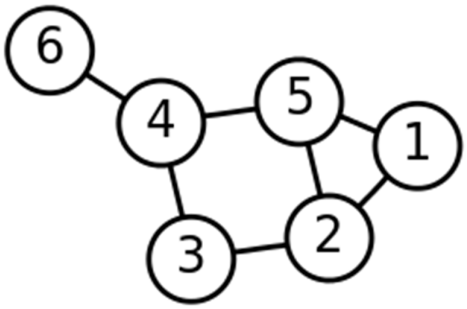
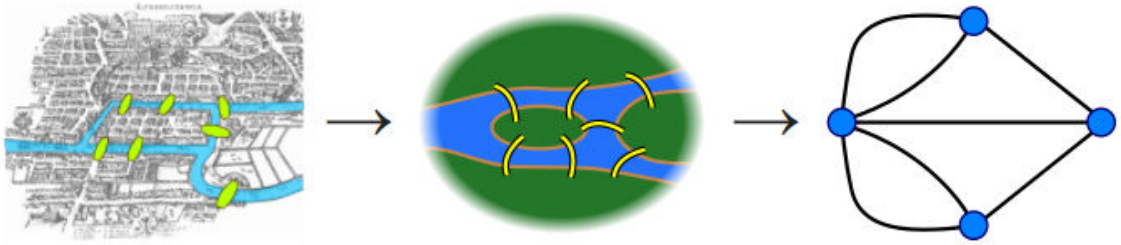
6 SISSEJUHATUS GRAAFITEOORIASSE



Graaf on struktuur, mida kasutatakse objektide vaheliste paarikaupa seoste kujutamiseks. Neid objekte nimetatakse graafi **tippudeks** (*vertex,* joonisel numbritega) ning seoseid graafi **servadeks** (*edge*, joonisel jooned tippude vahel). **Graafiteooria** on matemaatika haru, mis uurib graafe.

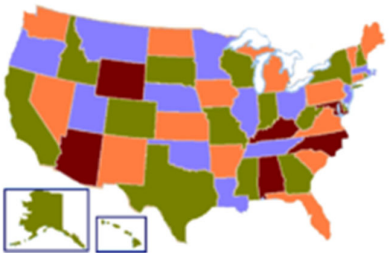
Graafe kasutatakse väga erinevate suhete ja protsesside modelleerimiseks, seda nii füüsilistes, bioloogilistes, sotsiaalsetes kui ka infosüsteemides. Näiteks võib graafina kujutada veebilehti ja nendevahelisi linke, sotsiaalseid „kes keda tunneb“ võrgustikke, aatomite vahelisi seoseid molekulis ja veel paljusid teisi struktuure.

Maailma esimeseks graafiteooria alaseks tööks peetakse 1736. aastal Leonhard Euleri avaldatud „Königsbergi sildade“ probleemi: Königsbergi linnas on seitse silda - kas on võimalik koostada jalutuskäiguteekond, mis ületab iga silla täpselt ühe korra?



Eitava vastuseni jõudmine pole raske, kuid Euleril tuli nullist alates luua ülesande lahendamiseks vajalik tehnika ja terminoloogia. Ülaltoodud joonis kujutab ka vastavat mõtlemisprotsessi: veekogude ja kallaste sisemised eripärad ei ole tegelikult tähtsad ja neid on võimalik abstraheerida sellisele tasemele, et alles jäävad vaid objektid ja seosed (tipud ja servad). Selline abstraktsioon ongi graafiteoreetilise mõtlemise tugevus - leitud tulemusi ja algoritme saab kasutada kõigis rakendustes veevärgist sotsiaalmeediani ja maakaartidest tuumauuringuteni.

Nagu diskreetse matemaatika puhul üldiselt, andis ka graafiteooriale hoo sisse arvutite areng ja antud juhul aitas see isegi täiesti otseselt uusi matemaatilisi resultaate saavutada.



Üks graafiteooria tuntumaid probleeme on **neljavärviprobleem**, mis küsib, kas mistahes tasapinnalise kaardi värvimiseks piisab neljast erinevast värvist, nii et iga riigi külg puutuks kokku vaid temast erinevat värvi naabriga. Tõestamaks, et selline värvimine on tõesti võimalik, leiti kõigepealt, et kõik erinevad värvimisvõimalused on taandatavad 1936 erinevale riikide asetuse kombinatsioonile. Nende kombinatsioonide läbivaatamiseks kasutasid Kenneth Appel ja Wolfgang Haken 1976. aastal arvuti abi – see oli ka esimene kord, kus olulise matemaatilise tulemuseni jõudmiseks arvutit kasutati.

6.1 GRAAFITEOORIA TERMINID

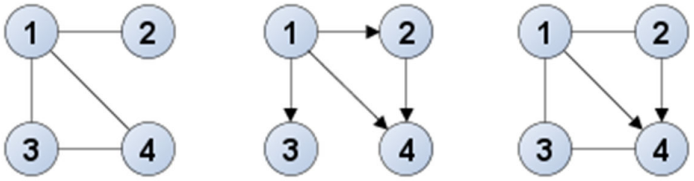
Matemaatiliselt tähistatakse graafi $G = (T, S)$, kus T on mittetühi tippude hulk ja S on paaride (x, y) hulk, kus $x, y \in T$.

Kui graafi servade hulk on tühi, nimetatakse sellist graafi **tühigraafiks**, kui serv on iga tipupaari vahel, nimetatakse sellist graafi **täielikuks**.

6.1.1 Suunatud ja suunamata servad

Graafi tippe x ja y ühendaval serv võib olla **suunatud** või **suunamata**. Kui suunda pole, siis see tähendab, et mööda serva saab liikuda nii tipust x tippu y kui ka vastupidi, sel juhul nimetatakse sellist serva suunamata servaks või lihtsalt servaks ja tähistatakse $s = \{x, y\}$. Kui aga serval on suund määratud, siis nimetatakse sellist serva suunatud servaks ehk **kaareks** ning tähistatakse $s = (x, y)$. Mööda sellist kaart saab liikuda tipust x tippu y, aga mitte vastupidi. Joonistel tähistatakse suunamata servi joontena ning kaari nooltena.

Kui graafi kõik servad on suunamata, siis sellist graafi nimetatakse **suunamata graafiks** *(undirected graph)* ehk lihtsalt graafiks. Graafi, milles kõik servad on suunatud, nimetatakse **suunatud graafiks** *(directed graph)*. Kui graafil on nii suunatud kui ka suunamata servi, siis on tegemist **segagraafiga**.

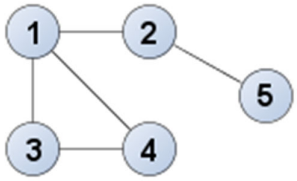


Esimene graaf joonisel on suunamata graaf, teine suunatud graaf ja kolmas segagraaf.

6.1.2 Teed graafis

Marsruudiks ehk **teeks** (*path*) graafis $G = (T, S)$ nimetatakse servade jada $s_1 = \{t_0, t_1\}$, $s_2 = \{t_1, t_2\}, \ldots, s_n = \{t_{n-1}, t_n\}$, milles iga serva lõpptipp on talle järgneva serva algustipp.

Ahel (*chain*) on selline marsruut, mille kõik servad on erinevad. Ahelat, mis ei läbi ühtki tippu rohkem kui üks kord, nimetatakse **lihtahelaks** (*simple chain*)**.** Järgneval joonisel moodustavad näiteks servad {1, 3}, {3, 4}, {4, 1}, {1, 2}, {2, 5} ahela ja servad {3, 4}, {4, 1}, {1, 2} ja {2, 5} lihtahela:



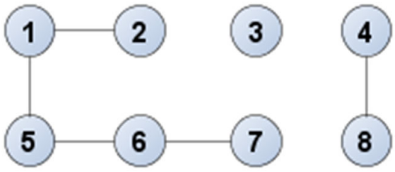
Tsükkel (*cycle*) on selline ahel, mille algus- ja lõpptipp on samad. Kui tsükli moodustab lihtahel, siis sellist tsüklit nimetatakse **lihttsükliks**. Ühest servast koosnevat tsüklit $s = \{t, t\}$ nimetatakse **silmuseks** (*loop*). Eelmisel joonisel on tsükkel näiteks {1, 3}, {3, 4} ja {4, 1}.

6.1.3 Sidususkomponendid

Graafi $G = (T,\ S)$ **alamgraafiks** (*subgraph*) nimetatakse mistahes graafi $G' = (T',\ S')$, mille korral kehtib $T' \subseteq T$ ja $S' \subseteq S$. See tähendab, et alamgraaf on saadav ülemgraafist mingi hulga tippude ja servade eemaldamise teel. Asjaolu, et G' on G alamgraaf, tähistatakse $G' \subseteq G$.

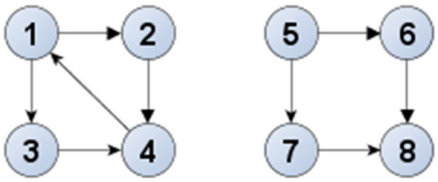
Öeldakse, et suunamata graaf on **sidus** (*connected*), kui iga tema tipupaari korral leidub marsruut ühest tipust teise. Seni toodud joonistel on olnud sidusad graafid.

Graafi G alamgraafi G' nimetatakse graafi G **sidususkomponendiks** (*connected component*), kui G' on sidus ja graafis G ei leidu sidusat alamgraafi G'', nii, et $G' \subset G''$. See tähendab, et sidususkomponent on maksimaalne sidus alamgraaf, seda ei saa kasvatada talle uute servade või tippude lisamisega ilma sidusust rikkumata.



Sellel graafil on kolm sidususkomponenti. Esimese moodustavad tipud 1, 2, 5, 6 ja 7 ning nendevahelised servad, teise tipp 3 ja kolmanda tipud 4 ja 8 koos nendevahelise servaga.

Sidusust ja sidususkomponente saab määrata ka suunatud graafidel. Suunatud graafi G nimetatakse sidusaks, kui kaarte asendamisel suunamata servadega saadud suunamata graaf G' on sidus. Kui suunatud graafis G leidub iga tipupaari korral marsruut esimesest tipust teise ja vastupidi, nimetatakse graafi G **tugevalt sidusaks** (*strongly connected*). Suunatud graafis saab eristada sidususkomponente ja **tugevalt sidusaid komponente**.



Joonisel on suunatud graaf, millel on kaks sidususkomponenti: esimene tippudega 1,2 3 ja 4 ning teine tippudega 5, 6, 7 ja 8 koos nendevaheliste kaartega. Vasakpoolne sidususkomponent on ka tugevalt sidus, parempoolne aga mitte.

6.2 GRAAFIDE ESITUSVIISID

Graafiteooriat õpetades on väga mugav graafe visuaalselt kujutada, programmi kirjutades on aga sellest vähe kasu. Kuidas siis graafe kujutada? Enim levinud on kolm põhimõttelist erinevat viisi, mida omakorda saab erinevaid andmestruktuure kasutades realiseerida. See, millist esitust valida, sõltub peamiselt sellest, mida graafiga edasi vaja teha on.

6.2.1 Naabrusmaatriks

Naabrusmaatriks (*adjacency matrix*) on kõige universaalsem ja lihtsam esitusviis. Naabrusmaatriks on maatriks, milles on üks rida ja üks veerg iga graafi tipu kohta. Tipule t_1 vastava rea ja tipule t_2 vastava veeru ristumiskohal on info kaare (t_1, t_2) kohta.

Naabrusmaatriksit on mugav hoida *n* x *n* massiivis M, kus n on graafi tippude arv. Minimaalne variant on see, kui tegemist on kaalumata graafiga – sellisel juhul piisab, kui massiivis hoida tõeväärtusi (või arve 0 ja 1), vastavalt sellele, kas antud kaar on olemas või mitte. Selle peatüki esimesel pildil olev graaf naabrusmaatriksina:

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

Kaalutud graafi korral võib maatriksis hoida serva kaalu, sellisel juhul on oluline valida serva puudumise märkimiseks sobiv arv, mis ei läheks sassi serva kaaluga. Kui serva kohta on vaja hoida rohkem informatsiooni, siis võib massiivis hoida ka keerulisemaid kirjeid või kasutada 3-mõõtmelist massiivi.

