Graaf

Kui ühendada elemendid omavahel viitadega vabamal kujul, kui seda lubavad puude või lineaarloendite reeglid, saadakse graaf. Graafi võib kirjeldada kui andmestruktuuri, mis ei pea olema lineaarne. Graaf on kõige üldisem võimalus andmete vaheliste seoste kujutamiseks. Nii lineaarnimistu kui ka puu on vaadeldavad graafi kitsamate erijuhtudena.

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

Graafi mõiste võttis kasutusele inglise matemaatik J. Sylvester 1878 a. Esimese graafiteooria probleemi Königsbergi sildadest esitas aga L. Euler 1736 (vaata allpool).

Põhimõisted graafi kohta

(Vaata ka diskreetse matemaatika loenguid!!)

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

> **Graaf** on tippude hulk ja servade hulk, mis ühendavad omavahel kahekaupa konkreetseid tippe.
>
> **Suunatud graaf** *(directed graph, digraph)* on graaf, mille kaartel on suund, st iga kaare jaoks on määratud, millisest tipust ta algab ja millises tipus lõppeb e. teisisõnu võib öelda, et on graafi tipud on paari kaupa järjestatud (joonisel tähistatakse noolega kaare otsas).
>
> **Suunamata graafi** ehk **graaf** *(undirected graph)* puhul on seos kahe tipu vahel mõlemas suunas. See kehtib kõigi graafi kaarte kohta. Joonisel nooli ei märgita.

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

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** 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.

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

Kaks tippu on seotud **külgnevussuhtega** *(adjacency relation)*, kui ühest tipust läheb kaar teise tippu. Öeldakse ka, et tipud külgnevad. Suunamata graafi puhul on tegemist sümmeetrilise seosega.

Suunatud graafil tuleb arvestada kaare suunaga ning seos ei ole sümmeetriline. Kui tipust w läheb tippu v kaar, siis v külgneb w-ga.

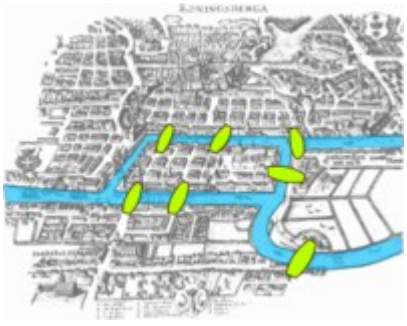
Kasutatakse ka mõistet **tipu naaber**. Kui kaar läheb tipust v tippu w, siis on w tipule v naaber ehk temaga külgnev tipp. Suunatud graafi puhul aga vastupidist seost ei ole, st v ei ole w-le naaber ega külgnev tipp.

Elementaarahel on selline ahel, mis ei lähe läbi ühegi tipu üle ühe korra.

Lihtahel on ahel, mis ei lähe läbi ühegi serva üle ühe korra.

Kui ahela algus- ja lõpp-punkt on samad, siis on tegemist **tsükliga**.

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



Hamiltoni tsükkel on elementaartsükkel, mis läbib kõiki graafi tippe.

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

Graafiteooria tutvustamine matemaatik Leonard Euleri poolt Königsbergi sildade probleemi läbi (1736 a.) – kas lugupeetav linnakodanik saab pühapäevasel jalutuskäigul linna läbiva jõe ääres ja saartel jalutada nii, et ta ületab kõik sillad ühe korra ja saab koju tagasi minna? Vaata kõrval olevat joonist!

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 graaf, kus puudub suunatud tsükkel. See graaf on oluline paljude ülesannete lahendamisel.

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

Info graafis

Kui graafi kasutatakse mõne ülesande lahendamiseks ja objektide vaheliste suhete modelleerimiseks, paigutatakse tippudesse info nimetatud objektide kohta. Tipud saavad sõltuvalt rakendusest endale nimed, seega on tipud kui mingid objektid, sündmused vms. Kaar kahe tipu vahel tähistab seost nende kahe objekti või sündmuse vahel. Suunamata graafi saab kasutada mõlemapidise seose näitamiseks (seos on võrdne tipust u tippu v ja vastupidi). Suunatud graafi puhul on seosel suund (näiteks võib see näidata sündmuste järgnevust). Kui suunatud graafis on vaja näidata mõlemapidist seost kahe tipu vahel, pannakse nende tippude vahele kaks kaart. Suunamata graafis on kahe tipu vahel maksimaalselt üks seos (kaar)..

