Graaf

Graaf on kõige üldisem võimalus andmete vaheliste seoste kujutamiseks. Nii lineaarloend kui ka puu on vaadeldavad graafi kitsamate erijuhtudena.

Graaf on struktuur, mille abil saab modelleerida objektide vahel esinevaid paari-kaupa suhteid/seoseid.

Esimese graafiteooria probleemi Königsbergi sildadest esitas 1736. a. Šveitsi matemaatik ja füüsik Leonhard Euler (1707-1783). Euleri tulemused jäid pikemaks ajaks unustusse ning graafe on korduvalt „taasavastatud”. Nii avastas need näiteks G. R. Kirchhoff (1824-1887) 1845. aastal vooluahelate kohta käivates Kirchoffi seadustes.

Graafi mõiste võttis kasutusele inglise matemaatik James Joseph Sylvester (1814-1897) 1878. aastal.

Graaf `G=(V, E)` on järjestatud paar **mittetühjast hulgast** `V` ja selle hulga elementide **paaride hulgast** `E`. Hulga `V` elemente nimetatakse **graafi tippudeks** ja hulga `E` elemente graafi **servadeks, kaarteks** või **seosteks**.

Põhimõisted graafi kohta

Graaf *(graph)* koosneb **tippudest** *(vertex, node)* ja tippe ühendavatest **kaartest (servadest)** *(edge)*.

> **Suunatud** ehk **orienteeritud graaf** *(directed graph, digraph)* on graaf, mille kaartel on suund, st iga kaare jaoks on määratud, millisest tipust ta algab ja millises tipus lõppeb. Teisisõnu võib öelda, et graafi tipud on paari kaupa järjestatud (joonisel tähistatakse noolega kaare otsas).
>
> **Suunamata** ehk **orienteerimata graafis** ehk lihtsalt **graafis** *(undirected graph)* on seos kahe tipu vahel mõlemas suunas. See kehtib kõigi graafi servade kohta. Joonisel sel juhul nooli ei märgita.

Suunatud graafis eristatakse neid tippe, kuhu ühtegi kaart ei sisene *(source)* ja tippe, kust ühtegi kaart ei välju *(sink)*.

Kaks tippu on seotud **külgnevussuhtega** ehk **naabrussuhtega** *(adjacency relation)*, kui ühest tipust läheb kaar teise tippu. Öeldakse ka, et need tipud külgnevad või nad on **naabertipud**. Suunamata graafi puhul on tegemist sümmeetrilise seosega (`u` on `v` naaber ja `v` on `u` naaber).

Suunatud graafil tuleb arvestada kaare suunaga ning seos ei ole sümmeetriline.

Kasutatakse ka mõistet **tipu naaber**. Kui kaar läheb tipust `v` tippu `w`, siis on `w` tipule `v` naaber ehk temaga külgnev tipp. Suunatud graafi puhul aga vastupidist seost ei ole, st `v` ei ole `w`-le naaber ega külgnev tipp.

Graafis saab leida **tee** (*path*) ehk **ahela** ühest tipust teise tippu (suunatud graafi puhul ei pruugi teed iga tipupaari jaoks leiduda). Teel on pikkus.

Tee, marsruut ehk **ahel** on selline kaarte järgnevus, kus ühe kaare lõpp-punkt on järgmise kaare alguspunktiks. Suunamata graafis saab tee minna läbi kaare mõlemat pidi, suunatud graafis tuleb arvestada kindlasti kaare suunaga. Sama kaar või sama tipp võivad ahelas korduda.

NB! Ahela mõistet käsitleb diskreetne matemaatika veidi teisiti: Ahel on marsruut, milles ükski serv ei esine korduvalt.

Kaks tippu `v` ja `u` on **seotud**, kui nende vahel on tee.

Elementaarahel on ahel (kaarte ja servade jada), mis ei lähe läbi ühegi tipu üle ühe korra.

Lihtahel on ahel, milles ei ole ühtegi serva topelt.

Kui ahela algus ja lõpp on samas tipus, siis on tegemist **tsükliga**.

Ja siit saame vastavalt **elementaartsükli** (elementaarahel, mis lõppeb samas tipus) ja **lihttsükli** (lihtahel, mis lõppeb sama tipus) mõisted.

Sidus graaf *(connected graph)* on graaf, milles iga kahe tipu jaoks leidub neid tippe ühendav lihtahel.

Hamiltoni tsükkel on elementaartsükkel, mis läbib kõiki graafi tippe.

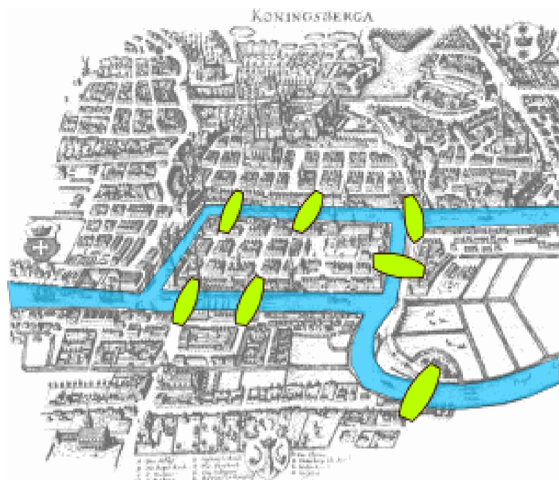
Euleri tsükkel on lihttsükkel, mis läbib kõiki graafi servi ühe korra.

Kui graafis ei ole ühtegi tsüklit (suvalisest tipust ei leidu teed samasse tippu tagasi), nimetatakse graafi **atsükliliseks** (*acyclic graph*).

Suunatud atsükliline graaf *(directed acyclic graph ehk DAG)* on suunatud graaf, kus puudub tsükkel. Sellist liiki graaf on oluline paljude ülesannete lahendamisel.

Graafi nimetatakse **täielikuks** *(complete graph)*, kui tal on kaared kõigi tippude vahel. **Tühigraafis** ei ole aga ühtegi kaart.

Königsbergi sillad



Joonis 1: Königsbergi sillad (Allikas: http://et.wikipedia.org/wiki/K%C3%B6nigsbergi_sildade_probleem)

Königsbergi sildade probleemi peetakse esimeseks kirjeldatud graafiprobleemiks (ehkki sõna „graaf“ sel ajal ei kasutatud).

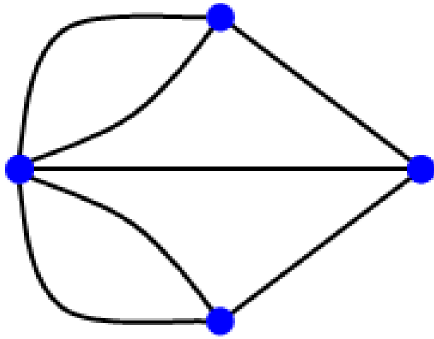
Pregeli jõel Königsbergis oli L. Euleri eluajal 7 silda (Joonis 1). Euler küsis: kas lugupeetavad linnakodanikud saavad pühapäevasel jalutuskäigul ületada kõik sillad ühe korral ja jõuda koju (probleem 1)? Kas kodanikud saavad planeerida jalutuskäigu nii, et iga sild ületatakse ühe korra (tagasi lähtepunkti jõudmine ei ole oluline - probleem 2)?