Täielikult suunamata graafi korral on naabrusmaatriks peadiagonaali suhtes sümmeetriline ning sellisel juhul piisab, kui täita vaid pool tabelit.

Naabrusmaatriksi eelised:

- Servade lisamine ja eemaldamine on lihtne ja kiire.
- Kontroll, kas mingi tipupaari vahel on serv, on lihtne ja kiire.

Naabrusmaatriksi puudused:

- Kui tippe on palju, aga servi vähe, siis on selline esitus küllaltki mälu raiskav. Kui tippe on juba tuhandeid, on tõenäoliselt kasulikum mõnd muud esitusviisi kasutada.
- Tipu kõigi naabrite leidmiseks – protseduur, mida tuleb graafialgoritmides sageli ette – on naabrusmaatriksis vaja läbi käia terve maatriksi rida. Operatsiooni keerukus on $O(n)$, kus n on tippude arv graafis.

6.2.2 Tippude loend

Tippude loendis (*adjacency list*) on iga tipu kohta loend, kuhu sellest tipust serv läheb:

1: 2, 5
2: 1, 3, 5
3: 2, 4
4: 3, 5, 6
5: 1, 2, 4
6: 4

Tipust väljuvate kaarte loendeid on mugav hoida kas lihtahelas või dünaamilises massiivis, kuna nende pikkused on muutlikud. Loendeid endid võib hoida samuti ahelas või tavalises massiivis

pikkusega n, kus n on tippude arv graafis. Kui on vaja säilitada lisainformatsiooni tipu kohta, võib kasutada 2-mõõtmelist massiivi:

	1	2	3	4	5	6
Tipu nimi	karu	rebane	hiir	elevant	siga	kass
Naabrid	[2, 5]	[1, 3, 5]	[2, 4]	[3, 5, 6]	[1, 2, 4]	[4]

Kui on vaja hoida infot kaarte kohta, võib naabrite ahelas/massiivis kasutada keerulisemaid kirjeid, näiteks kaalutud graafis võib kasutada paare siht-tipu indeksist ja serva kaalust.

Tippude loendi eelised:

- vajadusel on tippude kohta mugav säilitada lisainfot,
- tipu naabrite leidmine ja töötlemine on kiire, see on vajalik paljudes graafitöötlusalgoritmides.

6.2.3 Servade loend

Servade loend (*edge list*) on loend graafi servadest. Siin on joonisel 1 toodud graafi esitus loeteluna servadest:

{1, 2}, {1, 5}, {2, 3}, {2, 5}, {3, 4}, {4, 5}, {4, 6}

Üks võimalus on selle realiseerimiseks kasutada 2-mõõtmelist massiivi, eriti juhul, kui servade arv on ette teada:

Esimene tipp	1	1	2	2	3	4	4
Teine tipp	2	5	3	5	4	5	6
Kaal	76	87	73	62	14	36	43

Sageli on aga mugavam säilitada serv eraldi struktuurina (näiteks kolmikuna) ning graaf on siis servade massiiv või ahel.

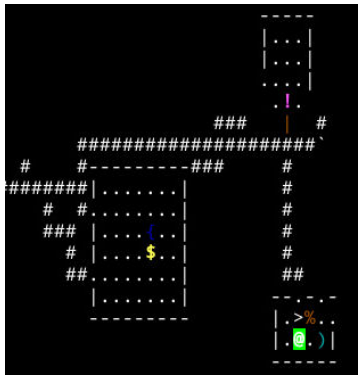
Servade loendi eelised:

- Mugav kasutada, kui servade kohta on palju lisainformatsiooni.
- Lihtsam kasutada, kui operatsioone on vaja sooritada just servadel. Näiteks on algoritme, kus on kasulik servi kaalu järgi jooksvalt sorteerida, seda on servade loendi abil mugav teha.

Servade loendi puudused:

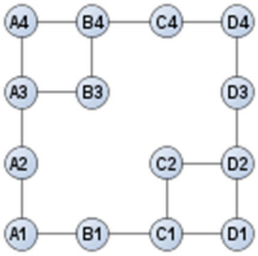
- Kui graafis on suunatud servi, tuleb kõik servad topelt salvestada.
- Tipu naabrite leidmiseks tuleb läbi käia kõik servad. Selle puuduse tõttu kasutatakse kõnealust esitlusviisi siiski harva.

6.2.4 Regulaarsete servadega graafid



Mõnel juhul on lihtne määrata, kas mingi kahe tipu vahel on serv või mitte. Nimetame siin selliseid graafe regulaarsete servadega graafideks. Üks lihtsamaid ja levinumaid regulaarsete servadega graafe on kahemõõtmelised kaardid:

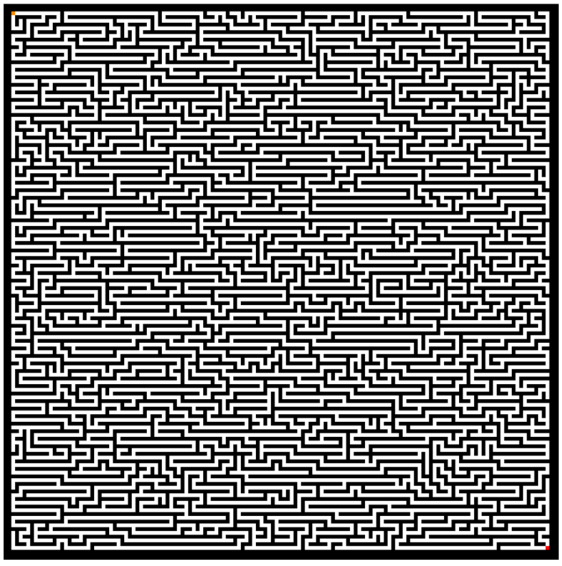
```
....
..#.
.#..
....
```



Kõikvõimalikke kaarte, labürinte jms saab sel moel graafina esitada. Iga märk selles ruudustikus on graafi tipuks ja servad on harilikult kõrvuti paiknevate märkide vahel. Kuna servad on nii regulaarsed, siis pole vaja luua eraldi andmestruktuuri nende hoidmiseks, piisab, kui hoida tippe kahemõõtmelises massiivis. Iga tipu naabertippude indeksid on kergesti arvutatavad:

(i-1, j-1)	(i-1, j)	(i-1, j+1)
(i, j-1)	(i, j)	(i, j+1)
(i+1, j-1)	(i+1, j)	(i+1, j+1)

6.3 GRAAFI LÄBIMINE



Graafi algoritmide juures on enamasti kõige olulisemaks protseduuriks graafi tippude töötlemine liikudes mööda servi tipust tippu. Sellisel moel graafi läbivaatamist nimetataksegi graafi läbimiseks. Graafi läbimiseks on kaks põhilist viisi: sügavuti läbimine ja laiuti läbimine.

6.3.1 Sügavuti läbimine

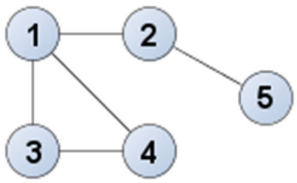
Sügavuti läbimise korral alustatakse graafi läbimist mingist tipust A ja liigutakse edasi selle uuele naabrile, siis omakorda järgmisele veel külastamata naabrile jne. Kui tipul on mitu naabrit, valitakse üks ja liigutakse mööda seda haru edasi. Kui kogu haru on läbitud, võetakse järgmine naaber ja vaadeldakse kogu see haru ja jätkatakse nii, kuni kõik harud on läbi vaadatud.

Kõige lihtsam on seda teha rekursiivselt. Lisainfona on vaja meeles pidada, millised tipud on juba külastatud, et vältida lõputusse tsüklisse minekut.

```
vector<int> graaf[tippude_arv]; // graaf tippude loendina
bool kaidud[tippude_arv] = { 0 }; // alguses kõik tipud on käimata

void otsi(int tipp)
{
    kaidud[tipp] = true; // siin me oleme
    tootle(tipp); // teeme midagi selle tipuga
    for (int i = 0; i < graaf[tipp].size(); i++) {
        int naaber = graaf[tipp][i]; // iga naabertipuga
        if (kaidud[naaber] == false) // kus ei ole veel käinud
            otsi(naaber); // kordame sama protseduuri
    }
}
```

Olenevalt sellest, kus asub töötlemise protseduur, eristatakse ees- ja lõppjärjestuses graafi töötlemist. Ülaltoodud näites töödeldakse tipp enne naabrite kallale asumist ja seega on tegu eesjärjestuses läbimisega. Kui töötlemine toimub pärast naabreid, on tegemist lõppjärjestuses läbimisega.

See, millises järjekorras tipud täpselt läbitakse, sõltub nii alguskohast kui sellest, kuidas naabreid valitakse (naabrite järjestusest). Selles mõttes ei ole graafil üht kindlat sügavuti läbimisel saadavat järjestust, isegi juhul, kui alustada läbimist samast tipust. Näiteks graafil:



on neli võimalikku tippude eesjärjestuses töötlemise järjestust sügavuti läbimisel, kui alustada tipust 1: need on (läbitavate tippude järjekorras) 1, 2, 5, 4, 3; 1, 2, 5, 3, 4; 1, 4, 3, 2, 5 ja 1, 3, 4, 2, 5. Tipust 5 alustades aga on võimalikke eesjärjestuses sügavuti läbimise järjestusi vaid 2: 5, 2, 1, 3, 4 ja 5, 2, 1, 4 3.

Sügavuti läbimise keerukus sõltub valitud andmestruktuurist. Oletades, et töötlemisoperatsioon on konstantse keerukusega, siis juhul, kui kasutatakse naabrusmaatriksit, tuleb kogu maatriks läbi käia ja keerukuseks on $O(T^2)$, kus T on tippude arv. Kui valitud on tippude loend, siis on keerukuseks $O(T + S)$, kus S on servade arv.

Sügavuti läbimist on hea kasutada ka tsüklite otsimiseks. Kui mõni vaadeldava tipu naabritest on juba vaadeldud, on graafis tsükkel:

```
bool on_tsykkel(int tipp, int eelmine)
{
    kaidud[tipp] = true;
    for (int i = 0; i < graaf[tipp].size(); i++) {
        int naaber = graaf[tipp][i];
        if (kaidud[naaber] == false) {
            if (on_tsykkel(naaber, tipp))
                return true;
        }
        else if (naaber != eelmine)
           return true;
    }
    return false;
}
```

6.3.2 Ülesanne: Ratsu teekond

Graafi läbimise illustreerimiseks kasutatakse sageli ülesandeid malelaual:

Leia selline ratsu teekond malelaual, mille puhul ratsu külastaks igat ruutu täpselt ühe korra. Antud on ratsu algpositsioon malelaual, väljastada ratsu teekond.
Sisendi esimesel real on üks number - ruudukujulise malelaua pikkus ruutudes. Teisel real on antud ratsu algpositsioon, mis koosneb veergu tähistavast tähemärgist ning numbrist, mis tähistab rida.
Väljastada ratsu teekond kasutades sama notatsiooni. Kui teed ei leidu, väljastada `EI SAA`.

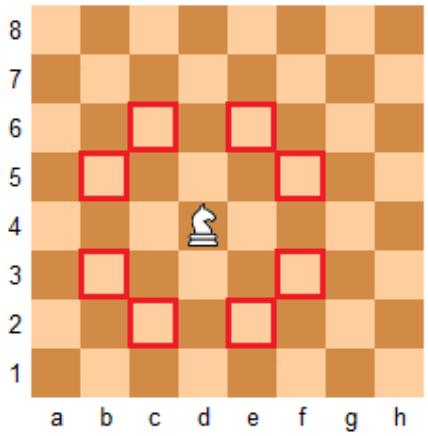
NÄIDE 1:

5
c3
Vastus:
`a2 c1 e2 d4 b5 a3 b1 d2 e4 c5 a4 b2 d1 e3 d5 b4 d3 e5 c4 a5 b3 a1 c2 e1`

NÄIDE 2:

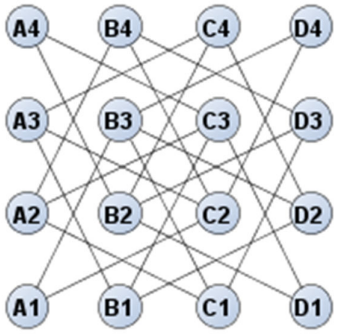
4
a1
Vastus:
`EI SAA`

Ratsu saab liikuda malelaual ratsukäikudega, st liikuda ühes suunas korraga kaks ja ristuvas suunas ühe ruudu. Malelaua keskel saab ratsu liikuda igast ruudust ühe käiguga kaheksasse teise ruutu:



Ratsu käigud. Punase raamiga on tähistatud ruudud, millele ratsu saab käia ühe käiguga.

