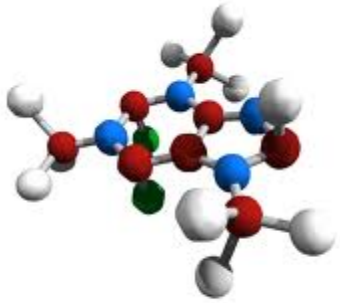
Algoritmid ja andmestruktuurid

- Graafid, nende esitused
- Graafi sügavuti ja laiuti läbimine
- Topoloogiline sorteerimine

Graaf - laialtkasutatav abstraktsioon

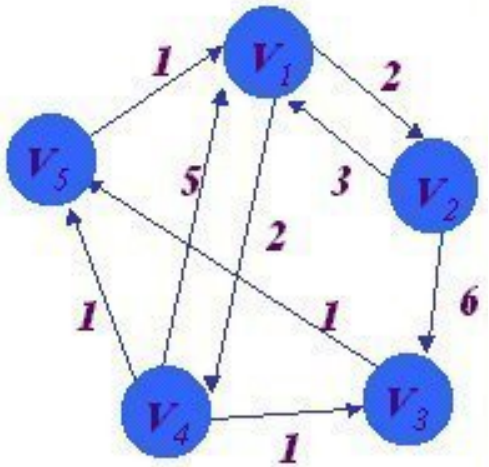
- Paljusid olukordi saab kirjeldada binaarsete suhetena kahe objekti vahel:
  - veebilehekülje ja nendelt lähtuvad viidad
  - sõbrasuhted Facebookis
  - teed punktide vahel, kaardid
  - mitmesugused võrgud
  - olekumuutuse diagrammid



- Graaf on abstraktne struktuur, mis kirjeldab binaarset relatsiooni kahe objekti vahel.
- Mitmete probleemide lahenduse saab taandada üldistele graafialgoritmidele: omavahelise kauguse, lühima tee, kliki, minimaalse katva puu leidmine jne

Graaf

- **Graaf** koosneb **tippudest (*vertex*)** ja **servadest** (***edge***).
- Servad võivad olla **suunatud (*directed*)** ja **suunata (*undirected*)** .
- Servad võivad olla **kaaludega (*weighted*)** ja **kaaludeta (*unweighted*)** .
- **Tee (*path*)** on tippude jada, nii et olemas serv, mis ühendab iga kahte järgnevat tippu.
- **Tsükkel (*cycle*)** on tee tipust iseendasse.
- Graaf on **sidus *(connected)*,** kui igast tipust on olemas tee igasse teise tippu
- **Puu (*tree*)** on sidus graaf, mis ei sisalda tsükleid.



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

Graafi mõistete definitsioon

- Graaf $G = (V, E)$
- Tippude hulk V
- Servade hulk $E = \{ (v_i, v_j) \}, \quad v_i \in V$
- Tee $p = < v_0 , v_1 , v_2 , \ldots , v_k >,$ $(v_i , v_{i+1}) \in E$
- Tsükkel tee kus $v_0 = v_k$

Graafi esitusviisid

Graafi kaks standardset esitust:

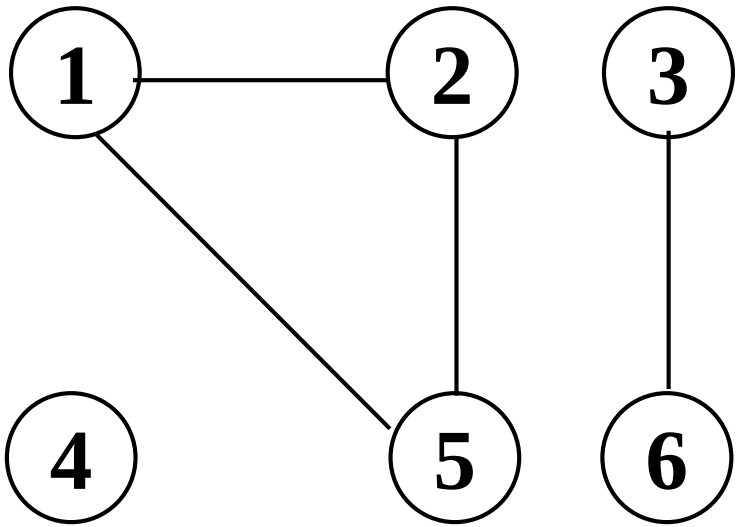
1. **Naabruslist** (*adjacency list*): iga tipu v kohta on olemas list L_v tema naabritest graafis.
   Esituse suurus: $\Theta(|V|+|E|)$.
   Sobivam hõredate graafide esitamiseks.
2. **Naabrusmaatriks** (*adjacency matrix*): $|V| \times |V|$ maatriks kus iga serva $e = (u,v)$ esitab nullist erineva väärtusega element (u,v).
   Esituse suurus: $\Theta(|V|^2)$.
   Sobivam tihedate graafide esitamiseks.

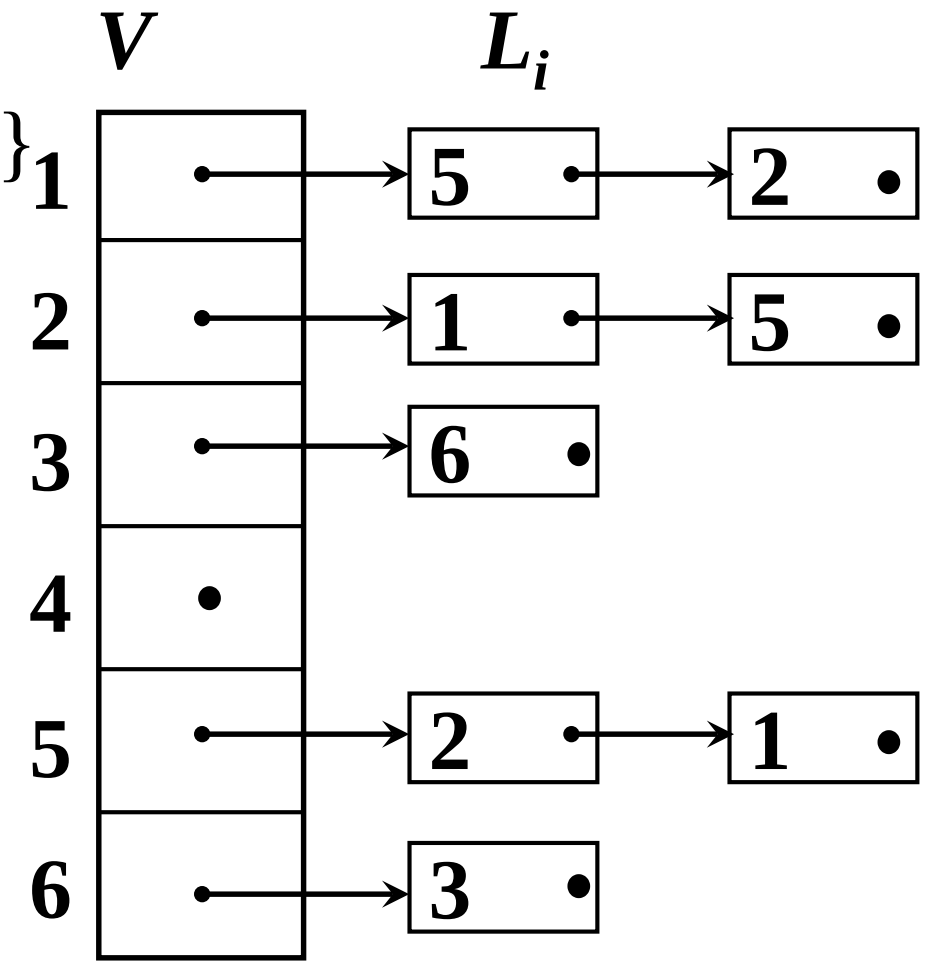
Mõned algoritmid vajavad esitust naabrusmaatriksina

