse voo väärtus on 3 – nii palju jõuab lähtest suudmesse.

10.6.4 Maksimaalse voo leidmine

Vaatame, kuidas leida maksimaalset voogu. Selleks kõigepealt leiame suvalise tee lähtest suudmesse, näiteks läbi tippude 2 ja 3. Mööda seda teed saab liikuda voog suurusega 1, kuna sel teel on serv, mille läbilaskevõime on 1



Kaarel lähtest tippu 2 on voog võrdne läbilaskevõimega ehk see kaar on küllastunud ja sel voogu enam suurendada ei saa, kuid paneme tähele, et kaartel (2,3) ja (3,S) on vood küllastamata ja neid kaari saaks kasutada võrgu voo suurendamiseks. Järgmise sammuna loome **jääkvõrgu** – samade tippude ja sarnaste servadega graafi, kus iga kaare läbilaskevõimeks on vooga graafi vastava kaare läbilaskevõime ja voo vahe. Seda vahet nimetatakse jääk-läbilaskevõimeks (i.k. *residual capacity*), lühidalt lihtsalt jäägiks. Kaared, mis on esialges graafis küllastunud (mille jääk on 0), jätame lisamata. Lisaks lisame tagasikaared nende tippude vahele, kus voog on positiivne. Nende läbilaskevõimeks on voo väärtus algses graafis:

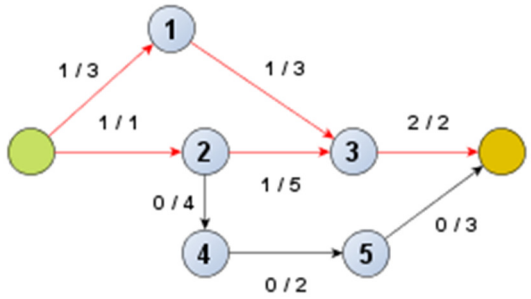


Eelmisele joonisele vastav jääkvõrk. Sinisega on tähistatud tagasikaared.

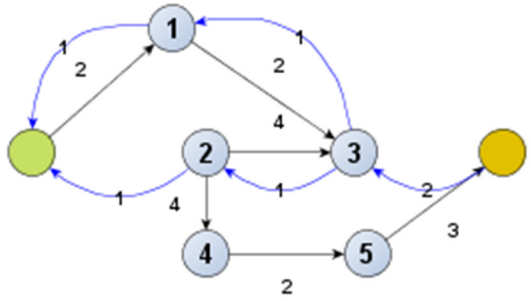
Kui vooga võrgu jääkvõrgus leidub selline tee, mis algab lähtest ja lõpeb suudmes ning mida mööda saab veel ressursse liigutada ehk iga kaare läbilaskevõime on positiivne, saame algse graafi voogu suurendada selle tee läbilaskevõime võrra. Sellist teed jääkvõrgus nimetatakse **täiendavaks teeks** (i.k. *augmenting path*).

Võrgul on leitud maksimaalne voog siis ja ainult siis, kui selle vooga võrgu jääkvõrgus ei leidu täiendavat teed.

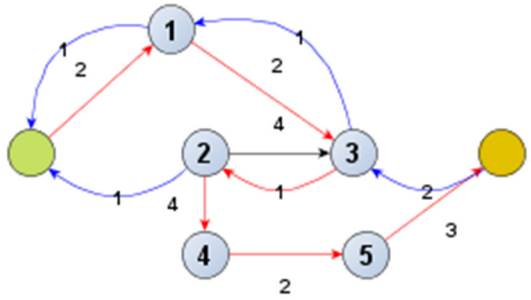
Ülaltoodud jääkvõrgus selline tee leidub, näiteks L-1-3-S. Kuna serva (3,S) läbilaskevõime on vaid 1, saame nii suurendada algset voogu ühe võrra. Selleks liidame algse graafi vastavate servade voogudele ühe juurde:



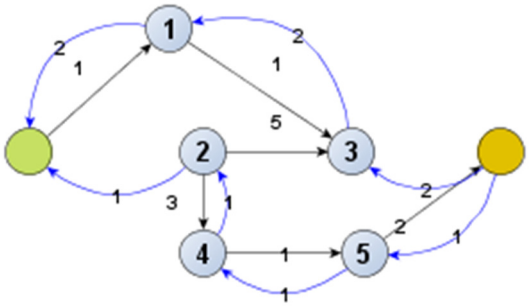
ja leiame tekkinud võrgu jääkvõrgu:



Nüüd tuleb jääkvõrgu tegelik olemus ja vajadus paremini ilmsiks, sest kui eelmisel sammul oli tee lähtest suudmesse olemas ka algsel võrgul, siis praegu enam ei leidu selles ühtki teed, mida mööda saaks voogu suurendada. Küll aga leidud tee algusest lõppu jääkvõrgus:



See tee kasutab tagasikaart (3,2). Kuna taas on minimaalse läbilaskevõimega serv selles tees väärtusega 1, saame suurendada voogu ühe võrra. Kanname selle tee algsele võrgule, pidades silmas, et tagasiviiva tee puhul tuleb voog esialgse graafi vastava serva voost liitmise asemel lahutada. Tulemuseks on eelpool toodud maksimaalse vooga võrk. Selle jääkvõrguks on



ning pole keeruline veenduda, et selles ei leidu enam ühtki teed lähtest suudmesse.

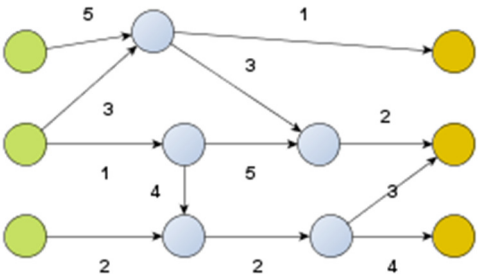
Kirjeldatud juhis ongi Ford-Fulkersoni algoritm. Mõnikord nimetatakse seda ka Ford-Fulkersoni meetodiks, kuna puudub täpne eeskiri, kuidas leida täiendavat teed jääkgraafis, viimasest aga võib sõltuda algoritmi keerukus.