Ratsu käike malelaual saab kujutada graafina, mille tippudeks on malelaua ruudud ning servadeks ratsu käigud. Kahe tipu vahel on serv parajasti siis, kui ratsu saab käia ühe käiguga ühele tipule vastavalt ruudult teisele tipule vastavale ruudule. Tippe on sellises graafis sama palju kui malelaual ruute.



Ratsu käigud 4X4 laual

Selles ülesandes võib graafi luua näiteks järgmiselt:

```
// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
vector<int> graaf[64];

void loo_graaf(int dim)
{
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++)
            for (int k = 0; k < 8; k++){
                int naaber_x = i + deltad[k][0];
                int naaber_y = j + deltad[k][1];
                if ((naaber_x >= 0) && (naaber_x < dim) &&
                    (naaber_y >= 0) && (naaber_y < dim))
                        graaf[i*dim + j].push_back(dim*naaber_x + naaber_y);
            }
}
```

Kuna aga ratsu võimalikud käigud on lihtsasti arvutatavad, siis on tegemist regulaarsete servadega graafiga ning graafi luua ei ole ilmtingimata vajalik. Järgnevas lahenduses arvutatakse naabrid jooksvalt.

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

const int MAX_DIM = 8;
int dim, ruutude_arv;

// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
bool kaidud[MAX_DIM][MAX_DIM] = { 0, 0 };
stack<string> vastus;
bool Otsi(int x, int y, int kaigu_nr);

int main()
{
    cin >> dim;
    ruutude_arv = dim*dim;
    string algus;
    cin >> algus;
    if (Otsi(algus[0]-'a', algus[1]-'1', 1)) {
        while (vastus.size()) {
            cout << vastus.top() << " ";
            vastus.pop();
        }
    }
    else
        cout << "EI SAA" << endl;
}
```

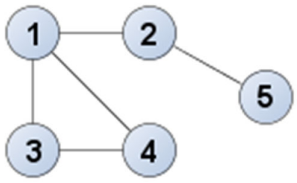
```
bool Otsi(int x, int y, int kaigu_nr)
{
    if (kaigu_nr == ruutude_arv) return true; // kõik ruudud on käidud, tee on leitud
    kaidud[x][y] = true; // siin me oleme

    for (int k = 0; k < 8; k++) { // iga võimaliku käigu jaoks
        int naaber_x = x + deltad[k][0];
        int naaber_y = y + deltad[k][1];
        if ((naaber_x < 0) || (naaber_x >= dim) || // kui on väljaspool..
            (naaber_y < 0) || (naaber_y >= dim) || //..mängulauda
            kaidud[naaber_x][naaber_y]) { // või on juba käidud,
            continue; // pole sobiv sihtkoht, jätkame
        }
        if (Otsi(naaber_x, naaber_y, kaigu_nr + 1)) { // Proovime järgmist käiku
            vastus.push(string() + (char)(naaber_x + 'a') + (char)(naaber_y + '1'));
            return true;
        }
    }
    // ei leidnud teed, mis läbiks kõik ruudud
    kaidud[x][y] = false; // võtame viimase käigu tagasi
    return false;
}
```

Selle ülesande näol on tegemist väga tuntud matemaatilise probleemiga, mille kohta võib rohkem lugeda siit: https://en.wikipedia.org/wiki/Knight's_tour

6.3.3 Laiuti läbimine

Laiuti läbimisel (*breadth-first search*, *BFS*) alustatakse ühest tipust A ja selle järel liigutakse järjekorras kõigile tipu A naabertippudele. Kui A kõik naabrid on töödeldud, liigutakse A ühe naabri kõikidele naabertippudele, seejärel teise naabri naabritele jne. See tähendab, et tipud lähevad töötlemisele nendeni jõudmise järjekorras. Seetõttu on algoritmiski hea kasutada tippude meelespidamiseks järjekorda. Järjekorra jaoks on muidugi vaja lisamälu, mis ongi üldjuhul kõige suuremaks miinuseks, võrreldes graafi sügavuti läbimisega.

Graafis



laiuti läbimisel alates tipust 1, töödeldakse järgmiseks tipud 2, 3 ja 4 (töötlemise järjekord oleneb sellest, kuidas meil servad järjestatud on) ning seejärel tipp 5. Esimesel ringil lisatakse järjekorda tipud, mis on lähtetipust ühe serva kaugusel, teisel need, mis on kahe serva kaugusel jne. Seetõttu laiuti läbimine sobib lühimate teede leidmiseks servade arvu mõttes (tee, mis läbib kõige vähem servi).

Lihtne laiuti läbimise algoritm on järgmine:

```
vector<int> graaf[tippude_arv];
bool lisatud[tippude_arv] = { 0 }; // alguses on kõik tipud lisamata

void labi_laiuti()
{
    queue<int> jarjekord;
    jarjekord.push(0); // lisame esimese tipu järjekorda
    lisatud[0] = true; // ja märgime, et on järjekorras

    while (!jarjekord.empty()) {
        int tipp = jarjekord.front();
        jarjekord.pop();
        tootle(tipp);
        for (int i = 0; i < graaf[tipp].size(); i++) {
            int naaber = graaf[tipp][i]; // iga naabertipu,
            if (!lisatud[naaber]) { // mis ei ole veel järjekorras olnud,
                jarjekord.push(naaber); // lisame järjekorda
                lisatud[naaber] = true;
            }
        }
    }
}
```

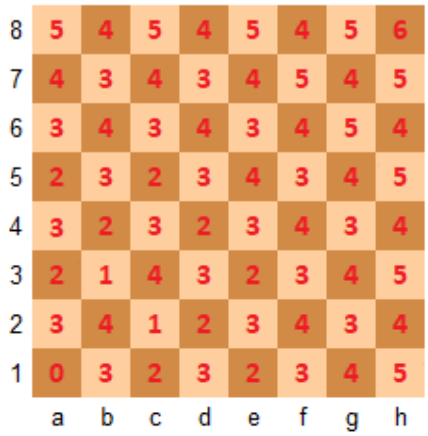
Vaatame järgmist ülesannet laiuti läbimise praktilisemast kasutusest:

6.3.4 Ülesanne: Ratsu teekond 2

Leida ratsu teekond ühelt malelaua ruudult teisele vähima võimaliku käikude arvuga. Sisendi esimesel real on üks number – malelaua ridade ja veergude arv, teisel real on kaks sõna: ratsu positsioon malelaual ja sihtkoht, väljastada üks võimalik ratsu teekond.
NÄIDE:
8
a1 h8
Vastus:
a1 c2 a3 c4 e5 g6 h8

Selle ülesande aluseks olev graaf on identne eelmise ratsu ülesandega ja seetõttu pikemat selgitust ei vaja. Kasutame siin taas selle graafi regulaarsust ning arvutame iga ruudu naabrid jooksvalt.

Vähima käikude arvuga tee leidmiseks saame kasutada laiuti läbimist. Alustades ratsu algkohast, asuvad kõik selle tipu naabrid algusest ühe käigu kaugusel. Naabrite naabrid on kahe käigu kaugusel jne:



Malelaua igas ruudus on kirjas, kui mitme käiguga ratsu sinna ruutu jõuab, kui ratsu alustab ruudult a1.

Seda, millise naabri kaudu ratsu kõige vähemate käikudega lõppu jõuab, ei ole ette teada. Selleks on kasulik meelde jätta kõige lühem tee iga ruudu jaoks, kuhu satume.

```
#include <iostream>
#include <string>
#include <stack>
#include <queue>
#include <map>
using namespace std;

const int MAX_DIM = 100;
int dim, ruutude_arv;

// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
bool kaidud[MAX_DIM][MAX_DIM] = {0, 0};
void otsi(pair<int, int> algus, pair<int, int> lopp);

int main()
{
    cin >> dim;
    string salgus, slopp;
    cin >> salgus >> slopp;
    pair<int,int> algus = make_pair(salgus[0] - 'a', salgus[1] - '1');
    pair<int, int> lopp = make_pair(slopp[0] - 'a', slopp[1] - '1');
    otsi(algus, lopp);
}
```

```
void otsi(pair<int, int> algus, pair<int, int> lopp)
{
    queue<pair<int, int>> jarjekord;
    // Siin hoiame iga ruudu jaoks meeles lühima tee, kuidas sinna algusest sai.
    map<pair<int, int>, string> teed;

    jarjekord.push(algus);
    kaidud[algus.first][algus.second] = true;
    teed[algus] = string() + char(algus.first + 'a') + char(algus.second + '1');

    while (!jarjekord.empty()) {
        pair<int, int> ruut = jarjekord.front();
        jarjekord.pop();
        if (lopp == ruut) { // oleme jõudnud sihtkohta
            cout << teed[ruut] << endl; // väljastame tee
            return; // rohkem otsida pole vaja
        }

        for (int k = 0; k < 8; k++) { // iga võimaliku käigu jaoks
            int x = ruut.first + deltad[k][0];
            int y = ruut.second + deltad[k][1];
            if ((x >= 0) && (x < dim) && (y >= 0) && (y < dim) && //mängulaual
                !kaidud[x][y]) { // ja käimata
                jarjekord.push(make_pair(x, y));
                kaidud[x][y] = true;
               // Laiuti läbimise puhul kohe kui uuele ruudule jõuame, on see lühim
               // võimalik tee servade arvu mõttes. Jätame selle tee meelde
                teed[make_pair(x, y)] =
                    teed[ruut] + " " + char(x + 'a') + char(y + '1');
            }
        }
    }
   cout << "EI SAA" << endl; // ei ole teed leidnud
}
```

6.4 SIDUSUSKOMPONENTIDE LEIDMINE

Kui nüüd vaadata hoolega eelnevaid graafi läbimise algoritme, siis need läbivad terve graafi vaid siis, kui see on sidus. Õnneks on neis kasutuses massiiv, mis peab meeles, millised tipud on töödeldud. Seega saab pärast iga otsingut käia selle massiivi läbi ja alustada uut läbimist järgmisest veel vaatlemata tipust. Selline lähenemine võimaldab ka loendada ja leida graafi sidususkomponendid. Järgmises ülesandes on justnimelt seda vaja teha.

6.4.1 Ülesanne: Ratsud

Normaalselt saab ratsu liikuda kõigi malelaua ruutude vahel. Aga praegu on seal ka hulk ettureid – nendele ruutudele ratsu käia ei saa. Leida, mitu ratsut saab malelauale paigutada nii, et nad ei saa üksteise ruutudele käia (ükskõik, mitme käiguga). Sisendi esimesel real on antud kaks arvu: `N` – ruudukujulise malelaua pikkus ruutudes, ja `E` – etturite arv . Teisel real on `E` sõna – etturite asukohad. Väljastada üks arv – ratsude arv, mida saab malelauale paigutada.
NÄIDE:
`4 5`
`a2 b2 c2 d2 d3`
Vastus:
`3`

Vaatame jällegi väiksemat, 4x4 lauda. Kui kõik ruudud on vabad, siis on tekkiv ratsu tee graaf sidus ning malelauale saab paigutada vaid ühe ratsu. Kui aga lauale on paigutatud mõned etturid, siis nendesse ruutudesse ratsu enam käia ei saa ning need võib graafist eemaldada.

Vasakpoolne graaf vastab etturiteta lauale, parempoolne aga olukorrale, kus etturid on asetatud ruutudesse A2, B2, C2, D2 ja D3. Jooniselt on hea näha, et nii tekib kolm sidususkomponenti (joonisel eri värvi tippudega) ja seega saab asetada malelauale kolm ratsut nii, et nende teed ei kattuks.

Nii on selleski ülesandes esiteks vaja tekitada korrektne graaf. Et mitte kaotada servade regulaarsust, on mugav tähistada hõivatud ruudud – kui käik viib ratsu asukoharuudult hõivatud ruudule, siis nende ruutude vahel serva ei eksisteeri. Kuna nagunii kontrollime, kas tipp on varem külastamata, siis võib antud ülesandes lihtsalt etturite asukohad märkida külastatuteks.

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

const int MAX_DIM = 100;
int dim, ruutude_arv;

// massiiv, mis on abiks arvutamisel, kuhu ratsu käia saab (st naabertippude
// leidmiseks graafis)
int deltad[8][2] =
    {{-2, -1}, {-2, 1}, {2, -1}, {2, 1}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2}};
bool kaidud[MAX_DIM][MAX_DIM] = { 0, 0 };
void otsi(int x, int y);

int main()
{
    int etturid, ratsud=0;
    cin >> dim >> etturid;
    for (int i = 0; i < etturid; i++) {
        string ettur;
        cin >> ettur;
        kaidud[ettur[0] - 'a'][ettur[1] - '1'] = true;
    }

    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++) {
            if (!kaidud[i][j]){
                ratsud++;
                otsi(i, j);
            }
        }
    cout << ratsud << endl;
    return 0;
}
```

```
void otsi(int x, int y)
{
    kaidud[x][y] = true;

    for (int k = 0; k < 8; k++) {
        int naaber_x = x + deltad[k][0];
        int naaber_y = y + deltad[k][1];
        if ((naaber_x >= 0) && (naaber_x < dim) && // mängulaual
            (naaber_y >= 0) && (naaber_y < dim) &&
            !kaidud[naaber_x][naaber_y]) { // ja käimata
                otsi(naaber_x, naaber_y);
        }
    }
}
```