Tänapäeval saame leida lahenduse järgmisi mõisteid ja teadmisi kasutades (kui ei viitsi lõputult sõrmega mööda kaarti teid taga ajada):

Euleri tee ehk **Euleri ahel** graafis on tee, mis läbib graafi kõik servad ühe korra (probleem 2). **Euleri tsükkel** on Euleri tee, mis moodustab tsükli ehk jõuab lähtetippu tagasi, olles eelnevalt läbinud kõik graafi servad ühe korra. (probleem 1).

Vastavalt Euleri teoreemile esineb **Euleri tsükkel** parajasti siis kui graaf on sidus ja selles ei ole paarituarvulise astmega tippe.

Euleri tee esineb graafis parajasti siis kui graaf on sidus ja ei sisalda rohkem kui kaht paarituarvulise astmega tippu.



Joonis 2: Königsbergi sillad (Allikas http://et.wikipedia.org/wiki/K%C3%B6nigsbergi_sildade_probleem)

Aga mis on tipu aste? **Tipu (lokaalne) aste** ehk **valents** on servade arv, mille otspunktiks antud tipp on.

Pregeli jõe sildu (graafi served) ja nende poolt ühendatavaid Königsbergi linnaosasid (graafi tipud) näed graafina joonisel (Joonis 2).

Lähtudes eespool toodud Euleri teoreemist - millised on lahendused Euleri püstitatud probleemidele?

Info graafis

Kui graafi kasutatakse mõne ülesande lahendamiseks ja objektide vaheliste seoste modelleerimiseks, paigutatakse tippudesse info nimetatud objektide kohta. Tipud saavad sõltuvalt rakendusest endale nimed, seega on tipud kui mingid objektid, sündmused vms. Serv kahe tipu vahel tähistab seost nende kahe objekti või sündmuse vahel. **Suunamata graafis** tähistab serv võrdset seost tippude vahel mõlemas suunas (seos on ühesugune tipust u tippu v ja vastupidi). **Suunatud graafis** on seosel suund (näiteks võib see näidata sündmuste järgnevust). Kui suunatud graafis on vaja näidata mõlemapidist seost kahe tipu vahel, lisatakse nende tippude vahele kaks kaart.

Graafi kaartega saab siduda arvud. Sel juhul kannab kaar lisainformatsiooni lisaks seoseinfole. Neid arve kutsutakse **kaaludeks** ning vastavat graafi **kaalutud graafiks** *(weighted graph)* ka **märgistatud graafiks** e. **võrguks** *(network)*. Seose kaaludega abil saab näidata seoste tugevust, pikkust vms.

Toimingud graafis

Olulisemad nö **lihttoimingud** graafis hõlmavad manipulatsioone kaarte ja tippudega. Lihttoiminguid on enamasti tarvis kasutada graafi loomisel ja töötlemisel, nad on osaks erinevates graafialgoritmides. Mõned näited lihttoimingutest:

- kaare lisamine (milliste tippude vahele?);
- tipu kustutamine (milline tipp?);
- kaare kustutamine (milliste tippude vahelt?);
- tipu nime küsimine;
- tipu nimega X otsimine (kas on või ei ole: vastus on *true* või *false*).

Seoses sellega, et graafi tippe on erinevate algoritmide käigus vaja külastada, on vaja:

- märgistada, et tippu või kaart külastati;
- küsida, kas tippu või kaart on külastatud.

Graafi kohta saab mitmesugust infot küsida:

- tippude arvu;

- kaarte arvu;
- leida ühe tipu naabreid (oluline enamuses algoritmides);
- kas graafis on tsükkel.

Ja graafis tuleb liikuda, st läbida tippe ja kaari mingis järjekorras.

- kõigi kaarte läbimine üks kord (teatatakse kaarte kaalud, kui neid on);
- kõigi tippude läbimine üks kord (teatatakse tippude nimed).

Lisaks on mitmed traditsioonilised algoritmid, mida erinevate ülesannete lahendamiseks kasutada saab. Need algoritmid kasutavad eelnevalt nimetatud lihttoiminguid Näiteks:

- süvitsi otsing – üks põhialgoritm, millele tuginevad teised algoritmid;
- laiuti otsing – teine põhialgoritm, millele tuginevad teised algoritmid;
- topoloogiline sorteerimine;
- lühima tee leidmine kahe tipu vahel;
- lühimate teede leidmine ühest tipust kõigisse tippudesse;
- tugevalt seotud komponendid (aluseks süvitsi otsimine);
- minimaalne toesepuu (aluseks süvitsi otsimine) jne.

Graafi ADT

Eelnevatele tegevustele tuginedes on võimalik kirjeldada graafi ADT *(abstract data type)*-liides:

`InitGraph(N)`
uue graafi algväärtustamine

`InsertEdge(u,v)`
kaare lisamine tippude u ja v vahele

`RemoveEdge(u,v)`
kaare kustutamine tippude u ja v vahelt

`GetAdjList(v)`
millised on tipu v naabrid? Tagastatakse nimekiri

Tegelikult vaid nende elementaaroperatsioonidega ADT ei saa piirduda. Kuid need operatsioonid on aluseks keerulisemate algoritmide kirjeldamisele.

Realisatsioon

Graafide kujutamiseks on kaks peamist võimalust, millistele tuginedes ka vastavad algoritmid on väljatöötatud: **külgnevusahelana** *(adjacency-list)* ja **külgnevusmaatriksina (**ka **naabrusmaatriks, seosmaatriks)** *(adjacency-matrix)*. Eelistatav realisatsioon sõltub graafi iseloomust, lahendamist vajavast ülesandest, tippude arvust (graafi suurusest) jms. Näiteks eristatakse **tihedaid** *(dense)* ja **hõredaid** *(sparse)* **graafe**. Esimestele on sobivam külgnevusmaatriks (pole liigset mälu raiskamist), teistele pigem külgnevusahel.

Graafi tihedust defineeritakse kui **keskmist tipu astet** *(average vertex degree)*: $2E/V$. Tipu aste näitab mitu kaart antud tipuga seotud on. Tippude arvu tähistatakse traditsiooniliselt V ja servade arvu E-ga.

Graaf on tihe siis, kui keskmine tipu aste on proportsionaalne tippude arvuga. Või kui servade hulk E on proportsionaalne tippude arvu ruuduga V^2. Loomulikult ei arvuta siin keegi välja täpset suhet, vaid tegemist on üldise hinnanguga.