Täisarvuliste läbilaskevõimete korral on selle keerukuseks $O(Ef)$, kus f on maksimaalse voo väärtus. Reaalarvuliste läbilaskevõimete korral ei pruugi antud algoritm tööd lõpetada.

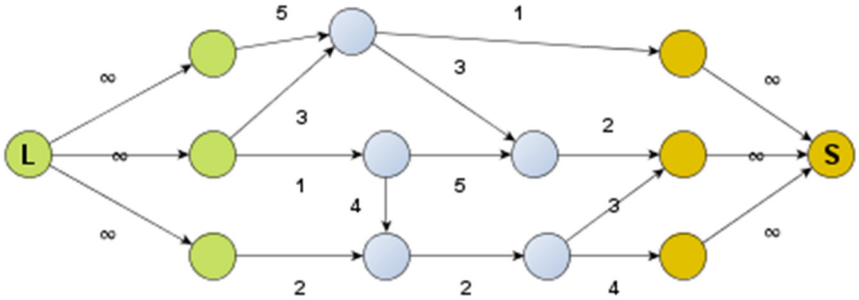
Kõige tuntum Ford-Fulkersoni meetodi implementatsioon on **Edmonds-Karpi algoritm**. See kasutab täiendava tee leidmiseks laiuti läbimist ning on keerukusega $O(VE^2)$.

10.6.5 Mitme lähte ja suudmega ülesanded

Mõnikord võib olla, et lähteülesandes on mitu lähet ja/või mitu suuet, näiteks võivad rongid väljuda mitmest depoost või kaup liikuda välja mitmest tehasest.



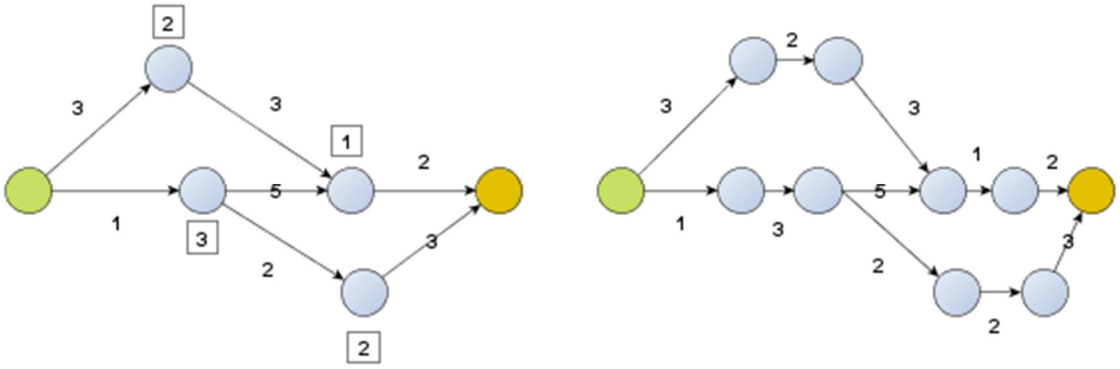
Sellisel juhul täiendatakse võrku kahe tipuga, nn superlähte ja –suudmega ∞:



10.6.6 Kaaludega tipud võrgus

Teiseks sagedaseks olukorraks on, kus ka tippudel on kaalud ehk läbilaskevõimed. Näiteks kauba vedamisel ladude vahel võib olla igal laol laadimisvõimsus, mis piirab kauba liikumist sellest laost

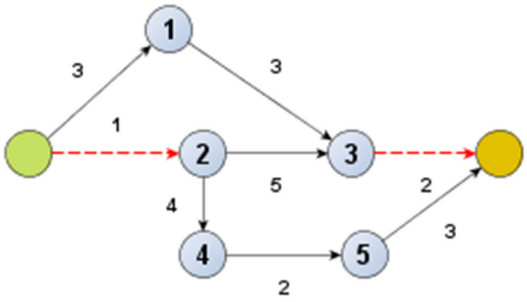
edasi. See lahendatakse nii, et tipp dubleeritakse ja sinna vahele lisatakse ekstra kaal tipu läbilaskevõimega:



10.6.7 Lõige

Vooga seotud on veel üks mõiste, nimelt **lõige** (i.k *cut*). Võrgu $G(T,S)$ lõikeks nimetatakse graafi G kaarte sellist hulka $S_L \subseteq S$, millesse kuuluvate kaarte eemaldamisel ei sisaldaks G enam ühtegi (suunatud) ahelat lähtest suudmesse. Lõike G_L läbilaskevõimeks nimetatakse tema kaarte läbilaskevõimete summat. Minimaalse võimaliku läbilaskevõimega lõikeid nimetatakse **minimaalseteks lõigeteks.**

Näiteks järgmise graafi



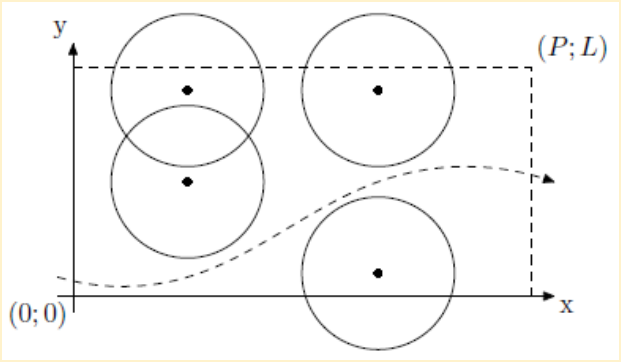
Minimaalseks lõikeks on kaared (L,2) ja (3,S) ning lõike väärtuseks 1+2=3.

Ford ja Fulkerson panid tähele, et võrgu maksimaalsete voogude väärtused on võrdsed selle võrgu minimaalsete lõigete läbilaskevõimetega. Seega saab maksimaalse voo leidmise algoritmi kasutada ka minimaalse lõike leidmiseks.