Graafi kaartega saab siduda arvud. Sel juhul kannab kaar informatsiooni lisaks seoseinfole. Neid arve kutsutakse **kaaludeks** ning vastavat graafi **kaalutud graafiks** *(weighted graph)* e. **võrguks** *(network)*. Kaalutud graafiga saab näidata seoste tugevust, pikkust vms.

Toimingud graafis

Olulisemad nö lihttoimingud graafiga hõlmavad manipulatsioone kaarte ja tippudega. Lihttoiminguid on enamasti tarvis kasutada graafi loomisel ja töötlemisel:

- Kaare lisamine (milliste tippude vahele)
- Tipu kustutamine (milline tipp)
- Kaare kustutamine (milliste tippude vahelt)
- Tipu nime küsimine
- Tipu nimega X otsimine (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
- ühe tipu naabreid
- kas graaf on orienteeritud

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:

- sügavuti otsimine – nö põhialgoritm, millele tuginevad teised algoritmid
- laiuti otsimine – nö põhialgoritm, millele tuginevad teised algoritmid
- topoloogiline sorteerimine
- lühima tee leidmine kahe tipu vahel
- lühima tee leidmine kõigi tippude vahel
- tugevalt seotud komponendid (aluseks sügavuti otsimine)

- minimaalne toesepuu (aluseks sügavuti otsimine)

Graafi ADT

Eelnevatele tegevustele tuginedes on võimalik kirjeldada graafi ADT-liides:

`__init__`

uue graafi algväärtustamine

`InsertEdge(u,v)`

kaare lisamine tippude u ja v vahele

`RemoveEdge(u,v)`

kaare kustutamine tippude u ja v vahelt

`IsNode(v)`

kas tipp v on graafis olemas?

`GetNodes(v)`

millised on tipu v naabrid? Tagastatakse list

Tegelikult nende operatsioonidega liides ei saa piirduda, sest lisada saab mahukamaid toiminguid.

Realisatsioon

Graafide kujutamiseks on kaks peamist võimalust, millistele tuginedes ka vastavad algoritmid on väljatöötatud: **külgnevusnimistuna** *(adjacency-list)* ja **külgnevusmaatriksina** *(adjacency-matrix)*. Eelistatav realisatsioon sõltub graafi iseloomust. Selleks eristatakse **tihedaid** *(dense)* ja **hõredaid** *(sparse)* **graafe**.

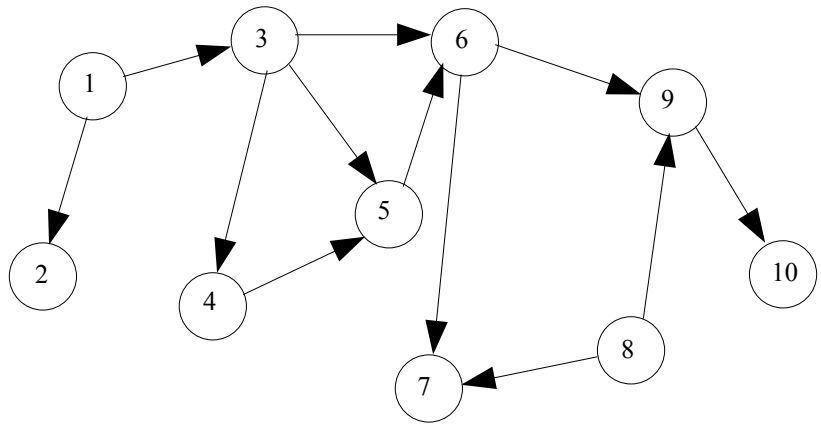
Graafi tihedust defineeritakse kui **keskmist tipu astet** *(average vertex degree)*: *2 E / V*. Tipu aste näitab mitu kaart antud tipuga seotud on. Tippude arvu tähistatakse traditsiooniliselt V ja servade arvu E-ga.
Graaf on tihe siis, kui keskmine tipu aste on proportsionaalne tippude arvuga. Või kui servade hulk E on proportsionaalne tippude ruuduga V^2. Loomulikult ei arvuta siin keegi välja täpset suhet, vaid tegemist on üldise hinnanguga.