Esitus naabruslistina

$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(1,5),(2,5),(3,6)\}$

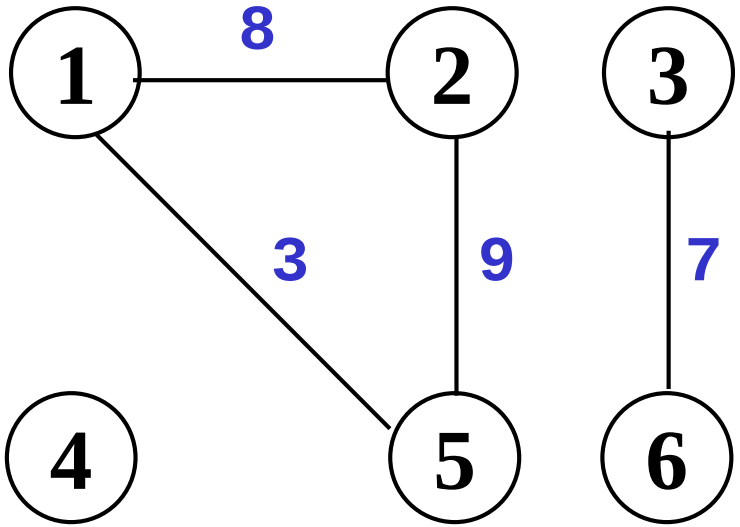


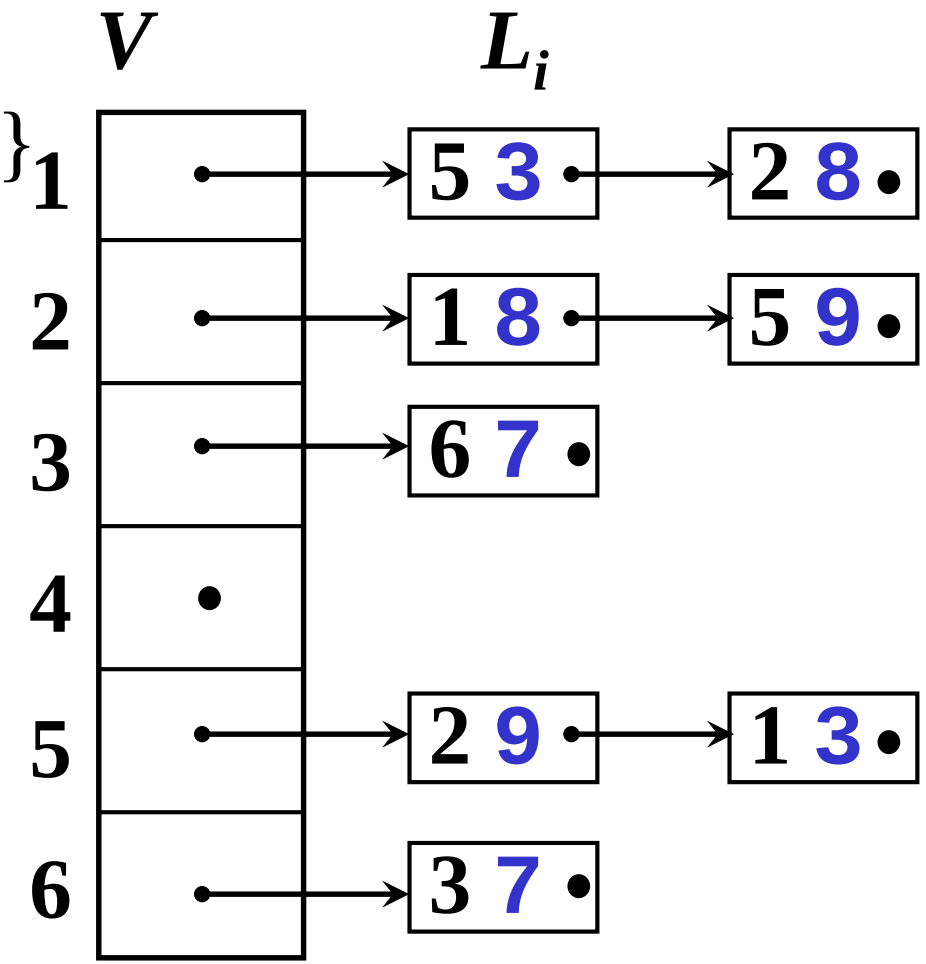


Kaaludega graafi esitus naabruslistina

$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(1,5),(2,5),(3,6)\}$

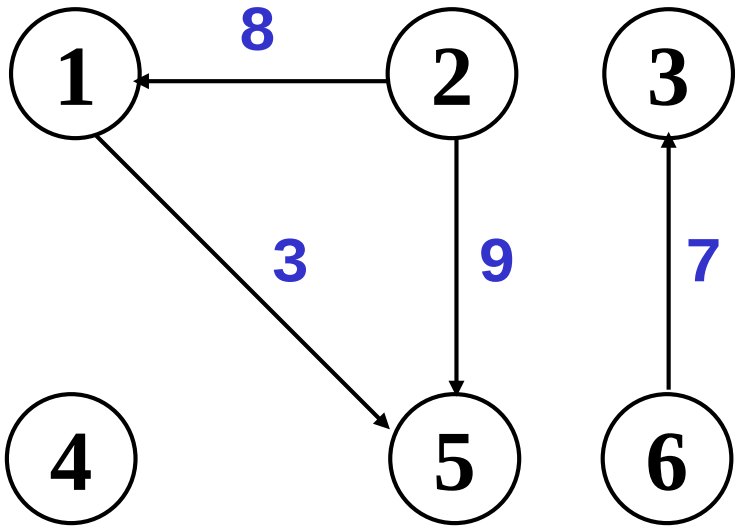


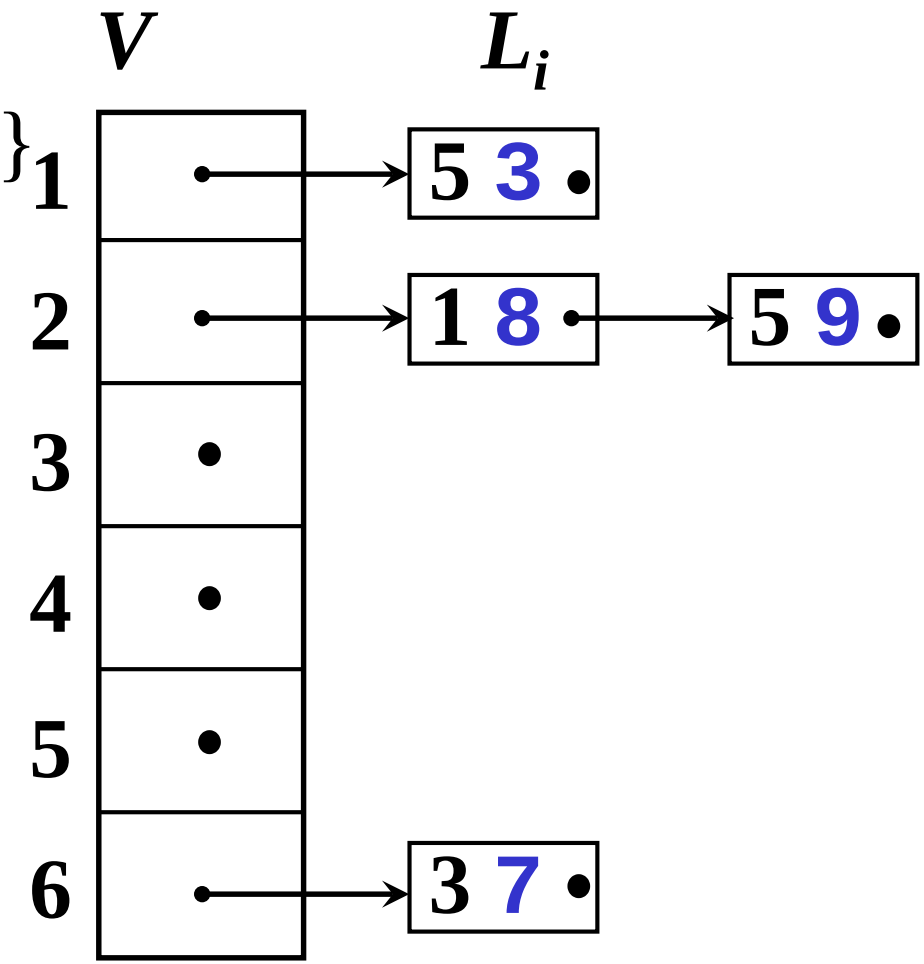


Suuna ja kaaludega graafi esitus naabruslistina

$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(1,5),(2,5),(3,6)\}$

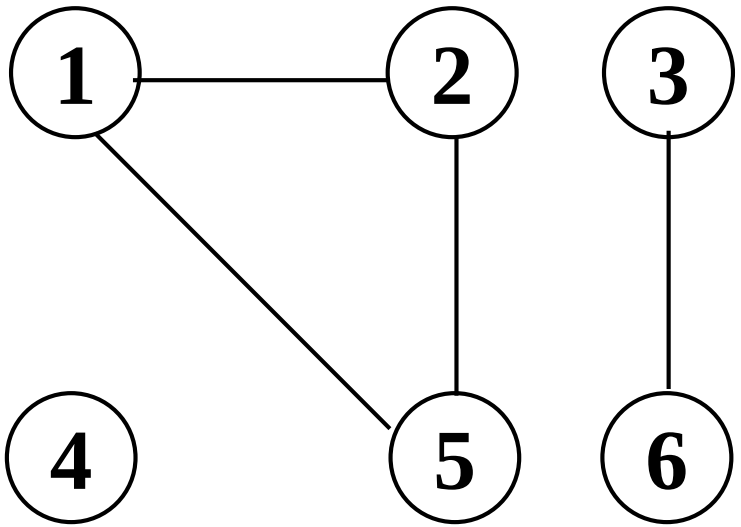




Kaaludeta graafi esitus naabrusmaatriksina

$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(1,5),(2,5),(3,6)\}$



A

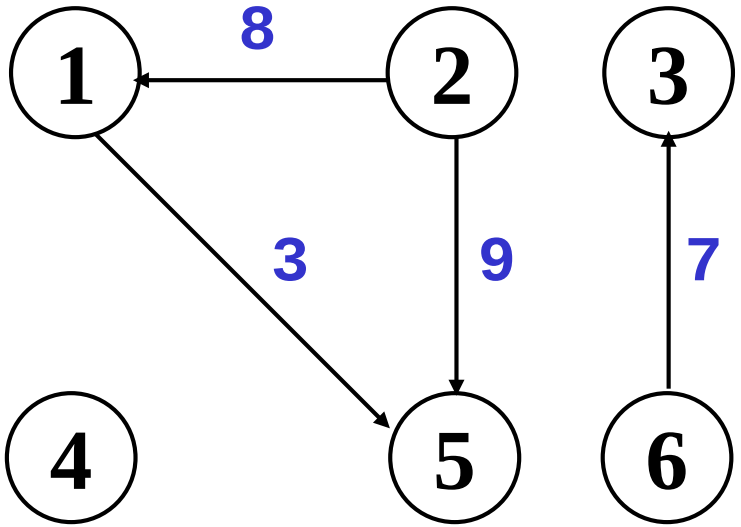
	1	2	3	4	5	6
1		1			1	
2	1				1	
3						1
4						
5	1	1				
6			1			