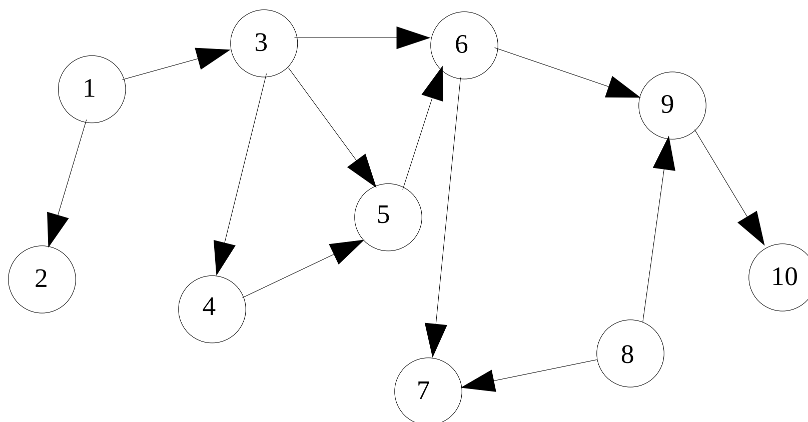
Täpsemad võimalused graafi realiseerimiseks sõltuvad kasutatavast programmeerimiskeelest. Järgnevas tekstis tuuakse näited keele C baasil, kasutades nii maatrikseid kui ka ühe või kahe viidaga ahelaid. Kuid tasub meeles pidada, et erinevad keeled pakuvad erinevaid võimalusi.

Staatiline realisatsioon

Staatiline realisatsioon esitab graafi tippudevahelised seosed **külgnevus-** ehk **naabrusmaatriksina** *(adjacency matrix)*. Nii ridu kui veerge on maatriksis sama palju kui graafis tippe. Igasse maatriksi lahtrisse kantakse 1 või 0. Kaar on kahe tipu vahel siis, kui maatriksisse on märgitud 1 *(true)* ning kaare puudumist tähistab 0 *(false)*. Suunamata graafi puhul on tegemist peadiagonaali suhtes sümmeetrilise maatriksiga.

Graafi tippude infot saab säilitada eraldi tippude kogumis, kuhu pannakse iga tipu kohta oluline info (võti jms). Maatriksi ülesandeks on näidata, millised on seosed tippude vahel. Seega oleks tippude kogum nagu tippude hulk V ning külgnevusmaatriks kui kaarte hulk E. Graafi esitamist maatriksina kasutatakse matemaatikas.

Programmeerides on vaja graafi jaoks deklareerida kahemõõtmeline massiiv, mille elemendid on kas täisarvud (kaalumata graafi puhul 0 ja 1) või ka boolean-tüüpi väärtused. Kaalutud graafi puhul sõltub kaalust. Massiiv täidetakse arvudega sama põhimõtte järgi nagu maatriks.



Joonis 3: Suunatud graaf

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	1	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	1	0	1	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

Joonisel (Joonis 3) olevat graafi näed massiivis ehk **naabrusmaatriksis**. See võib koosneda nii tõeväärtus kui ka täisarv-elementidest. Sõltub konkreetsest keelest, milliseid andmetüüpe kasutada. Kaalutud graafis on mõttekas naabrusmaatriksis salvestada kaarte kaalud. Nii võivad ühtesid asendada muud (täis)arvud. Seda aga eeldusel, kaal 0 ei ole võimalik. Kaare puudumist võib kokkuleppeliselt esitada ka muu arvuga.

Staatiline struktuur graafi hoidmiseks võiks C-s olla järgmine.

```
#define VertMax 100
    struct graph {
           int V,E;      /* tippude ja kaarte arvud */
```

```
                int Adj[VertMax][VertMax];
                int Vert[VertMax];
        };
```

Graafi kasutamiseks tuleb nii külgnevusmaatriks ja tippude massiiv algväärtustada. Seda võib teha järgmine funktsioon:

```
     struct graph *GraphInit(V) {
           int i,j;
           struct graph *myGraph;
           myGraph = malloc(sizeof *graaf);
           myGraph->V = 0;
           printf("%d",myGraph->V);
           myGraph->E = 0;
           for (i=0;i<V;i++){
               myGraph->Vert[i] = 0;
               for (j=0;j<V;j++){
                   myGraph->Adj[i][j] = 0;
               }
           }
      return(myGraph);
}
```

Funktsioon kaare lisamiseks tippude `u` ja `v` vahele on järgmine. NB! Graaf on suunamata.

```
     InsertEdge(struct graph *myGraph,int u,int v){
           myGraph->Adj[u][v] = 1;
           myGraph->Adj[v][u] = 1;
           myGraph->E++;
     }
```

Keerulisemad on algoritmid graafi läbimiseks ja teede leidmiseks. Näiteks selleks, et leida, kas tipust x läheb tee tippu y tuleb graafi maatriksiga arvutusi teha.

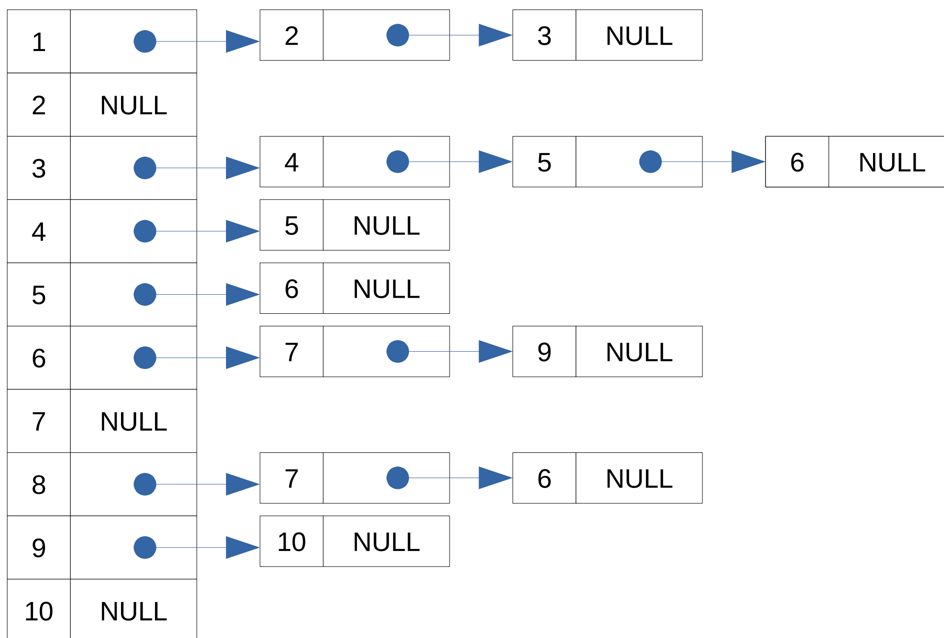
Graafide töötlemisel kasutatakse laiuti või süvitsi otsimise tehnikat. Nimetatud erinevatest algoritmide koostamise strateegiatest tuleb juttu edaspidi.

Dünaamiline realisatsioon

Hõreda graafi ökonoomsemaks kujutamiseks saab kasutada **külgnevusahelat** *(adjacency list)*.

Graafi tippudest moodustatakse massiiv, üks massiivi lahter / element graafi iga tipu jaoks ning iga tipu lahtri külge kinnitatakse ahel antud tipuga külgnevatest tippudest. Ahela lõpust annab märku `NULL` viit. Vaata Joonis 4.