6.5 Üleujutamine

Veel üheks küllalki sagedaseks graafi läbimise ülesandeks on nn üleujutamise (*flood fill*) ülesanded. Oma olemuselt on need väga sarnased sidususkomponentide leidmisele. Samas on tegemist küllaltki sageli esineva ülesandetüübiga, seega vaatame siin üht sellist lähemalt:

6.5.1 Ülesanne: Veekogud

On antud kahevärviline kaart, kus sinine tähistab vett ja roheline metsa. Leia, mitu veekogu on kaardil ja kui suur on neist suurim. Erinevateks loetakse veekogud siis, kui nende sinised pikslid ei puutu kuidagi kokku, ka nurkapidi mitte.
Sisendi esimesel real on kaks arvu m ja n ($1 \le m,\ n \le 100$), mis näitavad kaardi piksliridade ja -veergude arvu. Igal järgneval m real on n-täheline sõna, mis koosneb `S` ja `R` tähtedest. `S` tähistab sinist ja `R` rohelist pikslit. Väljastada kaks rida, millest esimesel on eraldatud veekogude arv ja teisel suurima veekogu pikslite arv.
NÄIDE:

```
5 5
RRRRR
RSSRS
RSRRS
SSSRS
SSRRR
```

Vastus:

```
2
8
```

Selleski ülesandes on tegemist regulaarsete servadega graafiga ning võib graafi hoidmiseks kasutada vahetult kahemõõtmelist märkide massiivi. Tuleb meeles pidada, et serv eksisteerib ainult sama tähemärgiga tippude vahel.

Peamiseks eesmärgiks ülesandes on leida sidususkomponent ning sageli ka selle suurus (mitut tippu sidususkomponent sisaldab). Selleks, et eristada juba vaadeldud ja vaatlemata tippe, on lisamassiivi

asemel kavalam tipud nii-öelda „üle värvida“ – siis on teada, milliseid tippe on juba arvestatud. Ühe komponendi leidmise ja üle värvimise võib lahendada järgmiselt:

```
int deltad[8][2] =
    {{1, 0}, {1, 1}, {0, 1}, {-1, 1}, {-1, 0}, {-1, -1}, {0, -1}, {1, -1}};
int ridu, veerge;
char** kaart;

int taida(int r, int v, char v1, char v2) {
    if (r < 0 || r >= ridu || v < 0 || v >= veerge)  // Kui on väljaspool kaarti
        return 0;
    if (kaart[r][v] != v1) return 0; // kui ei ole esimest värvi
    int vastus = 1; // lisab vastusele ühe
    kaart[r][v] = v2; // värvime teist värvi, et ei läheks tsüklisse
    for (int d = 0; d < 8; d++) {
        vastus += taida(r + deltad[d][0], v + deltad[d][1], v1, v2);
    }
    return vastus;
}
```

Kogu lahenduse juures peame jälle kogu graafi läbi käima ja leidma võimalikud komponentide alguskohad:

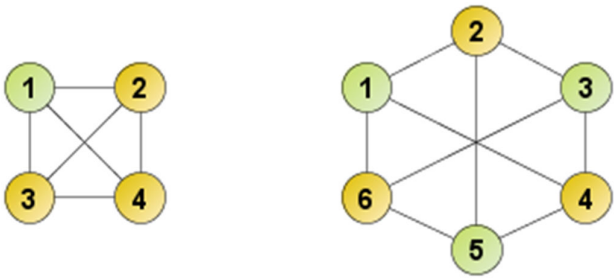
```
int main()
{
    cin >> ridu >> veerge;
    kaart = new char*[ridu];
    for (int i = 0; i < ridu; i++) {
        kaart[i] = new char[veerge];
        for (int j = 0; j < veerge; j++) {
            cin >> kaart[i][j];
        }
    }

    int veekogusid = 0;
    int suurim = 0;

    for (int i = 0; i < ridu; i++) {
        for (int j = 0; j < veerge; j++) {
            if (kaart[i][j] == 'S') {
                veekogusid++;
                suurim = max(suurim, taida(i, j, 'S', '*'));
            }
        }
    }
    cout << veekogusid << endl;
    cout << suurim << endl;
    return 0;
}
```

6.6 KAHEALUSELINE GRAAF

Kahealuseliseks (*bipartite*) graafiks nimetatakse sellist graafi, mille tipud jagunevad mingi tunnuse alusel kahte hulka ning servad saavad olla ainult erinevatesse hulkadesse kuuluvate tippude vahel.

Kõige lihtsam on ette kujutada graafi, mille tipud on värvitud kahe erineva värviga, näiteks on iga tipp värvitud kas punaseks või siniseks. Selline graaf on kahealuseline parajasti siis, kui kõik selle graafi servad on kahe erinevat värvi tipu vahel.

Graafiülesannetes tõstatub vahel küsimus, kas antud graaf saab olla kahealuseline? Oletame näiteks, et tantsuõpetaja soovib moodustada ühtlasi ringikujulisi formatsioone n lapsest nii, et iga poisi kõrval seisab tüdruk ja iga poisi vastas seisab tüdruk ning vastupidi – iga tüdruku kõrval ja vastas on poiss. Näiteks nelja last nii paigutada ei saa, kuus last aga küll:



Seda tüüpi ülesandeid võib esineda ka programmeerimisvõistlustel. Kui taibata ära, milline graaf tekib, ja see realiseerida, siis tippude „värvimine“ on taas lihtne graafi läbimise protseduur. Ka siin pole tegelikult oluline, kas läbida graafi laiuti või sügavuti – õige vastuse annavad mõlemad. Laiuti läbimine annab aga sageli kiiremini vastuolu.

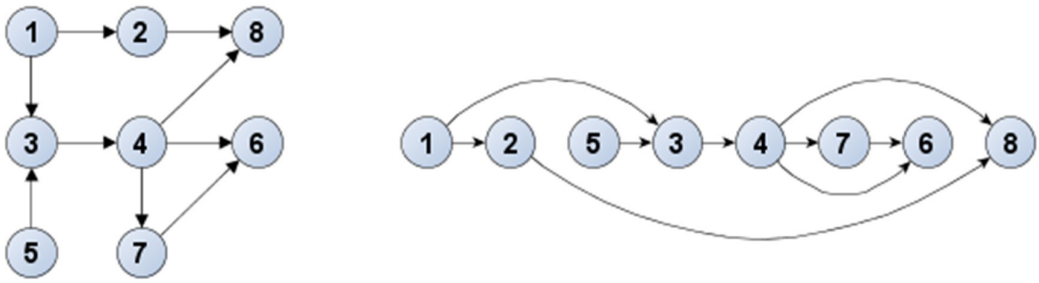
Siin on funktsioon, mis etteantud graafi tipud kaheks jaotab kasutades graafi laiuti läbimist:

```
vector<int> graaf[tippude_arv];
vector<int> jaotus(tippude_arv, -1);

bool jaota()
{
    queue<int> jrkd;
    jrkd.push(0);
    jaotus[0] = 0;

    while (!jrkd.empty()) {
        int u = jrkd.front();
        jrkd.pop();
        for (int i = 0; i < (int)graaf[u].size(); i++) {
            int v = graaf[u][i];
            if (jaotus[v] == -1) {// kui naaber ei ole veel jaotatud
                jaotus[v] = 1 - jaotus[u]; // paneme talle teise väärtuse
                jrkd.push(v); // ja lisame järjekorda
            }
            else if (jaotus[v] == jaotus[u]) { // kui serv on kahe sama omadusega tipu
                return false; // vahel, siis järelikult graaf ei ole kahealuseline
            }
       // juhul, kui naaber on juba jaotatud, aga teist värvi, on kõik korras, jätkame
        }
    }

    return true;
}
```

6.7 TOPOLOOGILINE SORTEERIMINE

Üheks huvitavaks graafi erijuhuks on suunatud tsükliteta graaf, millega tutvusime põgusalt eelmises peatükis. Sellises graafis ei saa liikuda tagasi tipule, mis on juba läbitud. Joonisel saab esitada sellise graafi lineaarse tippude reana, nii et servad on suunatud kõik ühes suunas (vasakult paremale). Sellist tippude järjestust nimetatakse **topoloogiliseks järjestuseks**.

Ühel graafil võib olla mitu topoloogilist järjestust, ainsaks tingimuseks on, et iga serva (t_i, t_j) korral $i < j$, kus i, j on tipu indeksid topoloogilises järjestuses.



Suunatud graaf ja selle topoloogiliselt sorteeritud esitus

Järgmiseks vaatame üht lihtsat topoloogilise sorteerimise ülesannet. Puhtalt topoloogilise sorteerimise ülesandeid esineb pigem lihtsamatel võistlustel, kuid sageli on topoloogiline sorteerimine mõne keerukama ülesande alamosaks.