Suunata graafidel, $A = A^T$

Suuna ja kaaludega graafi esitus naabrusmaatriksina

$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(1,5),(2,5),(3,6)\}$



A

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

Naabruslist ja naabrusmaatriks

Naabruslist

- Sobilik hõredate graafide esitamiseks
  - Mälu $\Theta(|V| + |E|)$
  - Tipu kõigi naabrite leidmine on proportsionaalne naabrite arvuga
  -
- Tippudevahelise serva kontroll
  - $\Theta(|V|)$

Naabrusmaatriks

- Võib olla ebaefektiivne hõredate graafide korral
  - Mälu $\Theta(|V|^2)$
  - Tipu kõigi naabrite leidmine on proportsionaalne tippude arvuga
  - Efektiivne mälukasutus täieliku graafi korral
- Efektiivne kahe tipu vahelise serva kontrolliks
  - $\Theta(1)$

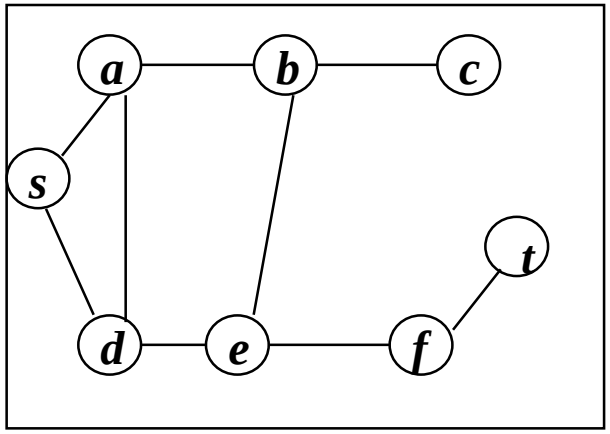
Graafi läbimine (otsing)