Selliselt saab mälu, aga ka töötlemiseks kuluvat aega oluliselt kokku hoida. Siit võite endale ise ette kujutada, mida sel juhul tähendaksid uue kaare lisamine (riputame ahela algusesse või lõppu uue elemendi), kaare kustutamine (kustutame elemendi) ja naabrite leidmine (läbime vastava ahela).



Joonis 4: Külgnevusahel joonisel 3 kujutatud graafile.

Kõigi tegevuste puhul tuleb arvestada, kas graaf on orienteeritud või mitte. Kaare lisamisel tippude u ja v vahele tuleb dünaamilise realisatsiooni korral lisada kaks elementi - tipp u tipu v naabrite ahelasse ning tipp v tipu u naabrite ahelasse. Realisatsiooniks tuleb mängu panna kõik oskused, mis ahelaid programmeerides tekkinud on ja enamgi veel.

Graafiprobleemid

Mõned probleemid, millele on võimalik vastuseid leida kasutades graafide algoritme:

- Kuidas jõuab kõige kiiremini Tallinnast Värskasse?
- Kuidas transportida kaupa kõige odavamalt erinevatelt müüjatelt erinevatele ostjatele.
- Kuidas korraldada tööde järjekord nii, et maja saaks valmis lühima ajaga?
- Millises järjekorras valida õppeained, et võimalikult lühema ajaga ülikool lõpetada?
- Kuidas suunata netiliiklus efektiivselt läbi ruuterite?
- On antud linnade ja nende vaheliste kauguste nimekiri. Milline on lühim võimalik teekond, mis läbib kõik linnad täpselt ühe korra ja jõuab algpunkti tagasi? See on nn rändkaupmehe ülesanne *(travelling salesman).* Naiivse jõumeetodil lahenduse korral (leime kõikvõimalikud teekonnad, valime lühima) on tegemist faktoriaalse keerukusega algoritmiga.
- Graafiteooria probleemiks liigitatakse ka tuntud nelja värvi probleem. Kui tasapind on jagatud pidevateks piirkondadeks, moodustades kaardi-taolise kujundi, saab selle värvida nelja erineva värviga nii, et naaberpiirkondi ei värvita sama värvi. Nelja värvi probleem *(four-color problem / theorem).*
- Kuidas joota trükiplaadil kokku kontaktid nii, et kõik oleksid omavahel seotud, kuid seoseid oleks minimaalne arv või et lisatavad "traadid" oleksid minimaalse pikkusega?
- Jne .....

Kõigi selliste probleemide lahendamiseks tuleb kõigepealt olukord / andmed kirjeldada graafina ja seejärel graafialgoritme kasutades leida lahendused. Olles probleemi graafina kirja pannud pole enam näiteks vahet, kas uurime tööde tegemise optimaalset järjestust või parimat õppeainete läbimise järjekorda – tegevus taandub samale (topoloogilise sorteerimise) algoritmile.

Graafi tippude läbimine

Graafi süstemaatiliseks uurimiseks tuleb tema tippe ja kaari ükshaaval uurida. Kui on näiteks vaja teada saada iga tipu järku, ei ole oluline, millises järjekorras graafi tippe uurida. Kuid mitmed graafi omadused on seotud teedega tippude vahel ja kõige loomulikum viis neid uurida, on liikuda tipust tippu, kasutades selleks graafi kaari. Graafi kõiki tippe võib vaja olla läbida mitmel erineval eesmärgil, kuid on oluline teha seda vaid lubatud viisil, st lähtudes olemasolevatest kaartest ja nende suundadest.

Klassikaliselt kasutatakse kaarte/tippude läbimiseks kahte erinevat strateegiat ja sõltub kaugemast eesmärgist, kumba nendest valida. Need strateegiad on **laiuti otsimine** ja **süvitsi otsimine**. Ehkki mõistes sisaldub sõna "otsimine", ei ole tegemist klassikalises mõttes otsimisega. Eesti keeles kasutatakse ka mõisteid **laiuti läbimine** ja **süvitsi läbimine**.

Mõlemas strateegias lähtutakse ühest graafi tipust ja liikudes tipult tema naabrile läbitakse kõik tipud. Erineb järgmise tipu valimise strateegia. Kõigi graafi tippude läbimine ei ole eesmärk omaette, kuid mõlemad algoritmid on aluseks teiste algoritmide koostamisel (näiteks laiuti otsimine lühimate teede leidmiseks ja sügavuti otsimine topoloogiliseks sorteerimiseks)

Laiuti otsimine

Laiuti otsimine (ka **laiuti läbimine**) *(breadth-first search)* on teatud algoritmi tüüp, mida on sobiv kasutada ülesannete puhul, kus otsitakse parimat lahendust ja see tuleks välja sõeluda paljude võimaluste hulgast. Kõige tüüpilisemaks ülesandeks on lühima pikkusega tee otsimine kahe tipu vahel. **Tee pikkuseks** loetakse antud juhul kaarte arvu, mis uuritavate tippude vahele jääb. Mitte ükski teine tee samade tippude vahel ei tohi sisaldada vähem kaari. Iseloomulikuks laiuti otsimise juures on, et kõiki lahendusvariante ei püüta kätte saada ükshaaval, et neid hiljem võrdlema hakata, vaid neid uuritakse paralleelselt.

Laiuti otsimisel võetakse aluseks lähtetipp `u`. Kõigepealt leitakse kõik tipud, kuhu on võimalik jõuda tipust `u` ühe kaare läbimise järel (st tipu `u` naabrid). Seejärel võetakse aluseks kõik eelnevalt leitud tipud (lähtetipu naabrid) ja fikseeritakse tipud, kuhu saab jõuda kahte kaart läbides jne. Kui sama tipp oli juba vähema hulga kaarte läbimisel saavutatav, siis uut pikemat teed meelde ei jääta. Nii toimides läbitakse lõpuks kõik tipud. Algoritmi töö võib ka varem lõpetada, kui eesmärgiks on lühima tee leidmine tippude `u` ja `v` vahel. Sel juhul võib lõpetada siis, kui tipuni `v` välja jõutakse. Paremat varianti hiljem kindlasti ei tule.

Laiuti otsimise põhimõtet ei kasutata ainult graafide juures.

Laiuti otsimiseks ei esitata graafile mingeid kitsendusi. Graaf võib olla nii suunatud kui ka suunamata. Kui algustipust leidub tee igasse teise graafi tippu (graaf on sidus), siis BFS-algoritmi abil ta ka leitakse.

Algoritm

Algoritmi alguses valitakse algustipp `S` ja sellest lähtudes uuritakse graafi. Lähtetipu valimine on seotud ülesande püstitusega.

Algoritmi töö **tulemuseks** on loetelu tippudest, mis on kättesaadavad tipust `S` lähtudes, lisaks on teada tee pikkus tipust `S` antud tippu (mitu kaart kahe tipu vahele jääb). Tekib puukujuline struktuur (mida küll füüsiliselt üles ei ehitata), nn **laiutiotsimispuu**.