Seda kasutame ka järgmise ülesande lahendamisel. See ülesanne oli 2017. aastal Eesti Lahtisel Programmeerimisvõistlusel.

10.6.8 Vangid

Sõjavangide grupp plaanib põgenemist. Ainus tee laagrist välja viib läbi P meetri pikkuse ja L meetri laiuse kanjoni. Kanjonis on N valvurit, kes seisavad igaüks oma postil ja kelle nägemisraadius on täpselt 100 meetrit. Vahelejäämise vältimiseks tuleks läbi kanjoni hiilida nii, et kaugus lähima valvurini on alati rangelt suurem kui 100 meetrit, nagu näha alloleval joonisel.



Vangid, kellel on vägivallast juba kõrini, tahavad põgenemistee vabastamiseks kõrvaldada minimaalse võimaliku arvu valvureid. Kirjutada programm, mis neile selle arvu leiab. Võib eeldada, et vangid on võimelised kõrvaldama ükskõik milliseid valvureid (isegi neid, keda mõni teine valvur näeb).

Sisendi esimesel real on kolm täisarvu P (1≤P≤50 000), L (1≤L≤50 000) ja N (1≤N≤250). Faili järgmisel N real on igaühel ühe valvuri täisarvulised koordinaadid X_i ja Y_i (0≤X_i≤P, 0≤Y_i≤L). Kanjoni edelanurga koordinaadid on (0; 0) ja kirdenurga koordinaadid (P; L). Vangid võivad kanjonisse siseneda mistahes punktis (0; Y_s), kus 0≤Y_s≤L, ja väljuda mistahes punktis (P; Y_v), kus 0≤Y_v≤L. Seejuures ei pea Y_s ja Y_v olema täisarvud.

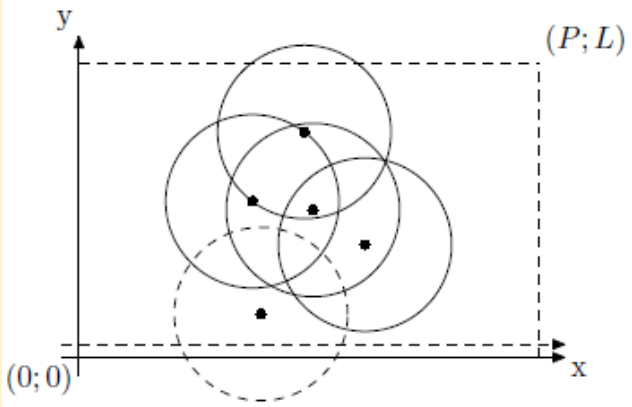
Väljundi ainsale reale väljastada mittenegatiivne täisarv, mis näitab vähimat võimalikku kõrvaldatavate valvurite arvu.

NÄIDE:

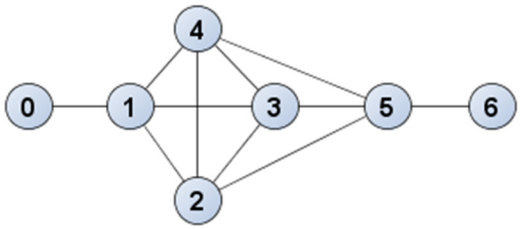
```
530 340 5
210 50
330 130
270 170
200 180
260 260
```

Vastus:

```
1
```

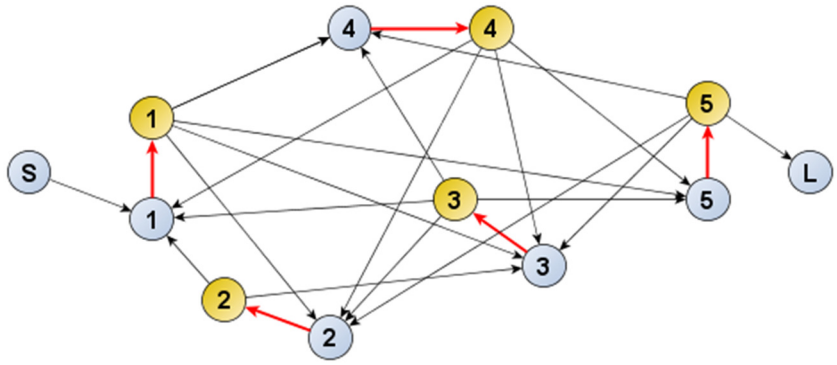


Esimeseks ülesandeks on jällegi koostada graaf. Võtame kõigepealt graafi tippudeks valvurid ja lisaks kaks tippu - kanjoni servad. Serv on kahe tipu vahel siis, kui nende valvealad puutuvad või kattuvad.



On küllaltki ilmne, et kui leidub tee tipust 0 tippu 6 ehk kanjoni seinte vahel, on põgenemine võimatu. Tuleb leida vähim arv tippe, mis tuleb graafist eemaldada, nii et tipp 0 ja tipp 6 ei ole enam ühendatud. Aga minimaalse lõike leidmiseks on tippude asemel vaja servi!

Selleks jagame iga valvurit tähistava tipu i kaheks eraldi tipuks i_s ja i_v ja lisame suunatud serva (i_s, i_v). Iga algse graafi suunamata serva (i, j) jaoks tekitame suunatud servad (j_v, i_s) ja (i_v, i_s). Loeme ühe kanjoni seina lähteks ja teise suudmeks. Kui tipp i puudutab lähteseina, lisame serva (L, i_s), kui suudmeseina, siis analoogselt serva (i_v, S), kus L ja S on vastavalt lähe ja suue. Saame:



Ülesande näiteandmetele vastav võrk.

Kuna küsitakse, mitu valvurit on vaja minimaalselt eemaldada, on iga valvuri eemaldamise hind sama ehk üks. Seega graafil punasega märgitud servadel on läbilaskevõimeks üks, teiste servade läbilaskevõime on piiramata – neid servi me nagunii eemaldada ei soovi. Samas võib arvestada ka iga serva läbilaskevõimeks ühe, sest graafi ülesehitus on selline, et rohkem nagunii ühestki tipupaarist edasi ei pääse.