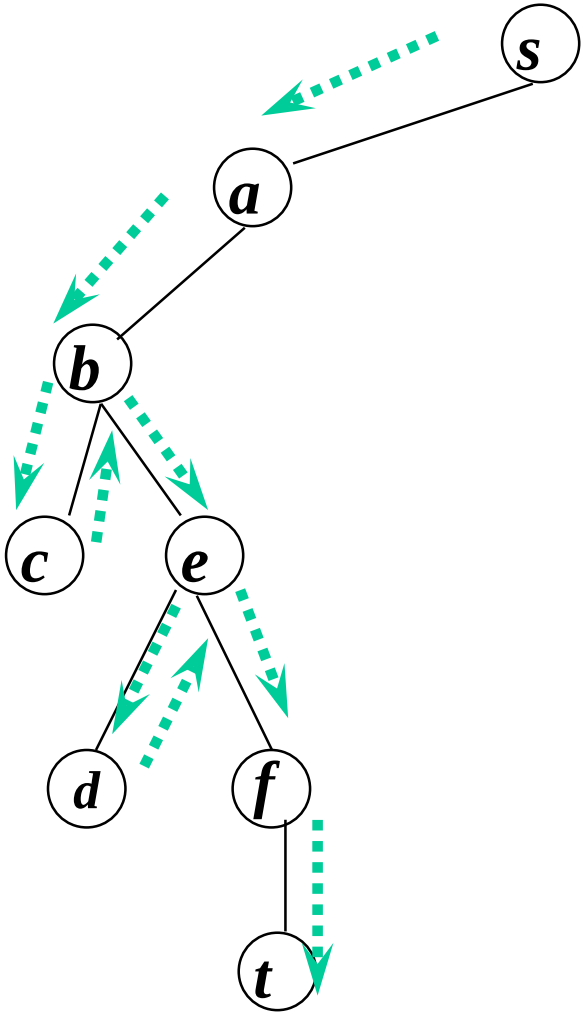
- Graafi läbimise (otsingu) eesmärgiks on läbida kõik tipud üks kord.

  Tipu läbimine võib tegelikus algoritmis tähendada arvutusi tipuga seotud andmetega vms. Abstraktselt tähistame seda funktsiooniga *visit(v)*

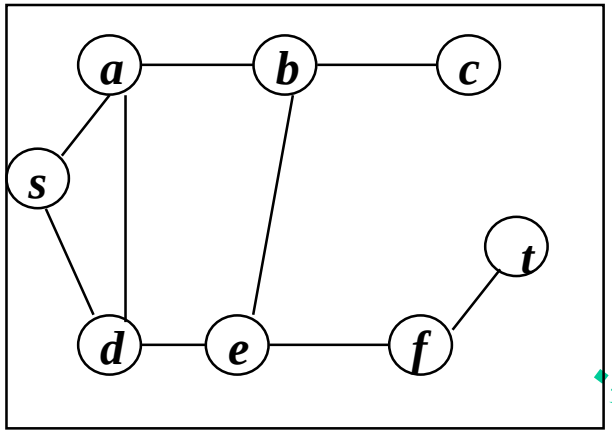
- Graafi tippe saab läbida erinevas järjekorras. Klassikaliselt eristatakse kahete järjekorda:
  - sügavuti otsing - liigutakse mööda servi nii “kaugele” kui saab
  - laiuti otsing - kõik sama “kauged” tipud läbitakse järjest