Algoritmi täitmiseks "värvitakse " tipud (`color[u]`), peetakse meeles iga tipu eellast (`P[u]`) ja sammude arvu (st tee pikkust) esimesest tipust antud tipuni (`d[u]`).

Tipud võivad olla valged, hallid või mustad (värvima kindlasti ei pea, kuid meeles peab pidama nende kolme olekut):

- **valge** *(white)* - tipuni pole veel jõutud (kõik tipud, mis on ees);

- **hall** *(grey)* – tipuni on jõutud, tema eellane on fikseeritud, kuid temaga külgnevad tipud pole veel kõik läbi uuritud (tegutsemise front);
- **must** *(black)* – tipu naabrid on läbi uuritud ja rohkem selle tipu kallale minna ei tohi (tagala).

Alguses on kõik tipud valged, ja esimesel sammul värvitakse halliks algustipp S. Edasi värvitakse halliks tema naabrid ja tipp ise mustaks. Iga halli tipuga tehakse järgmist:

- kui mõni naaber on valge, värvitakse ta halliks, halli ja musta naabrit ei puudutata;
- kui kõik naabertipud on hallid või mustad, värvitakse tipp ise mustaks.

Selliselt laieneb läbiuuritud mustade tippude hulk ja hallid tipud on piiriks uuritud ja uurimata ala vahel. Samal ajal saadakse paralleelselt teada teede pikkused algustipust kõigisse teistesse graafi tippudesse.

Realiseerimaks laiutiotsimise algoritmi, võetakse lisaks graafile kasutusse järjekord `Q` hallide tippude ajutiseks hoidmiseks. Järgnevas algoritmis kasutatakse kahte järjekorra protseduuri `Enqueue` (järjekorda lisamiseks) ja `Dequeue` (järjekorrast eemaldamiseks), mida meil seni realiseeritud pole, kuid mis ei tohiks erilist peavalu valmistada. Lisaks on kasutatud keelekonstruktsiooni `foreach`, mille abil töödeldakse läbi elemendid mingist hulgast. Vastav tegevus on dünaamilise ja staatilise realisatsiooni puhul erinev, seetõttu pole teda täpsustatud.

Järgnev algoritm on esitatud pseudokeeles NB! See ei käivitu ei C, Pythoni ega mingi teise keele kompilaatori abil.

```
Breadth_first(G,s)
{Algoritm graafis G laiutiotsimiseks lähtudes tipust s, (raamatust Cormen, Leiserson, Rivest, Introduction to Algorithms)}
Foreach u in G do        {Algväärtustab graafi, tehakse iga tipuga}
   color[u] <- white     {värv}
   d[u] <- -1            {sammude arv algtipust}
   p[u] <- nil           {eelnev tipp}
end foreach
color[s] <- grey         {Algväärtustab algustipu ja temaga seotud näitajad}
d[s] <- 0
p[s] <- nil
Enqueue(Q,s)             {Hall algustipp pannakse järjekorda}
while Q not Empty do
   Dequeue(Q,u)          {Järjekorrast võetakse hall tipp u}
   foreach v in Adj[u] do {Iga valge tipuga v, mis on tipu u naaber}
      if color[v]=white then
         color[v]<-grey {Tipp värvitakse halliks}
         d[v]<-d[u]+1   {Salvestatakse tema teepikkus algtipust}
         p[v]<-u        {Salvestatakse tema eellane}
         Enqueue(Q,v)   {Pannakse tipp järjekorda}
      endif
   endforeach
   color[u]<-black       {Värvitakse uuritud tipp mustaks}
endwhile
```

Algoritm sobib nii külgnevusahela kui ka külgnevusmaatriksi töötlemiseks. C-programmi kirjutamisel tuleb tegevused C-s ning sõltuvalt valitud realisatsioonist vastavas stiilis kirja panna. Järjekorra jaoks oleks hea kasutada vastavat andmetüüpi ja funktsioone põhitegevuste jaoks (võttes eeskuju pinu funktsioonidest).

Laiuti otsimise algoritm on aluseks teiste algoritmide koostamisele graafiprobleemide lahendamiseks.

Süvitsi otsimine

Süvitsi otsimise *(depth-first search)* starteegia on sarnane labürindi läbimisele. Juhul, kui seda vähegi süstemaatiliselt teha. Labürint, see on hulk käike, mis võivad omavahel lõikuda/hargneda. Mõtle, milline võiks olla starteegia, et labürint läbi käia ja kusagilt väljapääs leida või jõuda tagasi lähtepunkti. (Vana legendi järgi vedas Theseus laiali lõnga Minotauruse koobastes, et mitte ära eksida – ühelt poolt oli tal loodetavasti strateegia, teisalt märgistas ta oma teekonda.)

Süvitsi otsimise strateegia seisneb selles, et minnakse ühte teed mööda nii kaugele (sügavale), kui see võimalik on (st tipult tema naabrile ja sellelt omakorda tema naabrile jne). Kui tipul rohkem naabreid pole, pöördutakse tagasi ja otsitakse teist teed. Nii jätkatakse seni, kuni leidub tippe, mida pole veel külastatud, kuid mis on algustipust kättesaadavad. Kui sellisel viisil rohkemate tippude juurde ei pääse, kuid on veel uurimata tippe, võetakse neist suvaline ja korratakse tegevust. Sellise tegevuse tulemusena saadakse **süvitsi otsimise puu(d)** *(depth search tree)* ja mitme puu puhul tekib **mets**. Iga tipp võib sattuda vaid ühte otsimispuusse ja seega puud ei lõiku.

Algoritmi kasutamiseks sobib nii orienteeritud kui ka orienteerimata graaf.

Väljendades sama algoritmi rekursiivselt: tipp märgistatakse läbituks, seejärel läbitakse rekursiivselt kõik tipu naabrid, mida ei ole veel läbitud. Tavaliselt esitatakse seda algoritmi rekursiooni kasutades. Kuid on ka lihtne võimalus saada laiuti otsimisest süviti otsimine - nimelt kui vahetada hallide tippude hoiustamiseks kasutatav järjekord pinuga. Selleläbi muutub järgmise tipu valimise loogika ja koos sellega tippude läbimise järjekord.

DFS-i saab kasutada graafis tsüklite leidmiseks, kahe tipu vahelise seotuse (tee) leidmiseks, minimaalse toesepuu (leida V-1 kaart, mis ühendavad V tippu) leimiseks

Algoritm

Graafi tipud on algoritmi töö käigus kolmes erinevas olekus ja nende tähistamiseks võib kasutada värve. Algoritm kasutab värve sarnaselt laiuti otsimisele:

- **valge** tipp on avastamata
- **halli** tipuni on jõutud, kuid tema naabrid on üle vaatamata
- **musta** tipuga on kõik ühel pool, tema naabrid on üle vaadatud.