Staatiline realisatsioon

Staatiline realisatsioon esitab graafis olevad tippudevahelised seosed **külgnevusmaatriksina** *(adjacency matrix)*. Nii ridu kui veerge on maatriksis sama palju kui graafis tippe. Igasse maatriksi lahtrisse kantakse `True (1)` või `False (0)`. Kaar on kahe tipu vahel sel juhul, kui maatriksisse on märgitud 1 *(true)* ning kaare puudumist tähistab 0 *(false)*. Suunamata graafi puhul on tegemist peadiagonaali suhtes sümmeetrilise maatriksiga. Joonisel on sel juhul kaared ilma noolteta. Suunatud graafil võib olla kahe tipu vahel ka kaks kaart, kui seos mõlemas suunas on olemas.

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

Seega võib Pythonis luua listi, mis koosneb listidest. Alguses tuleks kõik listid nullidega algväärtustada, et vajaliku suurusega andmestruktuur saada ja seejärel on võimalik vajalikesse kohtadesse sisse märkida ühed.



Joonis 1. 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

Joonis 2. Joonisel 1 oleva graafi külgnevusmaatriks

Joonisel 2 olev kaarte massiiv võib koosneda nii tõeväärtustüüpi kui ka täisarvutüüpi elementidest. Sõltub konkreetsest keelest, milliseid andmetüüpe kasutada, kuid täisarv ja tõeväärtustüüp on esimesed valikud. Kui on tegemist kaalutud graafiga, on mõtekas külgnevusmaatriksis näidata kaare kaalu. Nii võivad ühtesid asendada muud (täis)arvud.

Klass graafi kujutamiseks koos konstruktoriga võiks olla järgmine:

```
class graph:

    def __init__(self, vertnum, edges = [], vert = []):
        "Graafi konstruktor, mis peaks tekitama listi tippude jaoks"
        "ja nullidega täidetud listi listidest kaarte jaoks."
        "Tippude arv on määratud vertnum-iga."
        self.edges = []
        self.vert = []
        self.V = vertnum      # atribuut tippude arvu jaoks
        self.E = 0            # atribuut kaarte arvu jaoks
        for u in range(vertnum):
            temp = []
            for v in range(vertnum):
                temp += [0]
            self.edges += [temp]
# Kaarte listi on vaja, kui nad olulist lisainfot sisaldavad.
        for u in range(vertnum):
            self.vert += [u]
```

Meetod kaare lisamiseks etteantud tippude vahele võiks olla järgmine. NB! Graaf on orienteerimata.

```
def InsertEdge(self,u,v):
    "Graafi lisatakse kaar (uv)"
    if self.edges[u][v] == 0:
        self.edges[u][v] = 1
        self.edges[v][u] = 1
        self.E += 1
```

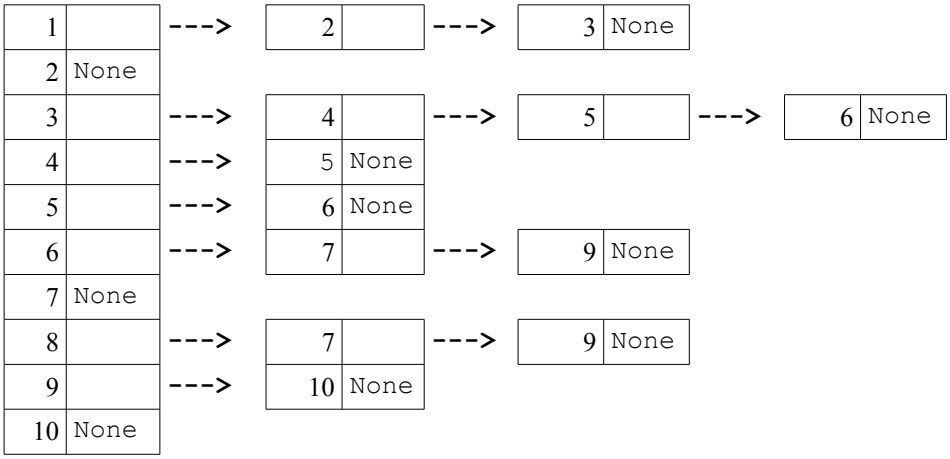
Keerulisemad on algoritmid graafi läbimiseks ja teede leidmiseks. Näiteks selleks, et leida, kas tipust x läheb tee tippu y tuleb maatriksiga arvutusi teha.