Edasi leiamse sellel graafil maksimaalse voo, mis teoreemile vastavalt on ka minimaalseks lõikeks selles graafis ehk minimaalseks valvurite arvuks, keda on vaja vangidel kõrvaldada.

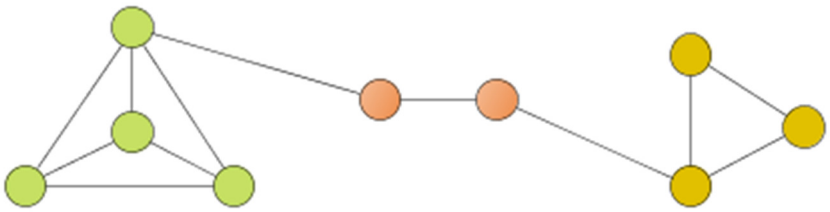
Siin on kood, mis ülesande lahendab:

```
#include <iostream>
#include <vector>
using namespace std;

int p, l, n;
int** kaarte_maht; // jääk-läbilaskevõimed
bool* vaadatud; // kas tippu on juba vaadatud
vector<int>* naabrid;  // tippude naabrid
int* naabrite_arv; // hoiame meeles iga tipu jaoks naabrite arvu

bool dfs(int tipp) // sügavuti läbimine
{
    if (tipp == n - 1) { // jõudsime seinani e suudmesse
        return 1; // maksimaalne voog leitud teel on alati 1
    }
    vaadatud[tipp] = 1; // siin me oleme
    for (int i = 0; i < naabrite_arv[tipp]; i++) {
        int naaber = naabrid[tipp][i];
        // kui naabertippu saab minna (läbilaskevõime on positiivne)
        // ja me ei ole seal veel olnud, otsime teed naabrist edasi
        if (kaarte_maht[tipp][naaber] > 0 && !vaadatud[naaber] && dfs(naaber)) {
            // kui see tee viib lõppu
            kaarte_maht[tipp][naabrid[tipp][i]]--; // kaare läbilaskevõime
            kaarte_maht[naabrid[tipp][i]][tipp]++; // tagasikaare läbilaskevõime
            return 1;
        }
    }
    return 0; // teed ei leidunud
}
```

```
int main()
{
    cin >> p >> l >> n;
    int* x = new int[n];
    int* y = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }
    int valvureid = n;
    n *= 2; // iga valvuri jaoks on vaja 2 tippu graafis
    n += 2; // lisaks on vaja seinte jaoks tipud. need on lähe ja suue
    vaadatud = new bool[n];
    kaarte_maht = new int*[n];
    naabrid = new vector<int>[n];
    naabrite_arv = new int[n];
    for (int i = 0; i < n; i++) {
        naabrite_arv[i] = 0;
        vector<int> vi;
        naabrid[i] = vi;
        vaadatud[i] = 0;
        kaarte_maht[i] = new int[n] {0};
    }

    for (int i = 0; i < valvureid; i++) {
        if (y[i] <= 100) { // kui on alumise serva lähedal
            kaarte_maht[0][i * 2 + 1]++; // lisame serva
            naabrid[0].push_back(i * 2 + 1);
            naabrid[i * 2 + 1].push_back(0); // tagasiserv
            naabrite_arv[0]++;
            naabrite_arv[i * 2 + 1]++;
        }
        if (y[i] + 100 >= l) { // kui on ülemise serva lähedal
            kaarte_maht[i * 2 + 2][n - 1]++;
            naabrid[n - 1].push_back(i * 2 + 2);
            naabrid[i * 2 + 2].push_back(n - 1);
            naabrite_arv[n - 1]++;
            naabrite_arv[i * 2 + 2]++;
        }
        for (int j = 0; j < valvureid; j++) {
            if (((x[i]-x[j])*(x[i]-x[j]))+((y[i]-y[j])*(y[i]-y[j])) <= 40000) {
                if (j == i)
                    kaarte_maht[i * 2 + 1][i* 2 + 2]++; // kaar iseendaga
                else
                    kaarte_maht[i * 2 + 2][j * 2 + 1]++;
                naabrid[j * 2 + 1].push_back(i * 2 + 2);
                naabrid[i * 2 + 2].push_back(j * 2 + 1);
                naabrite_arv[j * 2 + 1]++;
                naabrite_arv[i * 2 + 2]++;
            }
        }
    }

    int vastus = 0;
    while (dfs(0)) {
        vastus ++;
        for (int i = 0; i < n; i++) {
            vaadatud[i] = 0; // nullime järgmiseks läbimiseks
        }
    }
    cout << vastus << endl;
}
```

10.7 KLIKID

Klikiks (i.k *clique*) nimetatakse graafi täielikku alamgraafi. Formaalsemalt on lihtgraafi G tippude alamhulk $S \subseteq V(G)$ klikk, kui iga kahe tipu $u, v \in S$ jaoks, kus $u \neq v$, leidub graafis G serv nende kahe tipu vahel. Alamhulka $S \subseteq V(G)$ nimetatakse sõltumatuks hulgaks siis, kui ühegi kahe hulka S kuuluva tipu vahel ei ole graafis G serva.



Klikke nimetatakse selle järgi, mitu tippu klikis on. Ülaloleval joonisel on näiteks 9 ühetipulist klikki (iga tipp on klikk), 12 kahetipulist klikki (iga tipupaar, mille vahel on serv), 5 kolmetipulist klikki (tekkivad kolmnurgad roheliste tippudega alas ja kollaste tippudega kolmnurk) ning üks 4-tipuline klikk (rohelised tipud).

Maksimaalseks klikiks (i.k. *maximal clique*) nimetatakse sellist klikki, millele ei saa lisada ühtki naabertippu, st sellist klikki, mis ei ole ühegi suurema kliki alamgraaf. Joonisel on viis maksimaalset klikki: rohelise tipuhulgaga klikk, kollase tipuhulgaga klikk ja kolm kahetipulist klikki, servaga rohelise ja punase tipu vahel, kahe punase tipu vahel ning punase ja kollase tipu vahel.