Iga tipuga seotakse kaks **ajatemplit** *(time stamp)* – `d[v]` märgitakse siis, kui tipp esimest korda avastati ja `f[v]` siis, kui tipp on lõplikult töödeldud ja mustaks värvitud. Neid templeid saab kasutada mitmetes algoritmides, lihtsalt graafi läbimiseks neid vaja ei ole. Kui laiuti otsimise juures halle tippe hoiti ja töödeldi järjekorra põhimõttel, siis sügavuti otsimise jaoks tuleks neid hoopis pinus hoida. Järgnevas algoritmis on pinu tekitamiseks kasutatud rekursiivset funktsiooni väljakutsumist.

Algoritm `DFS(G)` sisaldab rekursiivset funktsiooni `DFS-Visit(u)`.
{Algoritm graafis G süviti otsimiseks lähtudes tipust s, (raamatust Cormen, Leiserson, Rivest, Introduction to Algorithms)}
{color[u] – tipu u värv
P[u] – tipu u vahetu eellane
d[u] – esimene ajatempel
f[u] – teine ajatempel}
time – aeg, mis teatud skeemi järgi tiksub

```
DFS(G)
{Tipud algväärtustatakse – eellast pole, värv on valge}
foreach u in G do
   color[u]<-white
   P[u]<-Nil
endforeach
time<-0
{Tsükkel kõigi tippude läbimiseks}
foreach u in G do
   if color[u]=white then       {Kui tipp on uurimata, siis külastame teda}
```

```
      DFS-Visit(u)
   endif
endforeach
end DFS

{Rekursiivne tipukülastamise funktsioon}
DFS-Visit(u)
color[u]<-grey          {Tipuni u esimest korda jõudmisel värvime ta halliks}
d[u]<-time<-time+1      {Väärtustame esimese ajatempli}
{Külastame rekursiivselt kõiki uuritava tipu valgeid naabreid}
foreach v in Adj[u] do
   if color[v]=white then
      P[v]<-u
      DFS-Visit(v)
   endif
endforeach
color[u]<-black         {Tipp u on lõpuni uuritud, st kõik naabrid on läbi käidud, värvime ta mustaks}
f[u]<-time<-time+1      {Väärtustame teise ajatempli}
end DFS-Visit
```

Funktsiooni `DFS-Visit(u)` ülesandeks on kõigi tipu `u` naabrite ülevaatamine. Tipp `u` värvitakse halliks ning talle lisatakse esimene ajatempel. Kõigi tipu `u` valgete naabrite jaoks rakendatakse rekursiivselt sama funktsiooni. Kui tipu `u` kõik naabrid on läbi uuritud, värvitakse ta mustaks ja pannakse külge teine ajatempel. Kahe ajatempli vahe on sisuliselt see aeg, kui kaua käidi tipust `u` lähtuvas alampuus.

Erinevus BFS-i ja DFS-i vahel tuleneb hallide tippude hoidmise meetodist. Esimesel juhul (BFS) on selleks järjekord, teisel juhul (DFS) aga pinu. Sellest piisab tippude erinevas järjekorras töötlemiseks ning täiesti erinevate tulemuste saavutamiseks.

Graafi kaarte klassifitseerimine

Graafi kaari klassifitseeritakse vastavalt nende osale tekkivas **sügavuti otsimise puus** ning klassifikatsioonil on tähtsus erinevate ülesannete lahendamisel (näiteks tsüklite avastamisel orienteeritud graafis). Sügavuti otismise puusse satuvad vaid need kaared, mida läbimise käigus kasutati.

Kaared jagatakse nelja klassi järgmiselt:

1. **Puu kaared** *(tree edges)* – need on kaared, mis paiknevad süvitiotsimise puus, mis on selle puu ehitamise käigus läbitud.
2. **Tagasiviivad kaared** *(back edges)* – see on kaar (u,v), mis ühendab süvitiotsimise puus tippu `u` tema eellasega `v` (sellisel puhul on ilmselt tegemist tsükliga)
3. **Edasiviivad kaared** *(forward edges)* – ühendavad tippu tema järglasega, kuid ei kuulu otsimispuusse
4. **Ristuvad kaared** *(cross edges)* – kõik ülejäänud kaared. Nad võivad ühendada ühe otsimispuu erinevaid tippe, kui üks tipp pole teise tipu eellane, samuti tippe erinevatest otsimispuudest.

Täiendades DFS-algoritmi on võimalik tuvastada erinevate kaarte klassid. Selleks tuleb vaadata tipu `v` värvi siis, kui kaart `(u,v)` esimest korda uuritakse: kui tipp `v` on valge, siis on tegemist puu kaarega, kui tipp `v` on hall, siis on tagasiviiv kaar ja kui tipp `v` on must, siis kas edasiviiv kaar või ristuv kaar.

Orienteeritud graafis **ei ole tsükleid** siis ja ainult siis, kui süvitiotsimise käigus ei leita ühtegi tagasiviivat kaart.

Oluline erinevus laiutiotsimisega: läbitakse kindlasti kõik graafi tipud ja tekkida võib seega mitu süvitiotsimise puud.

Lühim tee

Lühima tee *(shortest path)* leidmine graafis on oluline ülesanne. Selle abil on võimalik tuvastada sõna otseses mõttes lühimat teed näiteks kahe geograafilise punkti vahel (viimasel juhul on vaja arvestada ka kilometraažiga), kuid lühimaks teeks võib olla ka minimaalne toimingute arv või kontaktide hulk.

Lühima tee väljastamine

Laiuti otsimise algoritm annab tulemuseks info, kui kaugel on ühest tipust (algustipust) kõik teised tipud. Seal juures mõistetakse tee pikkuse all sammude ehk kaarte arvu. Laiuti otsimise tulemusena saadi iga tipu kohta teada, mitu kaart (sammu) on temani algustipust (massiiv `d`) ja millisest tipust siia tuldi (vahetu eellane, massiiv `P`).

Rekursiivne funktsioon, millega saadakse tee `s->v` on järgmine:

```
Print_Path(G,s,v)
{G - graaf
 s - algustipp
 v - suvaline graafi tipp - tee lõpp}
if v=s then Print(s)     {Rekursiooni lõpp – on jõutud tee alguseni}
else
{Tipul, milleni jõuti, puudub vahetu eellane}
   if P[v]=nil then Print("Puudub tee tippude s ja v vahel")
   else
{Rekursiivselt leia tee tipu v eellase juurest algusesse}
      Print_Path(G,s,P[v])
      Print(v)
   endif
endif
```

Mitterekursiivselt on väljastatav tee tagurpidi: `v -> s`.

Variatsioonid lühima tee probleemist

Lühim tee *(shortest path)* – leia lühim tee tipust `u` tippu `v`.

Lühimad teed ühest tipust *(single-source shortest paths)* – leida lühimad teed antud tipust kõigisse ülejäänud tippudesse. Sisuliselt saab selle info peale BFS-i kätte. Tuleb lihtsalt kõigi tippude jaoks teed välja kirjutada.

Lühimad teed kõigi tipupaaride vahel *(all-pairs shortest paths)* – selle ülesande lahendamiseks tuleb BFS teha kõigist tippudest alates ja edasi kirjutada välja kõik teed.