Graafide töötlemisel kasutatakse laiuti või sügavuti otsimise tehnikat (kuna teid võib olla palju tuleb kõik võimalused ära proovida, sest tavaliselt ei aita ühe tee leidmisest, vaid see tee peab vastama mingitele tingimustele). Reeglina tuleb siin kasutada rekursiooni. Aitab ka dünaamiline programmeerimine. Nimetatud erinevatest algoritmide koostamise strateegiatest tuleb juttu edaspidi.

Dünaamiline realisatsioon

Hõreda graafi ökonoomsemaks kujutamiseks võetakse kasutusele loendid, nn **külgnevusloend** *(adjacency list)*. Graafi tippudest moodustatakse massiiv, üks on lahter iga tipu jaoks (massiivi võib asendada ka lineaarahel) ning iga tipulahtri külge kinnitatakse lineaarahel nendest tippudest, mis külgnevad antud tipuga. Loendi lõpus on tühi viit. Selliselt saab mälu kokku hoida. Siit võite endale ise ette kujutada, mida sel juhul tähendaksid uue kaare lisamine (riputame loendi lõppu uue elemendi), kaare kustutamine (kustutame elemendi) ja naabrite leidmine (läbime vastava loendi).

Kahe viidaga loend külgnevate tippude hoidmiseks on sobivam.



Joonis 3. Külgnevusloend joonisel 1 olevale 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 tipuga v seotud ahelasse ning tipp v tipuga u seotud ahelasse.

Pythonit kasutades on võimalik tekitada hõreda graafi jaoks list listidest dünaamilise mäluhalduseta. Et puhas ahela tegemine on tülikas ja Pythonile mitte eriti omane, siis võiks järgnev olla loomulikum esitus. Eelnev graafi näide Pythoni listina üleskirjutatult oleks järgmine. NB! Esimene alamlist on tühi, sest tippu 0 antud näites ei ole:

```
graaf = [[], [2,3], [], [4,5,6], [5], [6], [7,9], [], [7,9], [10], []]
```

Kaarte lisamise meetod (kus põhiülesandeks lisada õiged tipud õigetesse alamlistidesse) on järgmine:

```
def InsertEdge(self,u,v):
        "Graafi lisatakse kaar (uv)"
        "Selleks lisatakse "
        "Võiks kaaluda veateate andmist, kui kaar on juba olemas"
        if (v not in self.edges[u]):
            self.edges[u].append(v)
            self.edges[v].append(u)
            self.E += 1
```

Eriti lühike ja lihtne on uuritava tipu naabrite tagastamine:

```
def GetNodes(self,v):
    "Listina tagastatakse tipu v naabrid."
    return self.edges[v]
```

Graafiprobleemid

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

- Kuidas jõuab kõige kiiremini Tallinnast Värskasse?
- Kuidas saab kõige odavamalt Tallinnast Värskasse?
- Kuidas transportida kaupa kõige odavamalt erinevatelt müüjatelt erinevatele ostjatele.
- Kuidas korraldada tööde järjekord, et saada maja valmis kõige lühema ajaga?
- Millises järjekorras valida õppeained, et võimalikult lühema ajaga ülikool lõpetada?
- Millises järjekorras käia ära kõigis linnades, et see toimuks võimalikult ruttu või lühemat teed mööda. Tegemist on klassikalise nn 'rändkaupmehe ülesandega' *(travelling salesman)*.
- Graafiteooria probleemiks liigitatakse ka tuntud nelja värvi probleem (geograafilise kaardi värvimine 4 värvi abil).
- 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 tuleks kõigepealt olukord kirjeldada graafina ja seejärel kasutades graafialgoritme, 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 algoritmile.