Mõnikord mõeldakse ka lihtsalt kliki all maksimaalset klikki.

Maksimumklikiks (i.k. *maximum clique*) nimetatakse kõige suurema tipuhulgaga klikki graafis. Joonisel oleva graafis on üks maksimumklikk – rohelise tipuhulgaga klikk.

Klikkide leidmise ülesanne on NP-keeruline.

10.7.1 Bron-Kerboschi algoritm

Bron-Kerboschi algoritm on nime saanud Hollandi teadlaste Coenraad Broni ja Joep Kerboschi järgi, kes avaldasid selle algoritmi kirjelduse aastal 1973.

Algoritmi näol on tegemist rekursiivse läbivaatusalgoritmiga, mis leiab kõik maksimaalsed klikid graafis. Täpsemalt, kui on antud kolm hulka R, P, ja X, leiab see maksimaalsed klikid, millesse kuuluvad kõik tipud hulgast R, mõned tipud hulgast P ja mitte ükski tipp hulgast X. Hulkade P ja X ühend on selliste tippude hulk, mis saavad klikki R laiendada. Kui P ja X on tühjad, on R maksimaalne klikk.

Algoritm ise on järgmine:

```
BK(R,P,X):
        kui P=∅ ja X=∅:
                R on maksimaalne klikk
        ∀v∈ P:
                BK(R∪{v}, P∩N(v), X∩N(v)))
                P = P \ {v}
                X = X ∪ {v}
```

10.7.2 Sõbrad

Keskkoolis on kõige tähtsam olla vinges sõprade grupis. Sõbrad korraldavad omavahel pidusid, kus kõik kohalolijad on omavahel sõbrad ja millest ükski õpilane, kes on kõigi kohalolijate sõber, ei puudu. Loomulikult võib iga õpilane osaleda erineva kosseisuga pidudel, kui eelnev tingimus on täidetud. Pidu aga pole päris see, kui kellelgi on mõned sõbrad peolt puudu.
Leia, mitu erineva koosseisu saab moodustada pidude tarbeks, nii et ühegi peo ühelgi osalejal ei oleks korraga rohkem kui p sõpra, kes sellel peol ei osale. Sõprus on sümmeetriline, st kui A on B sõber, siis B on ka A sõber.
Sisendi esimesel real on kaks mittenegatiivset täisarvu `n` ja `p`. Järgneval `n` real on info iga õpilase kohta alates õpilasest `i=0`. Iga rida algab täisarvuga m_i – `i.nda` õpilase väidetav sõprade arv. Selle järel on mi täisarvu `0..n-1` – tema sõprade numbrid. On teada, et õpilane ei nimeta kunagi ennast oma sõbraks.
Väljundi ainsale reale väljasta üks täisarv – ülesande tingimustele vastavate peokosseisude arv.

NÄIDE:

```
8 1
2 1 7
2 0 2
4 1 3 4 5
2 2 4
2 2 3
3 2 6 7
2 5 7
3 0 5 6
```

Vastus:

```
2
```

Selles ülesandes on võimalikud peokosseisud kõik graafis leiduvad klikid, seega on vaja kõigepealt leida kõik klikid sõprade graafis. Selleks saab kasutada eelpool kirjeldatud Bron-Kerboschi algoritmi:

```
vector<int>* graaf;
vector<vector<int>> klikid;

vector<int> yhend(vector<int> a, vector<int> b)
{
    vector<int> vastus;
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    int bi = 0;
    int ai = 0;
    while (ai < a.size()) {
        while (bi < b.size() && b[bi] < a[ai]) {
            bi++;
        }
        if (bi == b.size() || b[bi] > a[ai]) {
            vastus.push_back(a[ai]);
        }
        ai++;
    }
    return vastus;
}
```

```
vector<int> yhisosa(vector<int> a, vector<int> b)
{
    vector<int> vastus;
    sort(b.begin(), b.end());
    sort(a.begin(), a.end());
    int bi = 0;
    int ai = 0;
    while (ai < a.size()) {
        while (bi < b.size() && b[bi] < a[ai]) {
            bi++;
        }
        if (bi == b.size()) break;

        if (b[bi] == a[ai]) {
            vastus.push_back(a[ai]);
        }
        ai++;
    }
    return vastus;
}

vector<int> lisa_element(vector<int> a, int b)
{
    vector<int> vastus;
    vastus = a;
    vastus.push_back(b);
    return vastus;
}

void BronKerbosch(vector<int> r, vector<int> p, vector<int> x)
{
    if (p.size() == 0 && x.size() == 0) {
        klikid.push_back(r);
        return;
    }
    if (p.size() == 0) return;
    int rp = p[0];
    vector<int> pr = yhend(p, graaf[rp]);
    for (int i = 0; i < pr.size(); i++) {
        BronKerbosch(
          lisa_element(r, pr[i]), yhisosa(p, graaf[pr[i]]), yhisosa(x, graaf[pr[i]]));
        vector<int> dmp;
        dmp.push_back(pr[i]);
        p = yhend(p, dmp);
        x.push_back(pr[i]);
    }
}
```

Nüüd, kui kõik klikid on leitud, tuleb leida nende seast selliseid, mille ühelgi tipul ei ole rohkem kui p naabrit, mis ei kuulu antud klikki:

```
int leia_vastus()
{
    int vastus = klikid.size();
    for (int i = 0; i < klikid.size(); i++) {
        int kliki_suurus = klikid[i].size();
        for (int j = 0; j < kliki_suurus; j++) {
            int v = klikid[i][j];
            if (graaf[v].size() - kliki_suurus + 1 > q) {
               //lahutame need klikid, kus on mõnel tipul rohkem klikist väljuvaid
               //servi kui lubatud
                vastus--;
                break;
            }
        }
    }
    return vastus;
}
```

10.8 KONTROLLÜLESANDED

10.8.1 Laadimispunktid

Uus Euroopa Liidu seadus näeb ette, et iga linn, mida peab kindlasti läbima, et saada mingist teisest linnast kolmandasse linna, oleks varustatud elektriautode laadimispunktiga. Seega pöördus valitsus sinu poole, et teada saada, mis linnades peavad uue seaduse kohaselt laadimispunktid olema.
Sisendi esimesel real on arv `n` (`2≤n≤100`): võrgus olevate linnade arv. Järgmisel `n` real on väikeste ladina tähtedega linna nimi. Nendele järgneb eraldi real arv `m`: teede arv. Järgneval `m` real on kaks sõnet: tee alguslinn ning lõpplinn.
Väljundi esimesele reale kirjutada täisarv `d`: mitmes linnas peavad laadimispunktid olema. Järgmisele `d` reale kirjutada tähestiku järjekorras vastavate linnade nimed.