Lühimad teed antud tippu: on antud lõpp-tipp `v`, tarvis on leida lühimad teed, mis ülejäänud tippudest antud tippu viivad (originaalülesanne tagurpidi)

Lühim tee kaalutud graafis

Kui iga sammu (kaare) kaal graafis on 1, on lühima tee probleem lahendatud laiuti otsimise algoritmi abi. Kui aga tahaksime teada saada, milline on lühim tee Pärnust Narva, siis kirjeldatud algoritm ei sobi, sest arvestada tuleb iga teejupi tegelikku pikkust, milleks on vaja kasutada kaalutud graafi.

Graafi Pärnust Narva sõidu optimiseerimiseks saab selliselt, et tuleb atlase järgi kõik teede ristid (või kui olete otsustanud vaid asfalti mööda sõita, siis asfaltteede ristid) märkida kui graafi tipud ja kaared tõmmata vaid nende tippude vahele, milliste ristmike vahel tegelikult teed on. Edasi märgitakse igale kaarele (teejupile) tema pikkus kilomeetrites, mis ka autoatlases olemas on, ning kaalutud graaf ongi valmis. Kaaluks võib ka olla eeldatav aeg antud teelõigu läbimiseks.

Tee kaaluks *(path weight)* on kõigi teed moodustavate kaarte kaalude summa.

Lühima tee kaal *(shortest-path weight)* on vähim kaal kõigi teede kaaludest. See ei pruugi olla sama tee, kus vähim arv samme tehakse. Kaalutud graafis nimetatakse lühimaks teeks kahe tipu vahel teed, millel on väikseim kaal.

Lühima tee iga lõik on ise lühim tee.

Dijkstra algoritm

Algoritme kaalutud graafis lühima tee leidmiseks on mitmeid. Järgnevalt kirjeldatakse **Dijkstra algoritmi**. Nimetatud algoritm töötab orienteerimata kui ka orienteeritud graafis. Teoreetiliselt oleks küll õigem väita, et orienteerimata graafist tehakse kõigepealt orienteeritud graaf, kus iga tavalise kaare asemel on kaks suunatud kaart. Graafis ei tohi olla tsüklit, kus kaarte pikkuste summa tuleks negatiivne. Algoritm töötab ahne algoritmi põhimõttel, st igal sammul tehakse lokaalselt parim otsus ja need otsused viivad kogu probleemi lahenemiseni. Üldine idee on väga sarnane laiutiotsimisele, selle vahega, et juba leitud teepikkuseid võib muuta, juhul kui tuleb välja mõni lühem tee. Ning teepikkus ei ole kaarte arv vaid kaarte kaalude summa.

Algoritm vajab tööks tippude tabelit, kuhu kirjutatakse iga tipu jaoks tema kaugus lähtetipust ja tipu number, kust antud tippu jõuti.

Tippude tabel algväärtustatakse selliselt, et tippudel puuduvad eellased ja kõik kaugused on lõpmata suured.

Alustatakse tipust `s` ja selle tipu kauguseks märgitakse 0.

WHILE-tsükli töö on järgmine: kõigi nende tippude hulgast, mille naabrid on läbiuurimata, valitakse tipp `u`, mille kaugus lähtetipust on minimaalne (esimesel kordusel on selleks lähtetipp). Seejärel vaadatakse üle kõik tipu `u` naabrid `v`, kaasaarvatud need, mida on juba eelnevalt töödeldud, ning kui hetke kaugus tipus `v` on suurem kui kaugus, mis tekiks tippu `v` liikumisel üle tipu `u`, asendatakse nii tipu `v` kaugus `d[v]` kui ka eelmine tipp `P[v]`.

WHILE-tsükkel töötab seni, kuni kõigi graafi tippude naabrid on läbiuuritud.

Ajaliselt kõige kriitilisem on läbiuurimata tippude hulgast kõige väiksema kaugusega tipu leidmine. Seetõttu soovitatakse graafi tippude ajutiseks hoidmiseks kasutada **prioriteetidega järjekorda** *(priority queue)* Q, mille ülalpidamine peaks olema kiirem. Prioriteedi määrab kaugus – mida väiksem kaugus, seda kõrgem prioriteet. Peale tutvumist **kuhjaga** *(heap)*, võiks proovida sellise järjekorra loomist.

Väikese tippude hulga juures võib leida lihtsalt miinimumi kauguste massiivist, jättes kõrvale läbiuuritud tipud.

Algoritmis kasutatakse ka hulka `S`, kuhu pannakse läbiuuritud tipud (sellele võib järjekorrata realisatsioonis vastata tipu märkimine läbiuurituks).

Dijkstra algoritm

{Algoritm graafis G lühima tee leidmiseks lähtudes tipust s, (raamatust Cormen, Leiserson, Rivest, Introduction to Algorithms)}

```
Dijkstra(G,w,s)    {G – graaf; w – kaarte kaalud; s – algustipp}
foreach v in G do {Teepikkuste ja eellaste massiivid algväärtustatakse}
   d[v] <- ∞
   P[v] <- nil
endforeach
d[s] <- 0          {Algustipu s kaugus iseendast on 0}
S <- nil           {Läbi uuritud tipud pannakse edaspidi hulka S}
Q <- V[G]          {Kõik graafi G tipud pannakse prioriteetidega järjekorda
vastavalt teepikkusele massiivis d}
while Q <> nil do
   u <- Min(Q)     {Järjekorrast võetakse vähima teepikkusega tipp}
```

```
        S <- S + u      {Tipp u lisatakse läbiuuritud elementide hulka S}
        foreach v in adj[u] do          {Vaadatakse üle kõik u naabrid}
           if d[v] > d[u] +w (u,v) then   {Kui läbi u minev tee}
              d[v] <- d[u] + w(u,v)       {on lühem seni leitud teest,}
              P[v] <- u                 {asendatakse teepikkus ja eelnev tipp}
           endif
        endforeach
     endwhile
```

Kui graaf sisaldab negatiivse kaalude summaga tsükleid, tuleb kasutada **Bellmann-Fordi** või **Floyd-Warshalli** algoritme.

Topoloogiline sorteerimine

Kui graaf on atsükliline ja orienteeritud (DAG), siis on graafi tippude vahel olemas (ja leitav) **osaline järjestus**.

Topoloogilise sorteerimise *(topological sort)* eesmärgiks on saada selline tippude järgnevus, kus iga tippu töödeldakse enne kui neid tippe, millele ta osutab. **NB!** Mõistest "sorteerimine" ei tohi lasta enda segadusse viia – tüüpilise sorteerimisülesandega siin tegemist ei ole.

Kõigi graafi tippude omavaheliste seoste kohta ei saa otsustada, et üks on suurem kui teine või eelneb teisele, võimalik on seda väita osade või paaride kaupa. Sellised omavahelised suhted on võimalik kirja panna graafina ja järgnevuse leidmise protsessi kutsutakse topoloogiliseks sorteerimiseks. Õigeks vastuseks on tavaliselt mitu erinevat järgnevust.