6.7.1 Ülesanne: Loomad

Antud on hulk loomade paare, kus esimene loom on teisest tugevam. Sorteerida kõik loomad tugevuse järjekorda.
Sisendi esimesel real on loomapaaride arv `N`. Järgneval `N` real on igaühel kaks looma, esimene tugevam kui teine. Väljastada tugevuse järjekorras sorteeritud loomade nimekiri kõige tugevamast alates.
NÄIDE:

```
4
rebane jänes
karu hunt
hunt rebane
jänes hiir
```

Vastus:

```
karu hunt rebane jänes hiir
```

Taas tuleb alustada sellest, et milline graaf antud ülesandes tekib. Sisendina on antud loomapaarid, mille kohta on teada, et esimene on tugevam kui teine. Loomad on graafi tippudeks ja seos „on tugevam“ servaks – sisendi näol on meil tegemist juba servade loendiga ja nii ei ole graafi moodustamine keeruline. Kuna siin iseloomustab tippu looma nimi, siis tavalise massiivi asemel on graafi mugavam hoida andmestruktuuris, mis võimaldab kiiret ligipääsu nime järgi, näiteks `map`'is

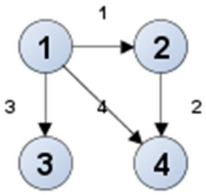
```
#include <iostream>
#include <list>
#include <queue>
#include <map>
#include <string>

using namespace std;

typedef struct Loom
{
    string Nimi;
    map<string, Loom*> Sisse;
    map<string, Loom*> Valja;
} Loom;
```

```
map<string, Loom> graaf;

Loom* leia_loom(string nimi)
{
    if (!graaf.count(nimi)) { // kui selle nimega looma veel ei ole
        Loom n;
        n.Nimi = nimi;
        graaf[nimi] = n; // lisame selle
    }
    return &graaf[nimi];
}

int main()
{
    list<string> valjund;
    queue<Loom*> jrkd;
    int servi;
    cin >> servi;
    for (int i = 0; i < servi; i++) {
        string yks, kaks;
        cin >> yks >> kaks;
        Loom* esimene = leia_loom(yks);
        Loom* teine = leia_loom(kaks);
        esimene->Valja[teine->Nimi] = teine;
        teine->Sisse[esimene->Nimi] = esimene;
    }
    // kõigepealt leiame kõik tipud, millel sissetulevaid kaari ei ole. Need loomad
    // saavad olla kõige tugevamad teadaoleva info põhjal.
    for (map<string, Loom>::iterator it = graaf.begin(); it != graaf.end(); ++it) {
        if (it->second.Sisse.empty())
            jrkd.push(&it->second);
    }

    while (!jrkd.empty()) { // kuni on veel töötlemata tippe
        Loom* n = jrkd.front();
        jrkd.pop();
        valjund.push_back(n->Nimi); // lisame vastusesse

        for (map<string, Loom*>::iterator it2 = n->Valja.begin();
             it2 != n->Valja.end(); ++it2) {
            Loom* jrgm = it2->second;
            // eemaldame naabrilt kõik vaadeldavast tipust sissetulevad kaared
            jrgm->Sisse.erase(n->Nimi);
            if (jrgm->Sisse.empty()) // kui naabril rohkem sissetulevaid kaari pole
                jrkd.push(jrgm); // lisame selle järjekorda
        }
        n->Valja.clear(); // eemaldame tipu kõik väljuvad kaared
    }

    for (map<string, Loom>::iterator it = graaf.begin(); it != graaf.end(); ++it) {
        if (it->second.Sisse.empty()) continue;
        cout << "EI SAA" << endl;
        return 0;
    }

    for (list<string>::iterator it = valjund.begin(); it != valjund.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
    return 0;
}
```

Topoloogilist sorteerimist nõuavad harilikult igasugu planeerimisülesanded, kus objektidel on omavahel range järjestus – näiteks tuleb mingid tööd sooritada enne teisi.

6.8 KAALUTUD SERVADEGA GRAAFID

Programmeerimise seisukohalt on oluline eristada ka kaalutud graafe. Kaalutud graafis on igal serval kaal ehk väärtus. Oluline erinevus on näiteks lühima tee leidmisel graafis – kui kõik servad on võrdsed, taandub lühima tee leidmine vähima servade arvuga tee leidmisele, kaalutud graafis tuleb kasutada keerulisemaid algoritme.



Sellel graafil lühim tee tipust 1 tippu 4 läbi tipu 2, kuna 1+2 < 4.

6.9 DIJKSTRA LÜHIMA TEE LEIDMISE ALGORITM

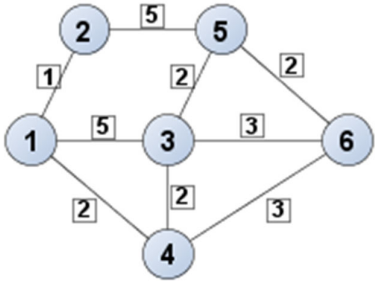
1956. aastal leiutas hollandi arvutiteadlane Edsger Dijkstra (1930-2002) efektiivse algoritmi, kuidas leida lühim tee graafi ühest tipust teise.

Dijkstra algoritmi võib vaadelda kui graafi laiuti läbimise edasiarendust. Laiuti läbimise puhul on meil järjekord vaadeldavatest tippudest, kus kõigepealt tulevad tipud, mis on algtipust kaugusel 1, siis kaugusel 2 jne. Kui kõigi servade kaal on 1, siis on Dijkstra algoritm ekvivalentne tavalise laiuti läbimisega. Kui servadel võivad olla ka teiste väärtustega kaalud, muutub algoritm järgmiseks:

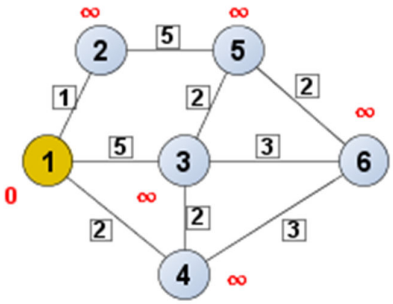
Dijkstra puhul tekitab tema tööst arusaamisest suurema hulgal segadust see, kuidas tema perekonnanime hääldama peaks. Nime esimeses silbis on diftong, mille esimene pool on a, e ja ä vahepealne ning teine pool on i. Kuula näidet siit: https://upload.wikimedia.org/wikipedia/commons/8/85/Dijkstra.ogg

1. Sea algtipu kauguseks 0 ja kõigi teiste tippude kauguseks lõpmatus.
2. Märgi algtipp **aktiivseks** ja kõik ülejäänud tipud **vaatamata** tippudeks.
3. Olgu aktiivne tipp A. Vaata kõiki tema naabreid B ja:
    a. Arvuta B-sse jõudmise kaugus A kaudu (kaugus tippu A pluss A ja B vahelise serva kaal).
    b. Kui see kaugus on väiksem kui seni leitud vähim kaugus tippu B, siis paranda tipu B kaugus selleks väiksemaks väärtuseks.
4. Märgi tipp A **vaadatuks.** (Siia pole vaja enam tagasi tulla!)
5. Vali vaatamata tippude seast **vähima senise kaugusega** tipp uueks aktiivseks tipuks.
    a. Kui kõigi vaatamata tippude kaugus on lõpmatus, pole nõutud teed võimalik leida.
    b. Kui uus aktiivne tipp on otsitav sihtkoht, siis on lühim tee leitud – see on võrdne parima seni leitud kaugusega.
    c. Vastasel korral jätka uuesti alates sammust 3.

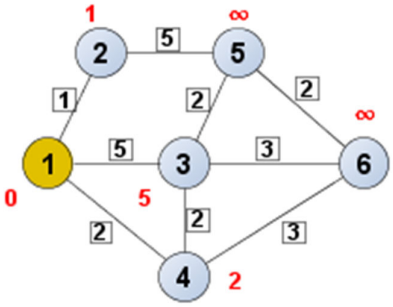
Vaatame, kuidas leida järgneval graafil lühim tee tipust 1 tippu 6:



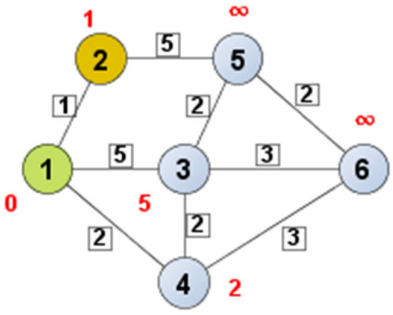
Kõigepealt märgime algtipu kauguseks 0 ja teiste tippude kauguseks lõpmatuse (joonisel punasega) ning seame tipu 1 aktiivseks (joonisel kollane):



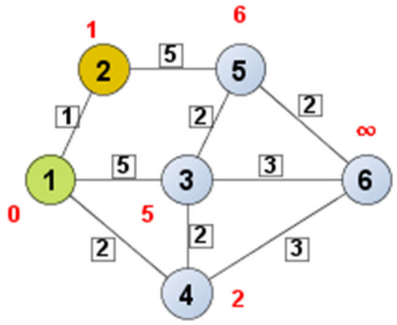
Leiame kõigi tipu 1 naabertippude kaugused algtipust: tipu 2 kaugus on 0 + 1 = 1, tipu 3 kauguseks saab 5 ja tipu 4 kauguseks 2:



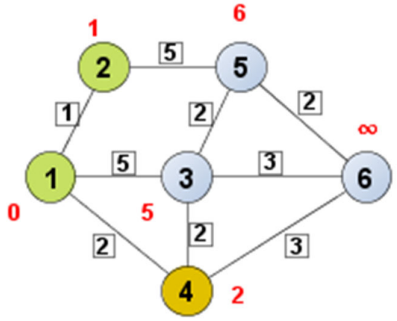
Nüüd on tipp 1 ja selle märgime vaadatuks (joonisel rohelisega), et seda uuesti mitte vaadelda. Valime mitteaktiivsetest vaatlemata tippudest kõige väiksema kaugusega tipu, milleks on tipp 2 kaugusega 1. Märgime selle aktiivseks:

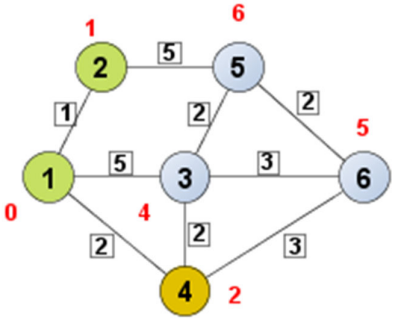


Vaatame tipu 2 naabreid: 1 on juba vaadatud ja sellega ei tee midagi. Leiame tipu 5 kauguse, milleks saab 1 + 5 = 6:

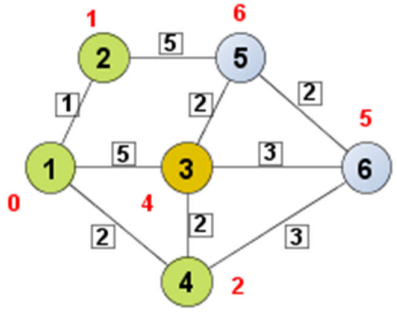


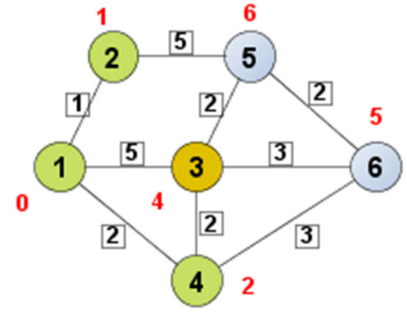
Nüüd on ka tipp 2 töödeldud ja valime uue vähima kaugusega tipu, milleks nüüd on tipp 4 ja arvutame selle kaudu kaugused. Kuigi tipul 3 on juba leitud kaugus, milleks on 5, on tipu 4 kaudu kaugus tippu 3 2 + 2 = 4. Parandame tipu 3 kauguse ning leiame ka lõpptipu 6 kauguse:



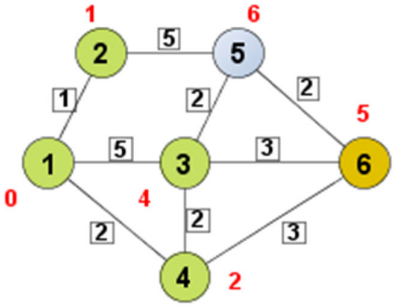


Kuigi oleme leidnud mingi kauguse lõpptippu, ei pruugi see olla veel lõplik vastus, seega jätkame valides uueks aktiivseks tipuks tipu 3 kaugusega 4. Siitkaudu kaugused siiski ei parane:





Nüüd on vähima kaugusega vaatlemata tipuks tipp 6 kaugusega 5. Märgime selle aktiivseks:



Kuna tegemist on lõpptipuga, siis oleme leidnud lühima teepikkuse tipust 1 tippu 6 ja selleks on 5. Tippu 5 meil enam vaadata ei ole vaja, sealtkaudu enam teepikkus lõpp-punkti parandada ei ole võimalik.

Dijkstra algoritmi kirjelduses on üks koht, mille realisatsioon pole täiesti ilmne: see on vähima kaugusega tipu leidmine. Üks võimalus on kõik vaatamata tipud ükshaaval läbi käia, mis võtab $O(T)$ aega, kus T on graafi tippude koguarv. Algoritmi kogukeerukuseks tuleb siis $O(S + T^2)$, kus S on graafi servade arv. Seda kasutas Dijkstra ka oma esialgses töös.