NÄIDE:

```
6
tallinn
tartu
narva
parnu
kuressaare
kardla
7
parnu narva
narva tallinn
parnu tallinn
tartu kardla
tallinn tartu
kuressaare tallinn
kardla kuressaare
```

Vastus:

```
1
tallinn
```

10.8.2 Internetiühendus

Kord ennemuistsetel aegadel polnud igal Eesti koolil ligipääsu internetile ning valitsus otsustas probleemi lahendada. Kuna internet oli sel ajal veel väga haruldane asi, oleks igale koolile otseühenduse kindlustamine väga kallis olnud, seega otsustati selle asemel ühendada koole omavahel. Koolil on ligipääs Internetile juhul, kui sellel on kas otseühendus või on see ühendatud kooliga, millel on ligipääs Internetile. Leida minimaalne hind, millega kindlustada igale koolile ligipääs Internetile ning mitu otseühendust on selle jaoks vaja. Kuna riigile meeldib hea välja näha, väljastada samahinnalistest plaanidest see, mis sisaldab rohkem otseühendusi.

Sisendi esimesel real on kolm tühikutega eraldatud täisarvu: `n`, `m` ja `a` ($1 \le n \le 10000$, $1 \le m \le 100000$, $0 < a \le 10000$) – koolide arv, võimalike koolide ühendamise viiside arv ning otseühenduse loomise hind. Järgmisel `m` real on info ühenduste kohta. Igal real on kolm tühikutega eraldatud täisarvu: `x`, `y` ja `c` ($1 \le x, y \le n$, $0 < c \le 10000$) – ühenduses esimene kool, teine kool ning ühenduse hind. Tuleb tähele panna, et kahe kooli ühendamiseks võib olla mitu erihinnalist plaani.

Väljundi ainsale reale väljastada kaks tühikutega eraldatud täisarvu: a ja b – projekti läbiviimiseks vajalik minimaalne hind ning selle jaoks vajalik internetiga otseühenduste arv.

NÄIDE 1:

```
4 4 100
1 2 10
4 3 12
4 1 41
2 3 23
```

Vastus:

```
145 1
```

NÄIDE 2:

```
5 3 1000
1 2 20
4 5 40
3 2 30
```

Vastus:

```
2090 2
```

10.8.3 Kiviheitemasin

Keskajal oli ainus praktiline viis ühe sõnumi ühest kohast teise saamiseks kiviheitemasina kasutamine*. Kiviheitemasin saab visata teateid igasse linna, mis on selle viskekauguse ulatuses. Konfliktide vältimiseks otsustati, et igal linnal on kiviheitemasin võimeline täpselt sama kaugele laskma ning et see kaugus on minimaalne vajalik, et kõik linnad oleksid omavahel kuidagi ühendatud (A ja B on ühendatud parajasti siis, kui A saab B-ga otse ühendust pidada või kui A saab mingi B-ga ühendatud linnaga ühendust pidada). Leida see minimaalne kaugus.
Sisendi esimesel real on täisarv p $(0<p\leq 500)$ – linnade koguarv. Järgmisel p real on kaks tühikuga eraldatud täisarvu: x ja y $(0\leq x,y\leq 10000)$ linna x ja y koordinaat.
Väljundisse kirjutada üks reaalarv: minimaalne kaugus, mis kiviheitemasinal peab olema, et kõik linnad oleksid ühendatud.

NÄIDE:

```
4
0 100
0 300
0 600
150 750
```

Vastus:

```
300
```

*fakt!

10.8.4 Uus tee

Riigis on n (1≤n≤50) linna, mis on ühendatud omavahel m (1≤m≤1225) teega. Riigil läheb hästi ning seega otsustatakse kahe linna, millel puudub otseühendus, vahele luua otseühendus. Selle jaoks võetakse arvesse parima kasumi valemit: kasum(u,v)=Sum(pc(i,j)−cc(i,j)), kus iga linnapaari i,j jaoks on pc(i,j) lühima tee pikkus linnast i linna j, kui tee u ja v vahel puudub, ning cc(i,j) on lühima tee pikkus linnast i linna j, kui tee u ja v vahel eksisteerib. Leida, milliste kahe linna vahel on selle valemi järgi kõige parem teed ehitada. Juhul, kui selliseid maksimume on mitu, väljastada nendest teedest lühim. Juhul kui selliseid teid on ikka mitu, väljastada vähimate otspunktidega tee. Sisendi esimesel real on kaks arvu: n ja m. Järgmisel n real on kaks tühikuga eraldatud täisarvu: x ja y (-1000≤x,y≤ 1000), linna koordinaadid. Järgmisel m real on kaks tühikuga eraldatud täisarvu: a ja b (1≤a,b≤n), tee otspunktid. Väljundi ainsale reale kirjutada kaks tühikuga eraldatud täisarvu: uue tee algus- ja lõpp-punkt. Juhul, kui parim kasum ≤ 1 või pole võimalik uut teed luua, väljastada arvupaar 0, 0.