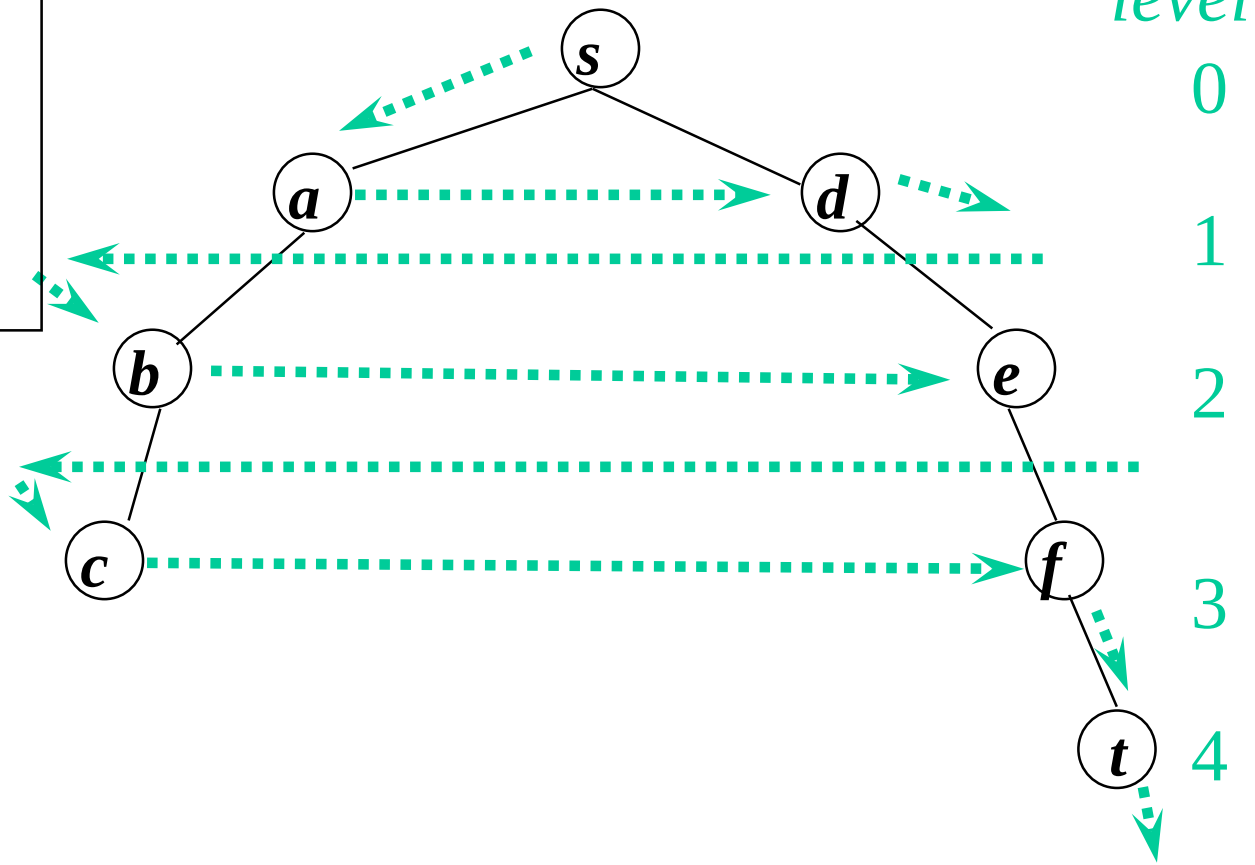
Sügavuti läbimine



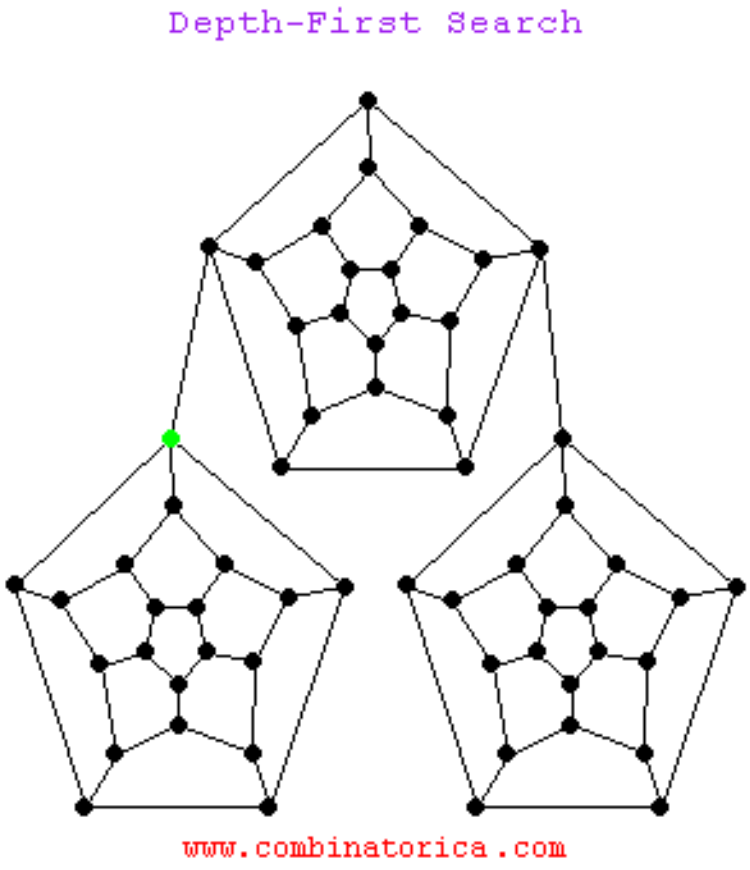


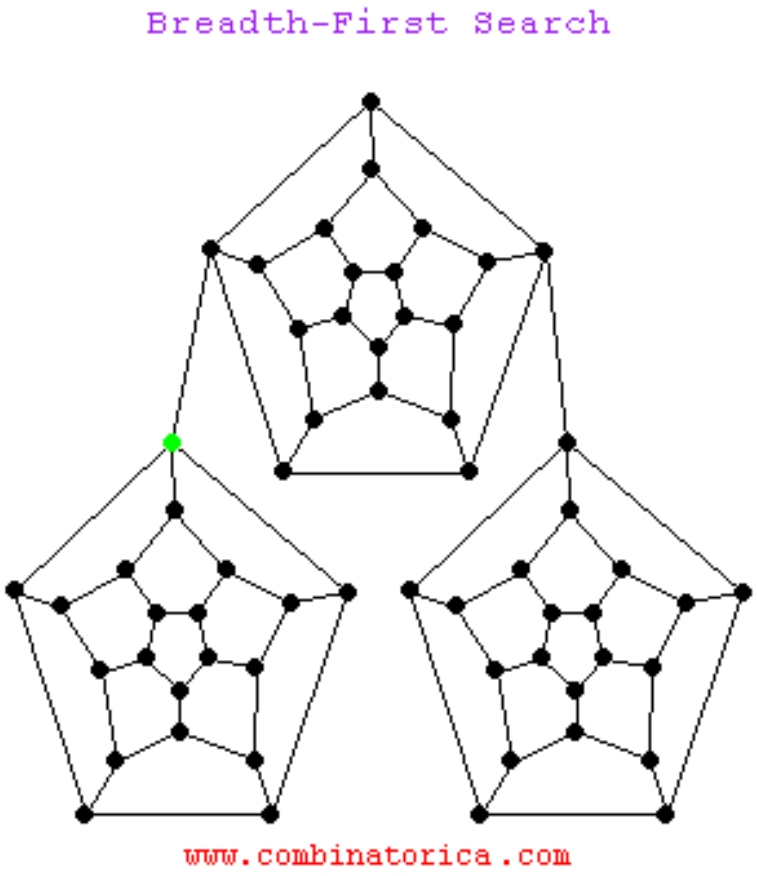
Laiuti läbimine





Läbimine sügavuti (DFS) ja laiuti (BFS)





Laiuti läbimise (BFS) algoritmi mõisteid ja omadusi

- Tipud märgendatakse ühega kolmest märgendist:
  - *visited (black)*: tipp ja selle naabrid on läbitud.
  - *current (grey)*: tipp on otsingu frondil.
  - *not_visited (white)*: tipuni pole veel jõutud.
- Läbitakse kauguse tasemete kaupa.
  dist(*v*) - kaugus juurtipust
- Läbimisel moodustatakse puu, mille juureks on esimene tipp.
  parent(*v*) - tipu *v* vanem läbimise puus
  - saadud puu esitab lühimaid teid juurtipust (servade arvu mõttes)
  - laiuti otsingut saab kasutada lühimate teede otsinguks
- Otsimisfronti *current* tippe hoitakse järjekorras Q

Laiuti läbimise (BFS) algoritm

BFS(*G*, *s*)

for all vertices *u* in *V* – {*s*} **do**

label[*u*] = *not_visited*; *dist*[*u*] = ∞; *parent*[*u*] = ***null***

label[*s*] = *current*; *dist*[*s*] = 0; *parent*[*s*] = ***null***

EnQueue(*Q*,*s*)

while *Q* is not empty **do**

u = DeQueue(*Q*)

for each *v* that is a neighbor of *u* **do**

if *label*[*v*] == *not_visited* **then** *label*[*v*] = *current*

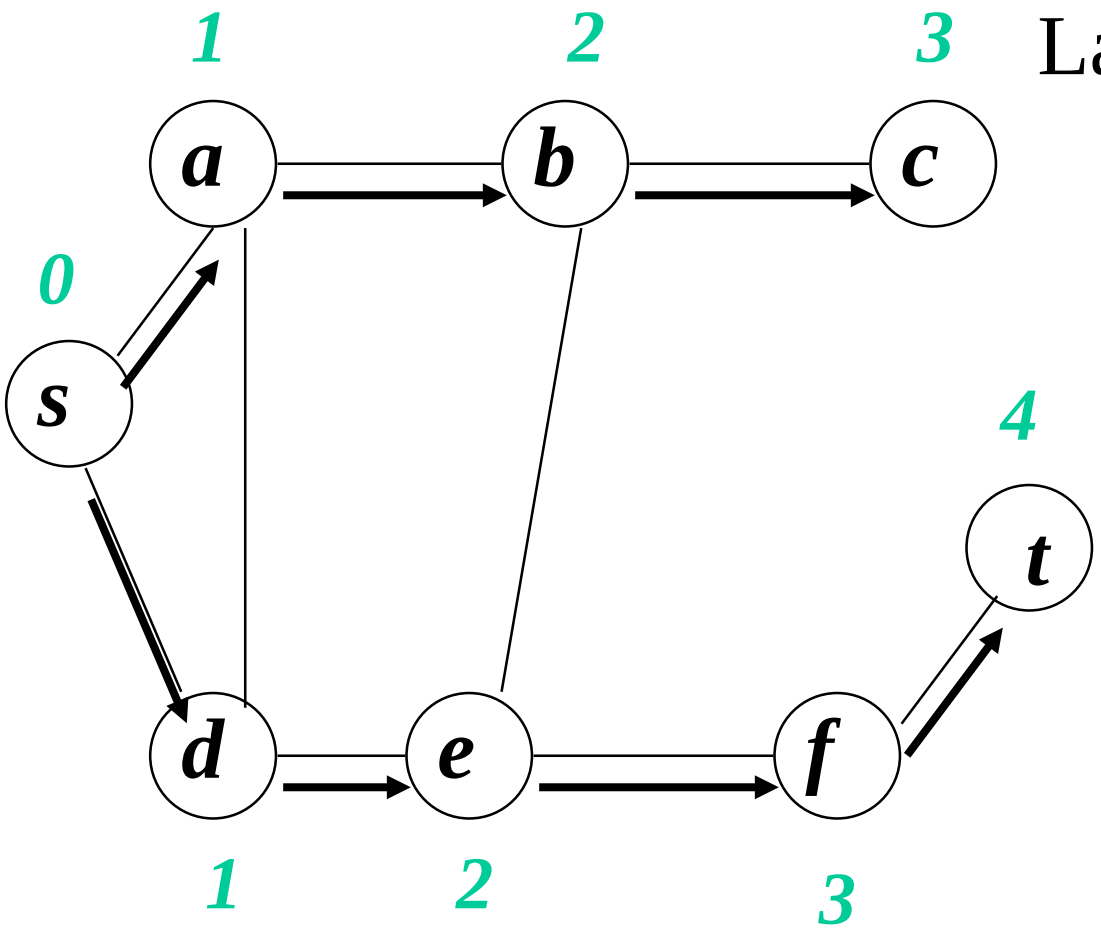
dist[*v*] = *dist*[*u*] + 1; *parent*[*v*] = *u*