Parem võimalus on aga kasutada andmestruktuuri, mis võimaldab pidevalt vähima väärtusega elementi kiiresti kätte saada, näiteks peatükist 3.13 tuttaval kuhjal põhinevat eelistusjärjekorda. Selle abil saab senise vähima kauguse leida ajaga $log(T)$, nii et algoritmi kogukeerukuseks tuleb $O(S + T \cdot log(T))$.

Algoritmi näitlikustamiseks vaatame järgmist ülesannet:

6.9.1 Ülesanne: sõiduaeg

`N` asulat on omavahel ühendatud `M` teega, kuid mõned neist teedest on head kiirteed, teised aga peaaegu läbimatud kruusateed. Seetõttu on vahel otsetee asemel kiirem minna teiste asulate kaudu ringi. Leida kiireim tee asulast `A` asulasse `B`.
Sisendi esimesel real on asulate arv `N`, teede arv `M`, alguspunkti indeks `A` ja sihtkoha indeks `B` (`0≤A,B<N`). Igal järgneval `M` real on kolm arvu: ühendatud asulate indeksid ja esimesest asulast teise jõudmise aeg minutites.
Leida, mitu minutit kulub selleks, et jõuda asulast `A` asulasse `B`. Kui selline teekond on võimatu, väljastada `-1`.

NÄIDE:

```
6 9 1 6
1 2 1
1 3 5
1 4 2
2 5 5
3 5 2
3 6 3
3 4 2
4 6 3
5 6 2
Vastus:
5
```

See on klassikaline lühima tee pikkuse leidmise algoritm. Graafi tippudeks on asulad, servadeks teed nende vahel ning kaaludeks minutid, kui kaua tee läbimiseks kulub. Kuna kuluv aeg ei saa olla negatiivne, siis saab lahendamiseks kasutada eespool põhjalikult kirjeldatud Dijkstra algoritmi:

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;
#define INF INT_MAX // Lõpmatus

const int tippude_arv = 10001; // Suurim võimalik tippude arv.
vector<pair<int, int> > Naabrid[tippude_arv];
int Min_kaugused[tippude_arv];// Siin hoiame vähimat leitud kaugust algtipust tippu i.
bool Vaadatud[tippude_arv] = { 0 }; // Siin hoiame infot, kas tipp on juba töödeldud

int leia_teepikkus(int start, int siht, int n)
{
    for (int i = 0; i < tippude_arv; i++)
        Min_kaugused[i] = INF; // alguses oletame, et kõik tipud on lõpmata kaugel
    class greater { public: bool operator ()(pair<int, int>&p1, pair<int, int>&p2)
        { return p1.second>p2.second; } };
    priority_queue<pair<int, int>, vector<pair< int, int >>, greater > kaugused;
    // Lähtekoha kaugus iseendast on 0
    kaugused.push(make_pair(start, Min_kaugused[start] = 0));
    while (!kaugused.empty())
    {
        pair<int, int> lahim_tipp = kaugused.top();
        kaugused.pop();
        int tipp = lahim_tipp.first, vahim_kaugus = lahim_tipp.second;
        if(tipp == siht) { // otsitud tee on leitud
            return vahim_kaugus; //tagastame tee pikkuse
        }
        if (Vaadatud[tipp]) // tipp vaadatud, võtame järgmise
            continue;
        Vaadatud[tipp] = true; // Siin me oleme
        for (int i = 0; i < Naabrid[tipp].size(); i++) // Iga naabri jaoks
            if (!Vaadatud[Naabrid[tipp][i].first] && // kui naaber on veel käimata
                Naabrid[tipp][i].second + vahim_kaugus <
                Min_kaugused[Naabrid[tipp][i].first])
                kaugused.push(make_pair(Naabrid[tipp][i].first,
                 (Min_kaugused[Naabrid[tipp][i].first] =
                     Naabrid[tipp][i].second + vahim_kaugus)));
    }
    return -1; //ei leidnud teed
}

int main()
{
    int n, m, a, b, x, y, t;
    cin >> n >> m >> a >> b;

    for (int i = 0; i < m; i++) { // Koostame graafi
        cin >> x >> y >> t;
        Naabrid[x].push_back(make_pair(y, t));
        Naabrid[y].push_back(make_pair(x, t));
    }

    std::cout << leia_teepikkus(a, b, n);
    return 0;
}
```

6.10 PUUD

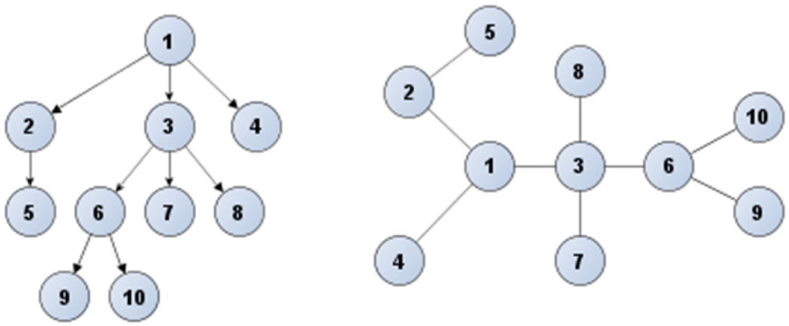
6.10.1 Puud kui graafid

Kolmandas peatükis alapeatükis „3.10 Kahendpuu“ oli juttu puudest andmestruktuurina, lähemalt just kahendpuust ja selle esitusest.

Kahendpuud (eriti kompaktset kahendpuud) on lihtne hoida massiivis, aga kui tegu on üldistatud puuga, mille tippudel võib olla kuitahes palju alluvaid, on kasulikum mõelda sellest kui graafi erijuhust. Puu hoidmiseks on tavaliselt kõige loomulikum esitusviis tippude loend.

6.10.2 Puu definitsioon

Pilti vaadates on üsna selge, et tegu on puuga:



Vasakul ja paremal on esitatud tegelikult sama puu. Vasakpoolsel joonise on tipp 1 välja toodud **juurtipuna** ja servad on suunatud juurest lehtedeni, see aitab algoritmi sageli paremini korrastada, kuid see pole kohustuslik. Parempoolsel joonisel olev graaf on samuti puu, kuigi see pole otseselt nii joonistatud.

Formaalselt saab seda, et *n* tipuga graaf *G* on puu, sõnastada mitmel viisil:

- *G* on sidus ja tsükliteta;
- *G* on tsükliteta ja mistahes uue serva lisamine loob graafile tsükli;
- *G* on sidus, kuid mistahes serva eemaldamine muudab selle mittesidusaks;
- Suvaline tipupaar graafis *G* on ühendatav lihtahelaga täpselt ühel viisil;
- *G* on sidus ja selles on *n* – 1 serva;
- *G* on tsükliteta ja selles on *n* – 1 serva.

Ülesannete sõnastuses on sageli kasutatud üht neist omadustest selle asemel, et lihtsalt öelda „tegemist on puuga“. Näiteks võib ülesandes olla juttu teedevõrgust, kus „igast linnast igasse linna saab liikuda unikaalsel viisil“, mille all on tegelikult mõeldud lihtsalt seda, et teedevõrk moodustab puu.

6.10.3 Graafitöötlusalgoritmid puul

Kuna puu on graaf, saab sellel kasutada ka graafitöötlusalgoritme. Samas puu küllaltki range struktuur võimaldab kasutada lihtsamaid lahendusi.

Kuna T tipuga puus on $T - 1$ serva, siis graafitöötlusalgorimid keerukusega $O(T + S)$ on puude puhul keerukusega $O(T)$. Realiseerides graafitöötlusalgoritme puudel tasub meeles pidada, et puudes ei esine tsükleid ja puu on sidus - seega ei ole põhjust puust tsükleid ega sidususkomponente

otsida. Samuti on puus iga kahe tipu vahel üheselt määratud tee, mis teeb lühima tee leidmise oluliselt lihtsamaks – triviaalseks puu läbimise ülesandeks. Samuti on puu kahealuseline graaf.

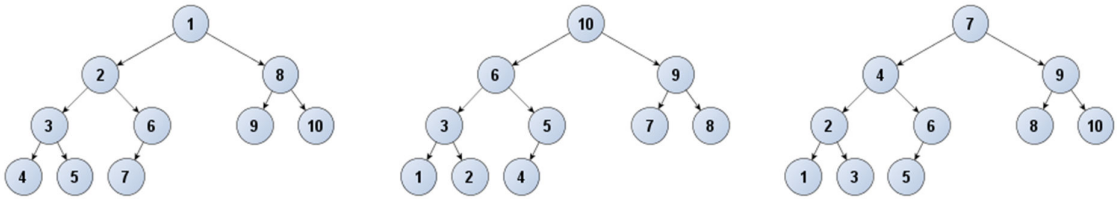
6.10.4 Kahendpuu sügavuti läbimine

Kui üldjuhul võib graafi läbimist alustada ükskõik millisest tipust, siis juurtega puudel on loomulik alguspunkt – puu juurtipp. Kahendpuus on määratud ka alluvate järjekord (eristatakse vasakut ja paremat alluvat), mistõttu saab eristada spetsiifilisemaid sügavuti läbimise võimalusi, kusjuures iga läbimise viis annab üheselt määratud tippude töötlemise järjekorra. Kahendpuu sügavuti läbimise viisid on järgmised:

Eesjärjestuses läbimise (*pre-order traversal*) korral töödeldakse kõigepealt tipp, seejärel selle vasak alampuu ja seejärel parem alampuu.

Lõppjärjestuses läbimise (*post-order traversal*) korral töödeldakse kõigepealt vasakpoolne alampuu, seejärel parempoolne alampuu ja alles siis tipp ise.

Keskjärjestuses (*in-order*) läbimise korral töödeldakse kõigepealt vasak alampuu, seejärel tipp ise ja siis parem alampuu.



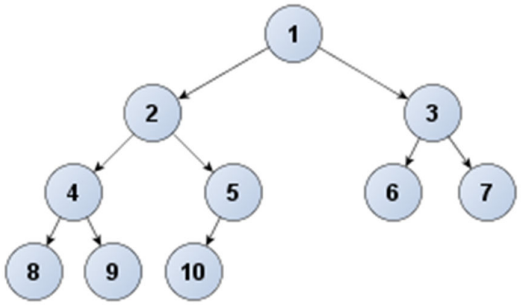
Puu läbimine ees-, lõpp- ja keskjärjestuses. Numbrid tippudes tähistavad läbimise järjekorda.

Programmi kirjutamise seisukohalt ongi ainus erinevus selles, kus asub funktsioon tipu töötlemiseks:

```
void labi_eesjarjestuses(int tipp)
{
    int vasak = puu[2*tipp + 1];
    int parem = vasak + 1;
    tootle(tipp);
    labi_eesjarjestuses(vasak);
    labi_eesjarjestuses(parem);
}

void labi_keskjarjestuses(int tipp)
{
    int vasak = puu[2 * tipp + 1];
    int parem = vasak + 1;
    labi_keskjarjestuses(vasak);
    tootle(tipp);
    labi_keskjarjestuses(parem);
}

void labi_loppjarjestuses(int tipp)
{
    int vasak = puu[2 * tipp + 1];
    int parem = vasak + 1;
    labi_loppjarjestuses(vasak);
    labi_loppjarjestuses(parem);
    tootle(tipp);
}
```

6.10.5 Puu laiuti läbimine

Ka puud saab läbida laiuti, mis tähendab tippude tasemete kaupa töötlemist – kõigepealt juurtipp, seejärel teise taseme tipud, siis kolmanda jne. Kõige viimasena töödeldakse puu lehed. Kui alluvad on järjestatud (nagu näiteks kahendpuul) on laiuti läbimise järjekord üheselt määratud:



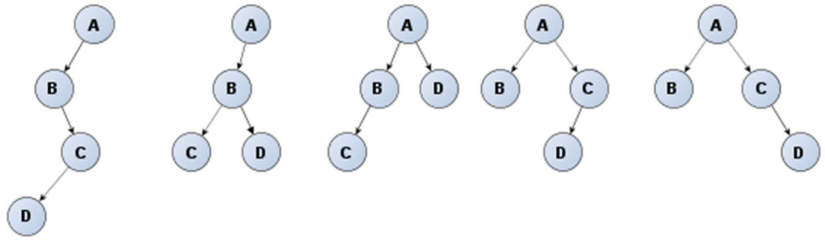
Kahendpuu tippude läbimise järjekord puu laiuti läbimisel

Kui kahendpuu on esitatud massiivina, nagu alampeatükis 3.10.1 kirjeldatud, siis sellise kahendpuu laiuti läbimine tähendab lihtsalt massiivi läbimist algusest lõpuni.

6.10.6 Ülesanne: Kahendpuud

Hannes magas hommikul sisse ja jõudis loengusse üsna lõpupoole. Loengu teemaks oli kahendpuude läbimine ja tahvlile oli joonistatud hulk erinevaid kahendpuid, mille tipud olid eristatud erinevate tähtedega. Kuna parajasti oli käsil puude erinevad läbimisviisid, siis Hannes ei viitsinud puid kohe üles joonistada, vaid kirjutas ruttu üles tekstistringi, mis saadi puud eesjärjestuses läbides. Õnneks Hannes taipas, et eesjärjestusest ilmselt ei piisa puude taastamiseks, ja lisas iga puu kohta ka selle keskjärjestuses läbides saadud järjestuse. Kodus asus ta puid taastama, kuid käsitsi oli see siiski tüütu ettevõtmine. Kuna kodutööks jäi nagunii leida nende puude tippude läbimise järjekord lõppjärjestuses, siis aita kirjutada Hannesel programm, mis leiab etteantud ees ja keskjärjestuse järgi antud puu läbimise lõppjärjestuses.
Sisendi esimesel real on puu tippude arv ja teisel real kaks stringi- antud puu tippude töötlemise järjekord eesjärjestuses ja keskjärjestuses. Väljastada üks string: antud puu tippude töötlemise järjekord lõppjärjestuses.
NÄIDE:
4
ABCD BACD
Vastus:
BDCA

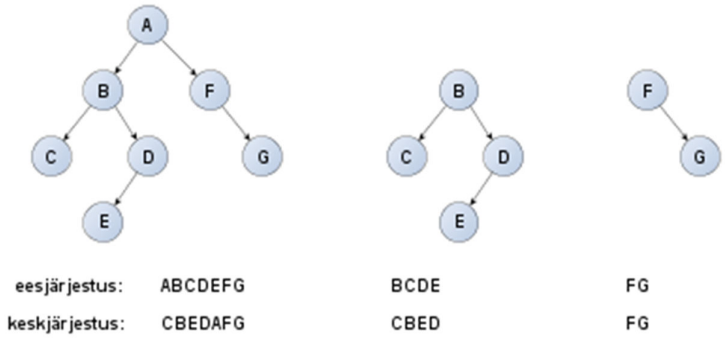
Tõesti, Hannesel on õigus. Kuigi konkreetse kahendpuu tippude töötlemise järjekord on puu eesjärjestuses (nagu ka kesk ja lõppjärjestuses) läbimisel üheselt määratud, ei kehti vastupidine: mitmel erineval kahendpuul võib olla sama tippude töötlemise järjekord.



Erinevaid kahendpuid, mille tippude töötlemise järjekord eesjärjestuses on ABCD

Keskjärjestus annab olulist lisainformatsiooni. Näiteks ülaltoodud graafide keskjärjestuses läbimise tulemused on BDCA, CBDA, CBAD, BADC, BACD, mis kõik on erinevad.

Eesjärjestuses töötlemisel töödeldakse esimesena alati puu juur, nii on juureks see tipp, mis esimeses stringis esimene. Juure leidmine on seega lihtne. Aga teise tipuga jääb juba hätta, kuna see võib olla nii parem kui ka vasak alluv. Keskjärjestuses läbimisel töödeldakse juurtipp vahetult pärast vasaku alampuu töötlemist ja enne parema alampuu töötlemist. Nii jagades keskjärjestuses saadud stringi juure kohalt kaheks, tekib kaks stringi: vasakpoolse alampuu keskjärjestuses läbimise järjekord ja parempoolse alampuu keskjärjestuses läbimise järjekord. Kuna ka eesjärjestuses läbitakse enne parema alampuu kallale asumist vasak, on meil olemas kogu info juure kummagi alampuu ees- ja keskjärjestuse kohta.



Vasakul on puu juurega A, keskel selle vasakpoolne alampuu ja paremal parempoolne alampuu

Nii võiks taastamise algoritm olla midagi sellist:

```
taasta_puu(eesjärjestus, keskjärjestus) {
    leia_alampuude_järjestused();
    taasta_puu(vasaku_alampuu_eesjärjestus, vasaku_alampuu_keskjärjestus);
    taasta_puu(parema_alampuu_eesjärjestus, parema_alampuu_keskjärjestus);
}
```

Jääb veel leida lõppjärjestus. Üheks võimaluseks on muidugi puu reaalselt tekitada ning seejärel läbida see lõppjärjestuses. Kindlasti ei ole see paha mõte, kuid saab veel kavalamini. `Taasta_puu` funktsiooni ülesehitus on sarnane kahendpuu sügavuti läbimise algoritmidele ning lõppjärjestuse saamiseks ei pea tegema midagi muud, kui funktsiooni lõpus parasjagu töötlemisel oleva tipu nimi vastusele lisada või miks mitte ka kohe väljastada.

Samuti võib loobuda stringide tegelikust tükeldamisest ja leida ainult alg- ja lõppindeksi, mille vahel olevale lõigule vastavat alampuud töötlema hakata. Kuna järjestused leitakse enne alampuude töötlema asumist ehk eesjärjestuses (kõigepealt leiame juurtipu, seejärel selle vasaku alampuu juure, mille järel omakorda selle vasaku alampuu juure jne), siis eesjärjestuse stringis piisab ainult jooksva indeksi meelespidamisest.

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

int ees_indeks;
string eesjarjestus, keskjarjestus;

void taasta_puu(int algus, int lopp) {
    if (algus>lopp) {
        return;
    }
    int i, kesk_indeks;

    char nimi = eesjarjestus[ees_indeks++];
    for (i = algus; i <= lopp; i++) {
        if (keskjarjestus[i] == nimi)
            kesk_indeks = i;
    }
    taasta_puu(algus, kesk_indeks - 1);
    taasta_puu(kesk_indeks + 1, lopp);
    cout << nimi;
    return;
}

int main() {

    int tippe;
    cin >> tippe;
    cin >> eesjarjestus >> keskjarjestus;
    ees_indeks = 0;
    taasta_puu(0, tippe - 1);
    cout << endl;

    return 0;
}
```

6.11 KONTROLLÜLESANDED

6.11.1 Robotivõistlus

Tänapäeval muutuvad järjest populaarsemaks mitmesugused robotivõistlused. Selles ülesandes on robotil vaja ristkülikukujulisel võistlusväljakul läbida hulk kontrollpunkte. Selleks on tal instruktsioon, mis koosneb märkidest 'V' (pööra 90 kraadi vasakule), 'P' (pööra 90 kraadi paremale) ja 'O' (liigu otse). Võistlusväljakul on ka takistused, millele robot sõita ei saa. Leida, mitu erinevat kontrollpunkti jõuab robot läbida.

Sisendi esimesel real on kolm täisarvu: `N` ja `M` ($1 \le N, M \le 100$) tähistavad väljaku mõõtmeid, `S` ($1 \le S \le 10\ 000$) tähistab instruktsiooni pikkust.

Järgmisel `N` real on igaühel `M` märgist koosnev string, kus:

- '.' tähistab tühja ruutu,
- '*' tähistab kontrollpunkti,
- '#' tähistab takistust,
- 'N', 'S', 'W' või 'E' tähistab roboti algset asukohta, kus konkreetne märk tähistab seda, millises suunas robot alguses liikuma hakkaks.

Viimasel real on `S` märgist koosne string, milles on ainult märgid 'V', 'P' ja 'O'.

Väljundisse kirjutada täpselt üks täisarv: mitu erinevat kontrollpunkti robot läbib.

NÄIDE 1:

```
3 3 2
***
*N*
***
PV
```

Vastus:

0 (robot keerutab koha peal ega liigu kusagile)

NÄIDE 2:

```
4 4 5
...#
*#W.
*.*.
*.#.
OOVOO
```

Vastus:

1

NÄIDE 3:

```
10 10 20
....*.....
.......*..
.....*....
..*.#.....
...#N.*..*
...*......
..........
..........
..........
..........
OPOOOOOOVVOOOOOOVOPO
```

Vastus:

3

6.11.2 Civilization

Arvutimängus Civilization võistlevad erinevad riigid omavahel selle pärast, kes saab maailmas edukaimaks. Mängus koosneb maailma kaart hulgast kontinentidest ja saartest ning sageli juhtub, et erinevad riigid saavad erinevatel kontinentidel domineerivaks ja haaravad endale kõik selle kontinendi ressursid. Et seejärel erinevate riikide võimsust hinnata, on kasulik teada, kui suured erinevad kontinendid on. Ülesandeks on leida maailma kaardi põhjal suurima kontinendi pindala.
Sisendi esimesel real on kaks arvu `M` ja `N` (`1≤M, N≤1000`), mis tähistavad maailmakaardi suurust. Järgmisel `M` real on igaühel `N` märgist koosnev string, mis koosneb kas 'v' (vesi) või 'm' (maa) märkidest. Kaks märki on osa samast kontinendist, kui neil on ühine külg. Kaart moodustab „silindri", s.t läänepoolne ja idapoolne serv puutuvad tegelikult omavahel kokku. Väljundisse kirjutada üks täisarv: suurima kontinendi pindala.

NÄIDE:

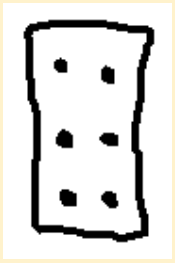
```
5 6
vvvvvv
vmmmvv
vvvvvv
mmvvmm
vvvvvv
```

Vastus:

4

(eelviimasel real on ainult üks kontinent, kuigi ta on kaardil kahes osas).

6.11.3 Doominod



Väike Martin joonistab graafikaprogrammis doominoklotse, näiteks selliseid nagu näha juuresoleval pildil.
On teada, et iga joonis on piiratud pideva joonega. Joone sees olevad silmad võivad olla ebakorrapärase kujuga, kuid on kõik ühes tükis ning ei puutu omavahel kokku.
Kirjutada programm, mis etteantud joonise põhjal leiab, mitu silma klotsil on.
Sisendi esimesel real on kaks täisarvu: `H (1≤H≤100)` ja `W (1≤W≤50)`. Järgmisel `H` real on igaühel string pikkusega `W`, mis koosneb märkidest `0-9` ja `A-F`. Neid märke tuleb tõlgendada 16-süsteemi arvudena ja teisendada bittideks (nt `A=1010`). Bitid väärtusega `1` tähistavad musti piksleid ja bitid väärtusega `0` tähistavad valgeid.
Väljundisse kirjutada täpselt üks arv: mitu silma on doominoklotsil.

NÄIDE:

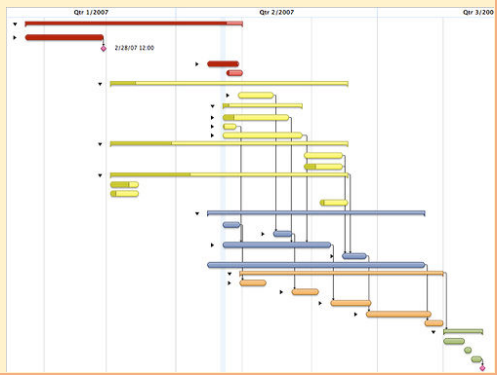
```
8 2
7C
86
92
8A
A2
AA
82
7E
```

Vastus:

3

6.11.4 Projektiplaan

Mitmesugused projektid, olgu nad siis ehituse, tarkvara või laulupeokorralduse valdkonnast, koosnevad paljudest tegevustest, mis tuleb ära teha teatud järjekorras. Mõnesid tegevusi saab teha paralleelselt, kuid sageli on need tegevused omavahelises sõltuvuses, s.t üks asi tuleb enne ära teha kui teise juurde saab asuda. Tavaline küsimus, mida suured ülemused projektijuhilt küsivad, on: „millal valmis saab?“ Eeldusel, et üksteisest mittesõltuvaid tegevusi võib teha paralleelselt, leida projekti minimaalne kestus.
Sisendi esimesel real on kaks täisarvu: tegevuste arv `N (1≤N≤1000)` ja sõltuvuste arv `M (1≤M≤10000)`
Teisel real on `N` positiivset täisarvu, mis tähistavad üksikutele tegevustele kuluvat aega.
Lõpuks tuleb `M` rida, igaühel kaks täisarvu, mis tähistavad tegevuste järjenumbreid: esimene tegevus peab lõppema enne, kui teine algab. Väljundisse kirjutada üks täisarv: projekti minimaalne kestus.