NÄIDE 1:

```
4 6
0 0
0 2
2 0
2 2
1 2
1 3
1 4
2 3
2 4
3 4
```

Vastus:

```
0 0
```

NÄIDE 2:

```
4 4
0 0
0 2
2 0
2 2
1 2
2 3
3 4
4 1
```

Vastus:

```
1 3
```

NÄIDE 3:

```
4 3
0 0
0 2
2 0
2 2
1 2
2 3
3 4
```

Vastus:

```
1 3
```

10.8.5 Mõjuvõim

Kuritegelikud organisatsioonid uurivad, kui palju neil mõjuvõimu on. Iga organisatsioon mõjutab mingit hulka teisi organisatsioone ning on ka teada, et kui organisatsioon A mõjutab organisatsiooni B ning organisatsioon B mõjutab organisatsiooni C, mõjutab organisatsioon A kaudselt organisatsiooni C, aga organisatsioon A ei mõjuta otseselt organisatsiooni C. Samuti ei ole tsükleid, nii et organisatsioon A ei mõjuta kaudselt iseennast. Me teame iga organisatsiooni kohta, mitu organisatsiooni see otseselt ja kaudselt mõjutab. Leida selle info põhjal, mis organisatsioone mõjutab iga organisatsioon otseselt.

Sisendi esimesel real on täisarv `n` ($1 \le n \le 120$), organisatsioonide arv. Järgmisel `n` real on üks sõne – organisatsiooni nimetus. Seejärel on eraldi real täisarv `m` ($1 \le m \le n$), organisatsioonide arv, mis mõjutavad mingeid teisi organisatsioone. Järgmisel `m` real on tühikutega eraldatud sõned. Esimene on mõjutava organisatsiooni nimi. Seejärel on arv `k` – mõjutatavate organisatsioonide arv. Siis on `k` sõnet – mõjutatava organisatsiooni nimi.

Väljundisse kirjutada tähestiku järjekorras `m` organisatsiooni nime, millele järgneb tühikuga eraldatuna selle organisatsiooni poolt otseselt mõjutatavate organisatsioonide arv ning nende organisatsioonide nimetused tähestiku järjekorras.

NÄIDE:

```
5
maranzano
profaci
mangano
luciano
gagliano
3
mangano 4 gagliano luciano maranzano profaci
gagliano 3 luciano maranzano profaci
luciano 1 profaci
```

Vastus:

```
gagliano 2 maranzano luciano
luciano 1 profaci
mangano 1 gagliano
```

10.8.6 Civilization

Arvutimängus „Side Meier's Civilization VI" kontrollivad mängijad riike, kellel on maavarad. Mängus on korraga n $(2 \le n \le 10)$ riiki ning m $(5 \le m \le 25)$ erinevat maavara. Igal riigil on igast maavarast mingi arv koopiat (koopiate arv võib olla ka 0). Riigid saavad omavahel vahetada maavarasid kursiga $1:1$, aga enamus riikidest vahetavad ära ainult varasid, millest neil on rohkem kui üks koopia, varade vastu, millest neil on null koopiat. Juku taipas, et kui eesmärk on saada võimalikult palju erinevaid maavarasid, on vahepeal kasulik teha selliseid tehinguid, kus ta annab ära olemasoleva maavara teise maavara vastu, mida tal juba on. Aita Jukul leida, mis on maksimaalne erinevate maavarade arv, mis tal saab olla, kui kõik teised mängijad vahetavad ainult eelnevalt mainitud reeglite põhjal.
Sisendi esimesel real on kaks tühikutega eraldatud täisarvu: n ja m. (n sisaldab ka Jukut). Järgmisel n real on kirjeldus iga mängija maavarade kohta. Esimene nendest kirjeldab Juku maavarasid. Igal real on mingi arv tühikutega eraldatud täisarve, millest esimene on k_i $(0 \le k_i \le 50)$, sellele riigile kuuluvate maavarade arv ning sellele järgneb k_i arvu, riigile kuuluvate maavarade indeksid.
Väljundisse kirjutada üks täisarv: Maksimaalne võimalik erinevate maavarade arv, mida Juku võib saavutada.

NÄIDE 1:

```
2 5
6 1 1 1 1 1 1
3 1 2 2
```

Vastus:

```
1
```

NÄIDE 2:

```
3 5
4 1 2 1 1
3 2 2 2
5 1 3 4 4 3
```

Vastus:

```
3
```

10.8.7 Põgenemine

Indiana Jones on kogemata käivitanud lõksu, mille tagajärjel variseb enamus toast, kus on ka palju teisi inimesi, kokku. Tuba on mõõtmetega `N * M (1≤N,M≤30)` ning koosneb viiest erinevast ruudust:
`*` – inimene ohtlikul platvormil. Inimene tahab siit võimalikult ruttu ära saada, sest lõpuks kukub see ka kokku. Inimene saab liikuda ükskõik missuguse põhiilmakaare poole ning platvorm kukub dramaatilisuse huvides alla kohe sel hetkel, kui inimene selle pealt maha astub.
`~` – kuristik. Kui inimene satub kuristikku, kukub ta alla ning hukkub.
`.` – ohtlik platvorm. Kui inimene astub sellele peale, käitub see nagu „inimene ohtlikul platvormil“ tüüpi ruut.
`@` - vähem ohtlik platvorm. See on platvorm, mis küll kukub lõpuks alla, aga see ei taha olla dramaatiline ning kukub alla alles siis, kui aeg läbi saab. Tuleb tähele panna, et ohtlikul ning vähem ohtlikul platvormil tohib korraga peal olla maksimaalselt üks inimene.
`#` - ohutu platvorm. See platvorm ei kuku kunagi alla. Igal ohutul platvormil võib olla maksimaalselt `P (1≤P≤10)` inimest.
Sulle antakse ette toa kirjeldus. Sinu eesmärk on leida, mis on maksimaalne võimalik ellujääjate arv, kui eeldada, et aega on piisavalt palju, et vastavat strateegiat kasutada.
Sisendi esimesel real on kolm tühikutega eraldatud täisarvu arvud `N, M` ja `P`.
Järgmisel `n` real on `m` tähemärgist koosnev jada: toa kirjeldus.