Topoloogilise sorteerimise ülesanne: on antud atsükliline orienteeritud graaf. Tuleb leida graafi tippude selline lineaarne järjestus, et iga kaar läheb nö väiksema tipu juurest suurema tipu juurde (tekkinud järjestuse mõttes).

Ülesande võib sõnastada ka läbi graafi tippude piltliku ümberpaigutamise: kas graafi tipud on võimalik paigutada ühte ritta selliselt, et kõik kaared oleksid suunatud vasakult paremale? Kui selline ümberpaigutus on loodud, ongi olemas topoloogilise sorteerimise tulemus.

Öeldakse, et tipp `u` eelneb tipule `v`, kui tipust `u` läheb kaar tippu `v`. Võib väita, et atsükliline orienteeritud graaf on alati topoloogiliselt järjestatav ja vastupidi, kui graafi on võimalik topoloogiliselt sorteerida, on tegemist atsüklilise graafiga.

Topoloogilise sorteerimise abil on võimalik lahendada näiteks tööde järgnevuse planeerimise ülesannet. Sel juhul võib kaart tipust teise tõlgendada kui sõltuvust (üks tipp sõltub teisest, ehk kui tipp on mingi töö, siis ühte tööd ei saa enne alustada, kui teine töö on lõpetatud). Topoloogilise sorteerimise tulemus annab aga ühe võimaliku järgnevuse tööde tegemiseks. Et sorteerimise tulemusi võib olla mitu, siis on võimalikud ka mitu erinevat tööde järjekorda.

Algoritm

Kõigepealt tuleb leida üles need tipud, kuhu ühtegi kaart ei sisene. Kui graaf on atsükliline, on vähemalt üks selline tipp kindlasti olemas. Kuidas neid leida (vähemalt ühte)? Selleks tuleks alustades suvalisest tipust, liikuda kaari pidi vastu kaare suunale. Selliselt liikudes jõutakse mingil hetkel kindlasti tipuni, kust enam edasi ei saa (ehk kuhu ühtegi kaart ei sisene).

Variant 1

Kasutatakse sügavutiotsimise algoritmi:

`Topological_sort(G)`

1. Kutsu välja `DFS(G)`

2. Lõpetades tipu töötlemise (`DFS-Visit` viimane või eelviimane lause), lisada see, tipp nimekirja algusesse
3. Väljastada tekkinud tippude nimekiri.

Variant 2

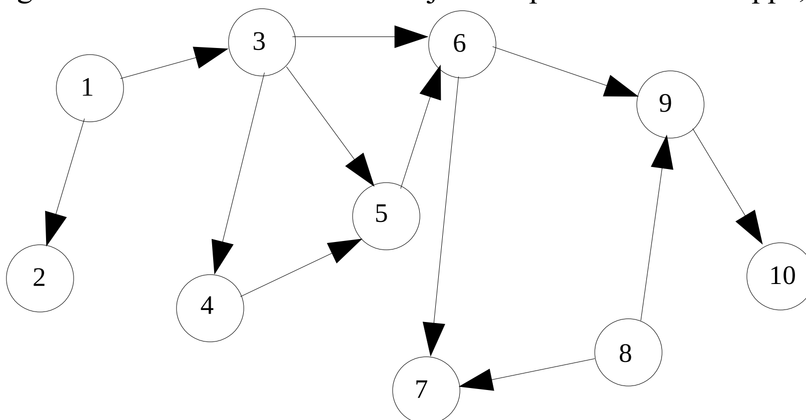
Sobib hästi külgnevusmaatriksit kasutades.

1. Loeme iga tipu kohta kokku, mitu kaart temasse siseneb ehk mitu eellast tal on.
2. Paigutame sorteeritud jadasse kõik need tipud, millel vahetud eellased puuduvad.
3. Vähendame jagasse paigutatud tippude arvel eellaste arvu (kui tipp on jagas ehk töödeldud, annab ta piltlikult öeldes vabaks oma naabertipu.
4. Kui tipu kõik vahetud eellased on jadasse paigutatud, võib sinna paigutada ka tipu enda.

Algoritmi esimene samm oleks seega vahetute eellaste arvu kindlaks määramine kõigi tippude jaoks. Eellaste arvu meelespidamiseks saab kasutada massiivi. Ebamugavam on ühe tipu eellaseid leida külgnevusloendist. Sest seal on info organiseeritud mugavalt just järglaste ehk naabrite leidmiseks.

Parem on neid leida külgnevusmaatriksist. Kui mõnes veerus 1-d puuduvad, ongi eellasteta tipp leitud. Ilmselt on selline meetod kõige lihtsam.

Kui tipp on paigutatud sorteeritud jadasse, tuleb kõigilt tema järglastelt üks eellane maha kustutada ja järgmisel sammul saab sorteeritud jadasse panna taas neid tippe, millel eellaste arv on 0.



Joonis 5: Orienteeritud atsükliline graaf topoloogiliseks sorteerimiseks.

Kui lugeda kokku kõigi joonisel (Joonis 5) oleva graafi tippude eellased, saab järgmise tabeli

tipp	1	2	3	4	5	6	7	8	9	10
eellasi	0	1	1	1	2	2	2	0	2	1

Selle järgi võiks tippude jada alustada nii 1. kui 8. tipuga. Ja edasi on võimalikud mitu erinevat järjekorda.

Kaks võimalikku väljundit oleksid:

```
1 2 3 4 5 6 8 9 10 7
```

```
8 1 3 2 4 5 6 9 7 10
```

Huvitavaks probleemiks on veel, millal tipp kõige varem ja kõige hiljem sorteeritud jadasse sattuda saab. Sellist teadmist võib vaja olla tööde planeerimise ülesande juures. Ja seda saab leida süviti otsimise käigus ajatempleid kasutades.

Toesepuu

Toesepuu *(spanning tree)* on alamgraaf (graafi osa), millesse arvatud servad moodustavad puu, ühendades samal ajal kõik graafi tipud. Servi on eemaldatud nii, et kaoksid tsüklid, ent sidusus kõigi tippude vahel säiluks.

Toesepuu ei ole üldjuhul üheselt määratud.

Kui servadel on mittenegatiivsed kaalud, saab leida **minimaalse toesepuu**. See on toes, mille kogukaal on kõigi toespuude hulgast minimaalne.

Näide Kruskali algoritmist minimaalse toesepuu leidmiseks: http://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Kokkuvõte

1. Graafe saab kasutada väga erinevate struktuuride ja seoste kujutamiseks
2. Paljude ülesannete juures, kui probleem korralikult formuleerida, saab teda modelleerida graafi kasutades selliselt, et lahenduseks sobib mõne tuntud efektiivse algoritmi kasutamine. Tavaliselt graafi töötlemiseks ise algoritme leiutada on keeruline. Seega tasub teada, mis on olemas.
3. Sügavuti ja laiuti otsimine on algoritmid, millega kõik tipud ja kaared läbitakse. Nad on aluseks mitmetele teistele algoritmidele
4. Kuluta suurem aeg graafi mõistlikuks modelleerimiseks, pärast on algoritmi lihtsam kirja panna.