EnQueue(*Q*,*v*)

label[*u*] = *visited*

O(|V| + |E|)

Laiuti läbimise (BFS) näide



Laiuti läbimise puu

Sügavuti läbimise (DFS) algoritmi mõisteid ja omadusi

- Tipud märgendatakse ühega kolmest märgendist:
  - *visited (black)*: tipp ja selle naabrid on läbitud.
  - *current (grey)*: tipp on otsingu frondil.
  - *not_visited (white)*: tipuni pole veel jõutud.
- Läbimisel moodustatakse puu (mets), mille juureks on esimene tipp.
  parent(*v*) - tipu *v* vanem läbimise puus
- Ei eelda, et graaf on sidus. Välise funktsiooni DFS tsükkel läbib kõik läbimata tipud. Mittesidusa graafi läbimisel saadakse tulemuseks *mets.*
- Tulemuseks saadav *metsa* (puude hulk) tippude järjestus sõltub naabrite valimise järjestusest

Sügavuti läbimise (DFS) rekursiivne algoritm

DFS(*G*, *s*)

for all vertices *u* in *V* **do**

label[*u*] = *not_visited; parent*[*u*] = ***null;***

for each vertex *u* in *V* **do**

if *label*[*u*] = *not_visited* **then**

DFS-Visit(*u*)

DFS-Visit(*u*)

label[*u*] = *current; time = time + 1; s*[*u*] = *time*

for each *v* that is a neighbor of *u* **do**

if *label*[*v*] == *not_visited* **then**

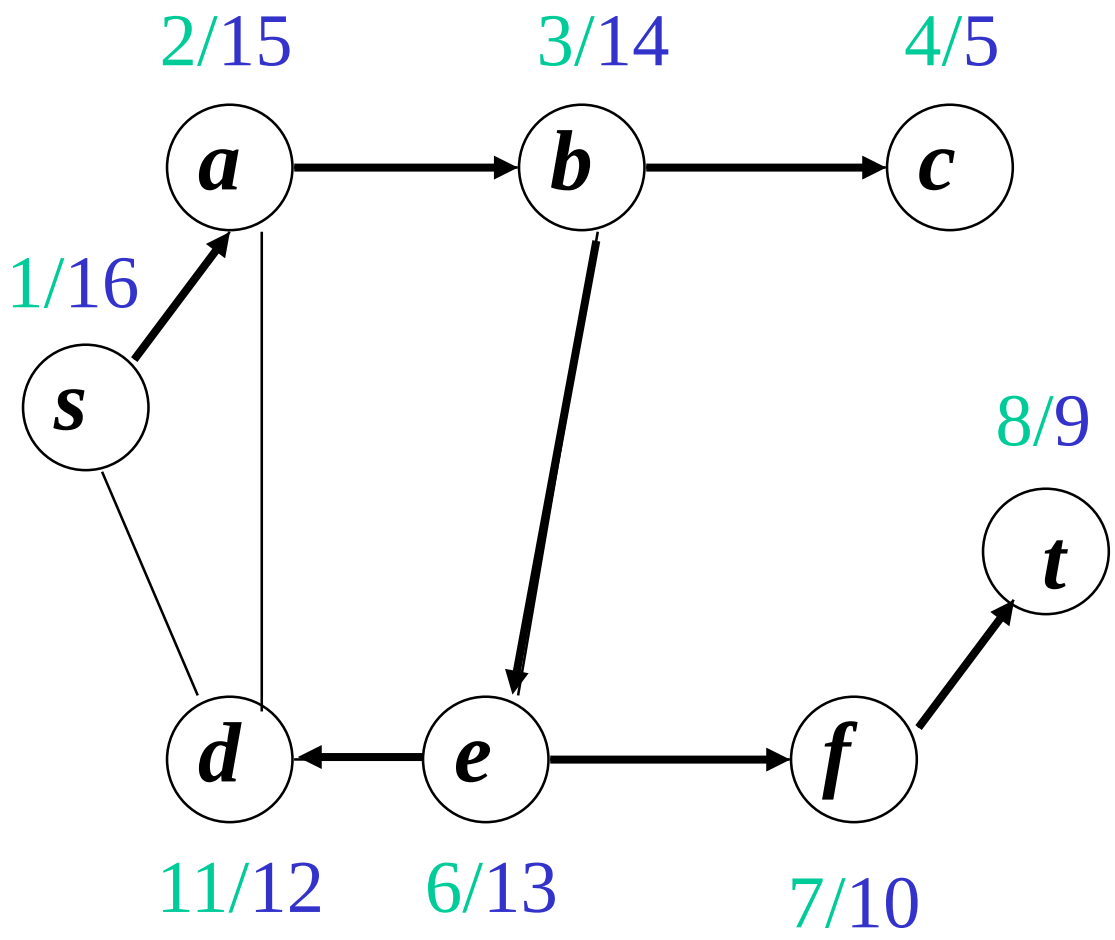
parent[*v*] = *u;*

DFS-Visit(*v*)

label[*u*] = *visited; time = time + 1; f*[*u*] = *time*

O(|V| + |E|)

DFS näide



sügavuti läbimise puu

märgendamise järjekord: current/visited

Sügavuti läbimise algoritm magasini (*stack*) abil

DFS(*G*, *s*)

for all vertices *u* in *V* – {*s*} **do**

label[*u*] = *not_visited*; *dist*[*u*] = ∞; *parent*[*u*] = ***null***

label[*s*] = *current*; *dist*[*s*] = 0; *parent*[*s*] = ***null***

Push(*S*,*s*)

while *S* is not empty **do**

u = Pop(*S*)

for each *v* that is a neighbor of *u* **do**

if *label*[*v*] == *not_visited* **then** *label*[*v*] = *current*

dist[*v*] = *dist*[*u*] + 1; *parent*[*v*] = *u*

Push(*S*,*v*)