Graafi tippude läbimine

Graafi süstemaatilisekls uurimiseks tuleb tema tippe ka kaari ükshaaval uurida. Kui on 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 tee 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 kaartest ja nende suundadest.

Klassikaliselt kasutatakse kaarte/tippude läbimiseks kahte erinevat strateegiat ja sõltub kaugemast eesmärgist, millist nendest valida. Need strateegiad on **laiuti otsimine** ja **sügavuti otsimine**. Ehkki mõistes sisaldub sõna "otsimine", ei ole tegemist klassikalises mõttes otsimisega.

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

Laiuti otsimine *(breadth-first search)*

Laiuti otsimine on teatud algoritmi tüüp, mida on sobiv kasutada ülesannete puhul, kus otsitakse parimat lahendust ja see tuleks välja sõeluda paljude võimaluste hulgast. Kõige tüüpilsemaks ülesandeks on leida lühimat teed kahe tipu vahel. **Tee pikkuseks** loetakse antud juhul kaarte arvu, mis nimetatud tippude vahele jäävad. Mitte ükski teine tee samade tippude vahel ei tohi sisaldada vähem kaari. Iseloomulikuks sellise lähenemise 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 (tipu `u` naabrid). Seejärel võetakse aluseks kõik eelnevalt leitud tipud ja fikseeritakse tipud, kuhu saab jõuda kahte kaart läbides jne. Kui sama tipp oli juba vähema hulga kaarte läbimisel saavutatav, siis uut pikemat teed ei fikseerita. Nii toimides läbitakse lõpuks kõik tipud. Algorimis võib ka varem lõpetada, kui eesmärgiks on lühima tee leidmine tippude `u` ja `v` vahel. Sel juhul võib lõpetada juhul, 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, siis BFS-algoritmi abil ta leitakse.

Algoritm

Algoritmi alguses valitakse algustipp `S` ja sellest lähtudes uuritakse graafi.

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, 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 järglased 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.

Järgneva algoritm on esitatud pseudokeeles NB! See ei käivitu ei Pythoni ega mingi teise keele kompilaatroi abil.

```
Breadth_first(G,s)
{Algoritm graafis G laiutiotsimiseks lähtudes tipust s}
Foreach u in G do        {Graafi algväärtustamine, 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)             {Halli algustipp järjekorda}
while Q not Empty do
   Dequeue(Q,u)          {Võtame järjekorrast halli tipu u}
   foreach v in Adj[u] do {Iga valge tipuga v, mis on tipu u naabriks}
      if color[v]=white then
         color[v]<-grey  {Värvime tipu halliks}
         d[v]<-d[u]+1    {Salvestame tema jaoks teepikkuse algtipust}
         p[v]<-u         {Salvestame tema eellase}
         Enqueue(Q,v)    {Paneme ta järjekorda}
      endif
   endforeach
   color[u]<-black       {Värvime uuritud tipu mustaks}
endwhile
```

Algoritm sobib nii külgnevusnimistu kui ka külgnevusmaatriksi töötlemiseks.

Kirjeldatud algoritm on aluseks teiste algoritmide koostamisele.

Sügavuti otsimine *(depth-first search)*

Sügavuti otsimise starteegia on sarnane labürindi läbimisele. Juhul, kui seda vähegi süstemaatiliselt teha. Labürint, see on hulk käike, kus juures käigud võivad omavahel lõikuda/hargneda. Mõtle, milline võiks olla konkreetne 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.)

Sügavuti otsimise strateegia seisneb selles, et minnakse ühte teed mööda nii 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 **sügavuti otsimise puu(d)** *(depth search tree)* ja mitme puu puhul tekib mets. Iga tipp saab 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.

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 2 **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 protseduuri väljakutsumist.

Algoritm `DFS(G)` sisaldab rekursiivset protseduuri `DFS-Visit(u)`.