NÄIDE:

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

Vastus:

```
15
```

6.11.5 Sõnateisendused

Lapsed mängivad sõnamängu, kus ühest sõnast tuleb sammhaaval saada teine, kusjuures igal sammul tohib muuta ära ühe tähe. Näiteks võime teha teisenduse `kala→pala→palk→pauk→puuk`.
Kui antud on lubatud sõnade hulk, algne sõna ja lõpuks saadav sõna, siis leida, mitu sammu on vastavaks teisenduseks tarvis teha. On teada, et teisendamine on alati võimalik.
Sisendi esimesel real on sõnade arv `N (1≤N≤200)`. Järgmisel `N` real on igaühel üks sõna. Sisendi viimasel real on kaks sõna, mille vahelist teisendust tuleb mõõta.

NÄIDE:

```
8
kada
kaja
kala
pada
paha
raba
raha
sada
kala raha
```

Vastus:

```
4
```

6.11.6 Kahevärviprobleem

Nagu peatüki sissejuhatuses kirjutatud, on üks graafiteooria tuntumaid probleeme neljavärviteoreem, kus uuritakse, kas iga kaart on värvitav nelja värviga. Teoreem on tuntud ka selle poolest, et selle tõestamiseks kasutati arvutit. Siin ülesandes vaadeldakse sarnast, aga palju lihtsamat küsimust: kas etteantud kaarti on võimalik värvida kahe erineva värviga?
Sisendi esimesel real on kaks täisarvu: riikide arv `N` (`1≤N≤200`) ning piiride arv `M`. Järgmisel `M` real on igaühel kaks täisarvu, mis näitavad, et neil kahel riigil on ühine piir. Väljundisse kirjutada `JAH`, kui kaart on võimalik värvida ainult kahe värviga ning `EI`, kui see ei ole võimalik.

NÄIDE 1:

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

Vastus:

```
EI
```

NÄIDE 2:

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

Vastus:

```
JAH
```

NÄIDE 3:

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

Vastus:

```
JAH
```

6.11.7 Joonejälgija robot

Robotexil ja teistel sarnastel võistlustel on üheks võistlusalaks sageli sellise roboti programmeerimine, mis järgib maha joonistatud joont. Antud ülesandes on maha joonistatud terve ruudustik, millel sõidab ümmarguse kujuga Roomba-suurune robot (diameeter 35 cm). Robot oskab sõita ainult nii, et jälgitav joon läbib kogu aeg roboti keskkohta. Jooned on üksteisest 20 cm kaugusel, nii et kõrvalolevate joonteni jääb alati pisut ruumi. Robotile on võimalik anda järgmisi käske: 20 cm edasi, 40 cm edasi, 60 cm edasi, 90° vasakule, 90° paremale. Iga käsu täitmine võtab ühe sekundi. Järgmiseks on robotile ehitatud takistusrada, kus ruudustikule joonte vahele on paigutatud kuubikud, mida robot peab vältima. Ülesandeks on leida kiireim tee, kuidas robot saab jõuda ühest etteantud punktist teise.

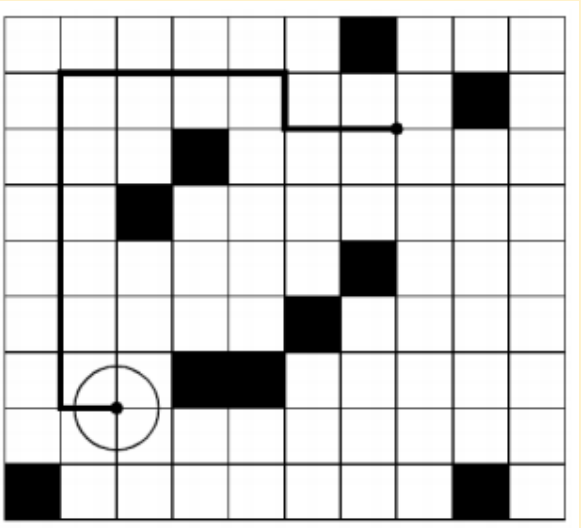
Lihtsuse mõttes kasutame edaspidi ühikuna ruutude enda suurust. Sisendi esimesel real on joonte vahele jääva ruudustiku pikkus `M (1≤M≤50)` ja laius `N (1≤N≤50)`. Järgmisel `M` real on igaühel `N` arvu väärtustega kas `0` või `1`. `0` tähistab, et ruut on tühi, `1` tähistab takistust.

Viimasel real on 4 täisarvu ja üks tähemärk, mis tähistavad roboti alguspunkti koordinaate, sihtkoha koordinaate ja roboti algset suunda. Koordinaadid algavad ruudustiku vasakust ülemisest nurgast. Roboti suund on üks märkidest `N`, `E`, `S` või `W`.

Väljundisse kirjutada, mitme sekundiga on robotil võimalik lõpp-punkti jõuda. Kui soovitud teekond pole võimalik, väljastada `-1`.

NÄIDE:

```
9 10
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 0
7 2 2 7 S
```



Vastus:

12 (vt joonist)

6.11.8 Numbriruudud

Ruudustik on täidetud numbritega 0-9 (vt näidet joonisel). Vasakus ülemises ruudus on mängunupp, eesmärgiks on jõuda paremasse alumisse ruutu. Igal käigul võib nupuga astuda sammu vasakule, paremale, üles või alla, aga iga kord liidetakse sinu skoorile vastavas ruudus oleva numbri väärtus. Eesmärgiks on jõuda lõppu võimalikult väikese skooriga.
Sisendi esimesel real on ruudustiku kõrgus `M` ja laius `N` (`1≤M, N≤999`). Järgmisel `M` real on igaühel `N` numbrit, millest moodustubki ruudustik. Väljundisse kirjutada üks täisarv: minimaalne skoor, millega on võimalik algusest lõppu jõuda.

NÄIDE 1:

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

Vastus:

24 (vastab joonisele)

0	3	1	2	9
7	3	4	9	9
1	7	5	5	3
2	3	4	2	5

NÄIDE 2:

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

Vastus:

15

6.11.9 Optimaalne ruuting

IP (*Internet Protocol)* on peamine reeglistik, mille abil arvutid Internetis andmeid edastavad. IP oluline omadus on see, et ta defineerib **ruutingu** ehk meetodi, kuidas andmepakette on võimalik saata läbi paljude arvutite. Tavaliselt on igas vahepealses arvutis (ruuteris) tabel, mis ütleb, millisesse järgmisse punkti antud sihtaadressiga pakett tuleb saata. Pakett rändab ühest ruuterist teise kuni jõuab sihtkohta. Ühenduse kogukiirus sõltub seega vahepealsete etappide kiirusest.
Antud on rida ruutereid nendevaheliste ühenduste ja kiiruste infoga. Leida minimaalne võimalik aeg, mis kulub info ühest arvutist teise saatmiseks.
Sisendi esimesel real on neli täisarvu: arvutite arv `M` (`2≤M≤20000`), ühenduste arv `N` (`0≤N≤50000`), algse arvuti indeks `A` (`0≤A<M`) ja sihtarvuti indeks `O` (`0≤O<M`). Järgmisel `N` real on ühenduste info, mis on väljendatud kolme täisarvuna: ühendatud arvutite indeksid ning ühendusele kuluv aeg millisekundites. Väljundisse kirjutada minimaalne aeg, mis kulub arvutist `A` arvutisse `O` paketi saatmiseks. Kui ühendust luua pole võimalik, väljastada `-1`.

NÄIDE 1:

```
2 1 0 1
0 1 100
```

Vastus:

```
100
```

NÄIDE 2:

```
3 3 2 0
0 1 100
0 2 200
1 2 50
```

Vastus:

```
150
```

NÄIDE 3:

```
2 0 0 1
```

Vastus:

```
-1
```

Joonisel on Eestit katva lairibavõrgu ESTWin kaart.

6.11.10 Liftid

Suurtes pilvelõhkujates on sageli erineva kiirusega liftid, mis teenindavad erinevaid korruseid. Näiteks võib seal olla tavaline lift, mis liigub korruste 41 ja 50 vahel ning ekspresslift, mis peatub igal kümnendal korrusel. Sa oled ühe sellise pilvelõhkuja fuajees (0-korrusel), ülesandeks on leida, kuidas võimalikult kiiresti jõuda ettenähtud korrusele. Liikuda võib ainult liftidega ja ühelt liftilt teise minek võtab alati ühe minuti.

Sisendi esimesel real on kaks täisarvu: liftide arv `N` (`1≤N≤5`) ja korrus `K`, kuhu on vaja jõuda (`1≤K≤100`). Teisel real on `N` täisarvu, mis tähistavad liftide kiirusi: sekundite arvud, mis kulub ühe korrusevahe läbimiseks. Järgmistel `N` real on igaühel rida täisarve, mis tähistavad, millistel korrustel vastav lift peatub. Väljundisse kirjutada üks täisarv: sekundite arv, mis kulub soovitud korrusele jõudmiseks. Kui sihtkohta jõudmine pole võimalik, kirjutada väljundisse `-1`.

NÄIDE 1:

```
2 30
10 5
0 1 3 5 7 9 11 13 15 20 99
4 13 15 19 20 25 30
```

Vastus:

`275` - Esimese liftiga 13. korrusele (130 sekundit), oodata teist lifti (60 sekundit), teise liftiga 30. korrusele (85 sekundit).

NÄIDE 2:

```
2 30
10 1
0 5 10 12 14 20 25 30
2 4 6 8 10 12 14 22 25 28 29
```

Vastus:

`285` – Esimese liftiga 10. korrusele, siis teisega 25. korrusele, siis jälle esimesega 30. korrusele, aeg kokku 10*10+60+15*1+60+5*10=285 sekundit.

NÄIDE 3:

```
3 50
10 50 100
0 10 30 40
0 20 30
0 20 50
```

Vastus:

`3920` – Esimese liftiga 30. korrusele, teisega 20. korrusele, siis kolmandaga 50. korrusele.

NÄIDE 4:

```
1 1
2
0 2 4 6 8 10
```

Vastus:

`-1`

6.12 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk6 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
SGL.cpp, SGL.java, SGL.py	Graafi sügavuti läbimine, tsüklilisuse kontroll
Ratsu1.cpp, Ratsu1.java, Ratsu1.py	Ratsu teekond (sügavuti läbimine).
LGL.cpp, LGL.java, LGL.py	Graafi laiuti läbimine.
Ratsu2.cpp, Ratsu2.java, Ratsu2.py	Ratsu lühim tee ühelt ruudult teisele (laiuti läbimine).
Ratsu3.cpp, Ratsu3.java, Ratsu3.py	Ratsude asetamine malelauale (sidususkomponendid).
Veekogud.cpp, Veekogud.java, Veekogud.py	Veekogude loendamine kaardil (üleujutamine).
Alused.cpp, Alused.java, Alused.py	Graafi kahealuseliseks jaotamine.
Loomad.cpp, Loomad.java, Loomad.py	Tugevuse järgi järjestamine (topoloogiline sorteerimine).
Soiduaeg.cpp, Soiduaeg.java, Soiduaeg.py	Asulatevahelised sõiduajad (Dijkstra algoritm).
Kahendpuud.cpp, Kahendpuud.java, Kahendpuud.py	Kahendpuu taastamine ees- ja keskjärjestuse põhjal ning lõppjärjestuses töötlemine.