NÄIDE 1:

```
3 4 2
*~~#
...@
.~.*
```

Vastus:

2

NÄIDE 2:

```
3 5 1
~~*~~
#.@.#
~~*~~
```

Vastus:

2

NÄIDE 3:

```
1 4 2
**#~
```

Vastus:

1

10.8.8 Teepuhastus

Tartu linnas on ainult üks lumepuhastusmasin. Leida minimaalne aeg, mis kulub, et see puhastaks kõik Tartu tänavad lumest ning jõuaks oma koju tagasi. Lumepuhastusmasina kiirus puhastamisel on 20km/h ning puhastatud teel sõitmine on 50km/h. Iga tänav on mõlemas suunas ühesuunaline ning alguses puhastamata olekus. Lumepuhastusmasin saab korraga ainult ühte sõidurida puhastada.
Sisendi esimesel real on kaks tühikuga eraldatud täisarvu: x ja y, lumepuhastusmasina jaama koordinaadid. Järgmisel real on tänavate arv n (1≤n≤100). Järgmisel n real on neli tühikutega eraldatud täisarvu: x_1, y_1, x_2, y_2 - tänava otspunktide koordinaadid. On teada, et jaamast saab igale tänavale sõita ainult tänavaid mööda. Iga koordinaat on antud meetrites ning on täisarv vahemikus -100000 kuni 100000.
Väljundi ainsale reale kirjutada aeg tundides ja minutites, mis on vaja, et kogu linn ära puhastada. Vastus väljastada täpsusega 1 minut.

NÄIDE:

```
0 0
3
0 0 10000 10000
5000 -10000 5000 10000
5000 10000 10000 10000
```

Vastus:

```
3:55
```

10.8.9 Doomino

Sul on n (1≤n≤1000) doominoklotsi, mida sa tahad järjest ringi panna, nii et kahel järjestikusel klotsil on ühendumiskohal sama number ning viimane klots on ühendatud esimesega. Leida, kas selline järjestus eksisteerib.
Sisendi esimesel real on täisarv n. Järgmisel n real on kaks tühikutega eraldatud täisarvu: a ja b (1≤a,b≤50), doominoklotsile kirjutatud arvud.
Väljundi ainsale reale väljastada üks sõna. Kui kõiki klotse on võimalik ringi panna, väljastada JAH, kui ei saa, väljastada EI.

NÄIDE 1:

```
5
1 2
2 3
3 4
4 5
5 6
```

Vastus:

```
EI
```

NÄIDE 2:

```
2 1
2 2
3 4
3 1
2 4
```

Vastus:

```
JAH
```

10.8.10 Valimised

USA-s on mängusaade, mida nimetatakse USA presidendivalimisteks, kus osaleb d demokraati ja v vabariiklast ($1 \le d, v \le 100$). Pärast saadet hääletavad n ($0 \le n \le 500$) vaatajat ühe inimese poolt, kellest nad tahavad, et jääks saatesse alles ning ühe inimese poolt, keda nad tahavad välja visata. Kuna iga ameeriklane on ise kas demokraat või vabariiklane, valivad demokraadid alati demokraadi, kes saatesse peaks jääma, ja vabariiklase, keda välja visata, ning vabariiklased alati vabariiklase, kes saatesse peaks jääma, ja demokraadi, keda välja visata. Saatejuhid otsustasid inimesi välja visata nii, et oleks võimalikult palju hääletajaid, kelle mõlemad hääled lähevad arvesse – see tähendab, et nende lemmik jääb saatesse ja vihatu lahkub. Leida see hääletajate arv.
Sisendi esimesel real on kolm tühikutega eraldatud täisarvu: `d`, `v` ja `n`. Järgmisel n real on kaks tühikuga eraldatud sõnet. Esimene on osaleja kood, keda vaataja tahab alles jätta ning teine on kandidaat, keda vaataja tahab välja visata. Osaleja kood koosneb tähest `D` või `V` – demokraat või vabariiklane ning ühest arvust, mis näitab, mitmenda demokraadi või vabariiklasega tegu on.
Väljundisse kirjutada 1 täisarv: maksimaalne arv rahulolevaid hääletajaid.

NÄIDE 1:

Sisend:

```
1 1 2
D1 V1
V1 D1
```

Vastus:

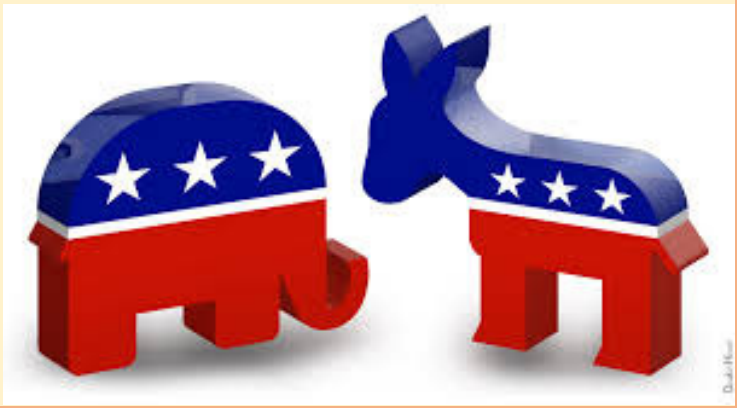
```
1
```

NÄIDE 2:

```
1 2 4
D1 V1
D1 V1
D1 V2
V2 D1
```

Vastus:

```
3
```



10.9 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk9 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
Tuleb hiljem	Floyd-Warshall
Servad.cpp, Servad.java, Servad.py	Graafi servade liigitamine
Sillad.cpp, Sillad.java, Sillad.py	Sildade leidmine graafis
Muinastuled.cpp, Muinastuled.java, Muinastuled.py	Artikulatsioonipunktide leidmine
Loomad.cpp, Loomad.java, Loomad.py	Euleri tsükkel (Hierholzer)
Elektrikatkestus.cpp, Elektrikatkestus.java, Elektrikatkestus.py	Minimaalne toes (Kruskal)
Vangid.cpp, Vangid.java, Vangid.py	Maksimaalne voog/minimaalne lõige
Sobrad.cpp, Sobrad.java, Sobrad.py	Klikid graafis