{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 protseduur}
DFS-Visit(u)
color[u]<-grey           {Jõudsime tipuni u, 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 uuritud, värvime ta mustaks}
   f[u]<-time<-time+1               {Väärtustame teise ajatempli}
   end DFS-Visit
```

Protseduuri `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 protseduuri. 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).

Kaared jagatakse nelja klassi järgmiselt:

1. **Puu kaared** *(tree edges)* – need on kaared, mis paiknevad otsimise puus, mis on selle puu ehitamise käigus läbitud.
2. **Tagasiviivad kaared** *(back edges)* – see on kaar (u,v), mis ühendab sügavutiotsimise 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 sügavutiotsimise käigus ei leita ühtega tagasiviivat kaart.

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

Lühim tee (shortest path)

Lühima tee leidmine graafis on tihti vajaminev 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 kõik teised tipud. Seal juures mõistetakse tee pikkuse all sammude ehk kaarte arvu. Vastav info on kirjas iga tipu küljes, lisaks on seal kirjas ka vahetu eellane, seega on konstrueeritav tee igast tipust algustippu. Rekursiivne protseduur, 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 (vt tee üleskirjutamist edasi)

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 kõigit 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 tuleks tegelikku teepikkust, 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.

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 töödeldudm, ning kui 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. Graafi tippude hoidmiseks soovitatakse kasutada prioriteetidega järjekorda Q, mille ülalpidamine peaks olema kiirem. Prioriteedi määrab kaugus – mida väiksem kaugus, seda kõrgem prioriteet. Peale tutvumist **kuhjaga** *(heap)*, võiks proovida sellise järjekorra ADT loomist.

Väikese tippude hulga juures võib leida lihtsalt miinimumi läbiuurimata.tippude hulgast, st teha tavaline tsükkel üle kõigi tippude ja töödelda kauguste massiivi.

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

Dijkstra algoritm

```
Dijkstra(G,w,s)   {G – graaf; w – kaarte kaalud; s – algustipp}
foreach v in G do {teedepikkuse ja eellaste massiivid nullitakse}
   d[v] <- lõpmatus
   P[v] <- nil
endforeach
d[s] <- 0         {algustipu s kaugus iseendast on 0}
S <- nil          {Läbi uuritud tipud pannakse hulka S}
Q <- V[G]         {Kõik graafi G tipud pannakse prioriteetidega järjekorda}
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 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 omavahelise suhete 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 nii läbi graafi tippude ümberpaigutamise "pildidl": 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 külgnevusmaatriksi jaoks

Üks võimalus TS algoritmi maha mängida on järgmine:

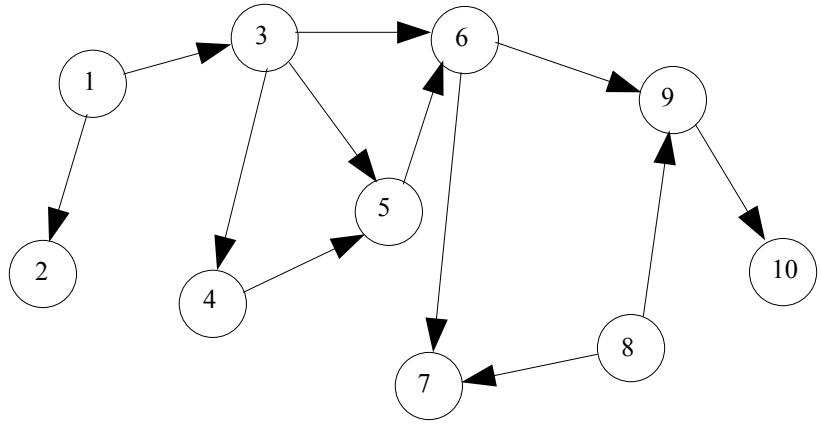
1. Paigutame jadasse kõik need tipud, millel vahetud eellased puuduvad.

2. 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. Sela on info organiseeritud mugavalt just järglaste leidmiseks.

Parem on neid leida külgnevusmaatriksist, liikudes selleks mööda veergu. 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 4 Orienteeritud atsükliline graaf, mida saab topoloogiliselt sorteerida

Kui lugeda kokku kõigi joonisel 4 olevate 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.

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.