label[*u*] = *visited*

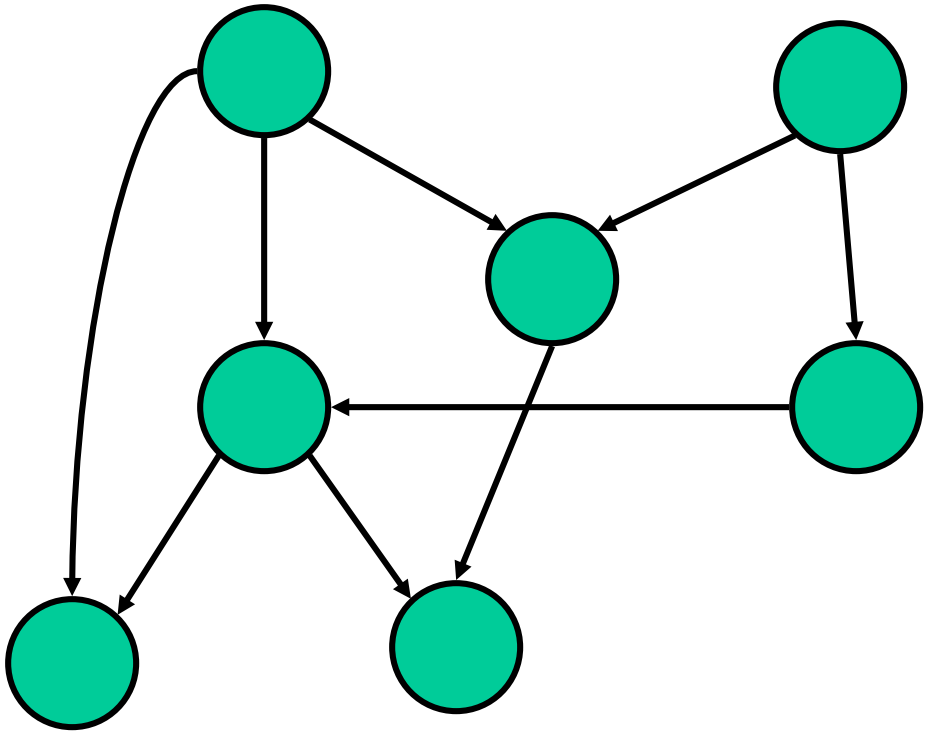
O(|V| + |E|)

Graafi läbimise algoritmid

- Laiuti läbimise algoritmi võib pidada üldiseks läbimise algoritmiks
  - seda saab modifitseerida sügavuti otisinguks andmestruktuuri asendamisega - magasin järjekorra asemel
  - laiuti läbimist toetav andmestruktuur Q kujutab endast otsimisfronti. Frondist liikmete väljastamise järjekord määrab läbimisstrateegia
- Sügavuti läbimise esimeses (rekursiivses) versioonis moodustatakse magasin peidetult.
  - kompilaator lisab rekursiivsed väljakutse magasini
  - nimetatakse ka tagasivõtmisega (*backtracking*) algoritmiks

Suunaga tsükliteta graafid
Directed Acyclic Graphs - DAGs

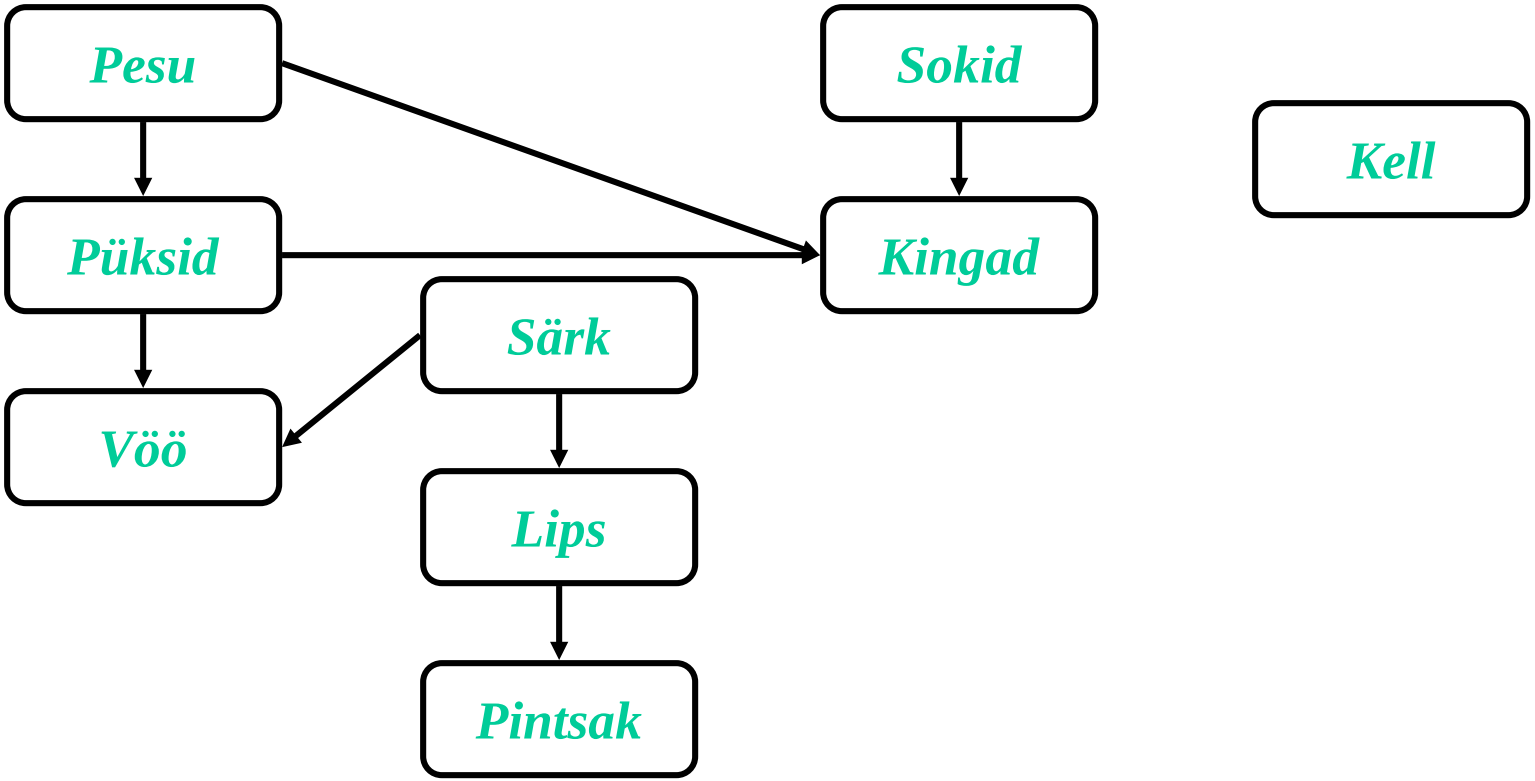
- DAG on suunatud tsükliteta graaf
- Esitab *osalist järjestust*

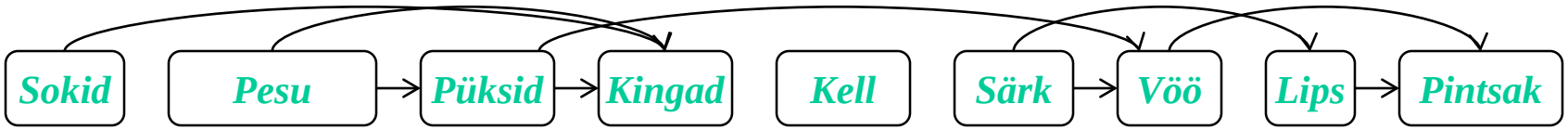


Topoloogiline sorteerimine

- DAGi topoloogiline sorteerimine:
  - Annab DAG *G* tippude lineaarse järjestuse, nii et tipp *u* on järjestuses eespool kui *v,* kui esineb serv $(u, v) \in G$.
- Kasutatakse näiteks tööde järjestuse leidmiseks
- Praktiline näide: riidesse panemine

Riietumine


Pesu
Sokid
Kell
Püksid
Kingad
Särk
Vöö
Lips
Pintsak


Sokid
Pesu
Püksid
Kingad
Kell
Särk
Vöö
Lips
Pintsak

Topoloogilise sorteerimise algoritm

```
Topological-Sort()
{
    Run modified DFS
        Add vertex to the list when it gets label visited
    Reverse the list
}
```

- Keerukus: O(V+E)
- Korrektsuse tõestuse idee: tipp saab *visited* alles pärast kõigi temast saavutatavate tippude (naabrite) *visited* märgendi saamist
  - Naaber oli juba enne tippu jõudmist *visited*
  - Naaber läbiti sügavuti läbimisega tippu jõudes

Sügavuti läbimise (DFS) rekursiivne algoritm

DFS(*G*, *s*)

for all vertices *u* in *V* **do**

label[*u*] = *not_visited; parent*[*u*] = ***null;***

for each vertex *u* in *V* **do**

if *label*[*u*] = *not_visited* **then**

DFS-Visit(*u*)

DFS-Visit(*u*)

label[*u*] = *current; time = time + 1; s*[*u*] = *time*

for each *v* that is a neighbor of *u* **do**

if *label*[*v*] == *not_visited* **then**

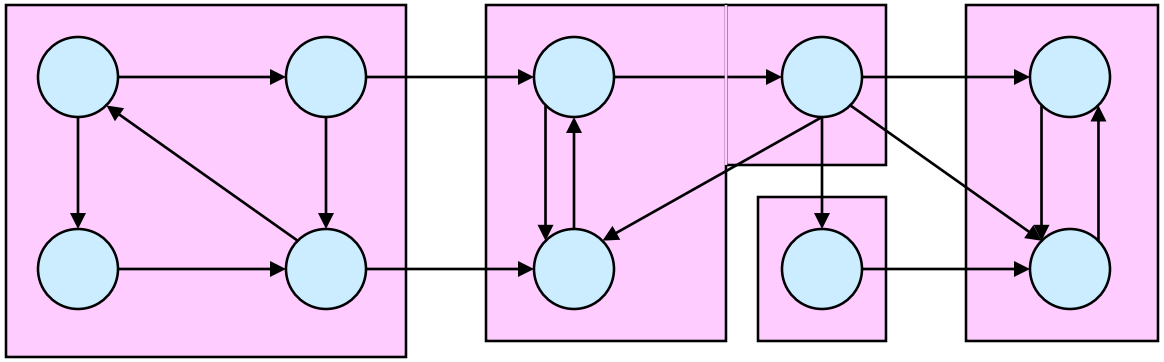
parent[*v*] = *u;*

DFS-Visit(*v*)

label[*u*] = *visited; time = time + 1; f*[*u*] = *time*

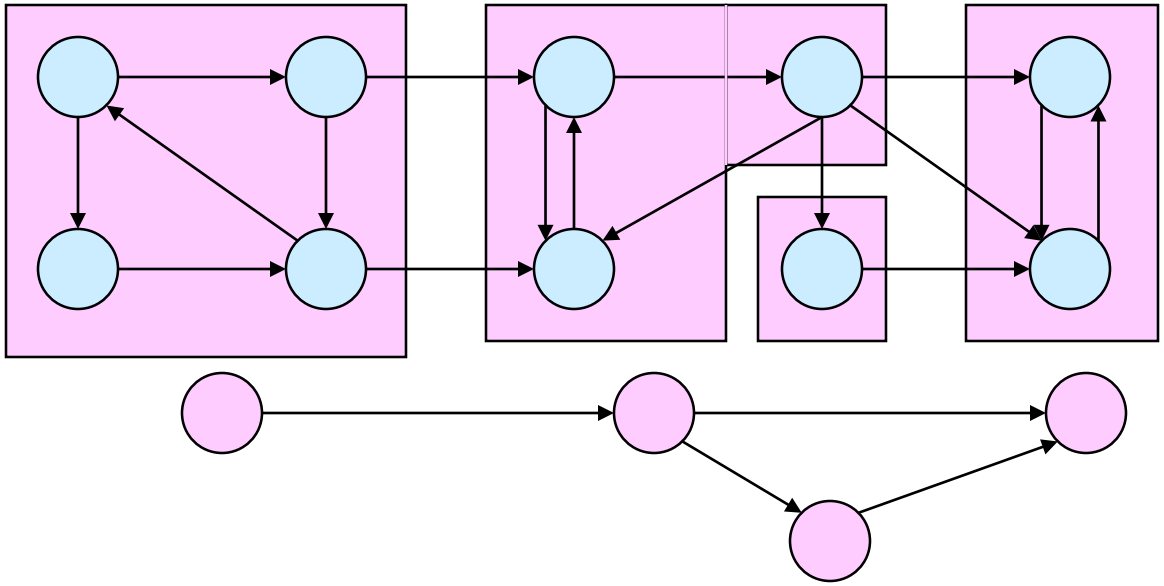
Tugevalt sidusad komponendid *Strongly Connected Component* (*SCC*)

- Suunatud graaf on tugevalt sidus, kui igast tipust on olemas tee igasse tippu
- Suunatud *graafi tugevalt sidus komponent* (SCC) on maksimaalne tippude hulk *C,* nii et iga tippude paari $u,v \in C$ jaoks on olemas tee $u \to v$ ja $v \to u$.
- Näiteks üksteise postitusi *like*-vad sõbrad.



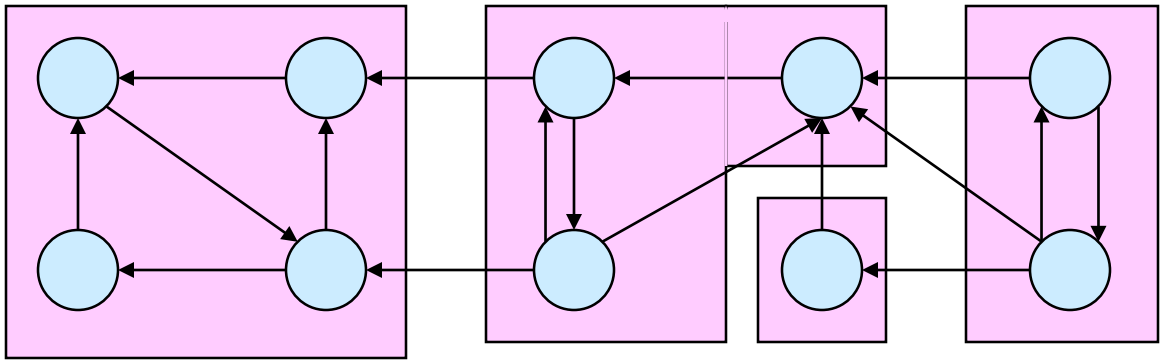
Komponentgraaf

- Komponentgraafis on tipp graafi iga SCC kohta
- Komponentgraafis on serv $U \to V$, kui graafi komponendi U mõnest tipust läheb serv komponendi V mõne tipuni
- Komponentgraaf on DAG



Transponeeritud graaf

- Transponeeritud graaf on graaf, kus servade suunad on ümber pööratud
  $G^T = (V, E^T), E^T = \{(u, v) : (v, u) \in E\}$
- Graafil ja tema transponeeritud graafil on samad tugevalt sidusad komponendid



SCC leidmise algoritm

SCC(G)

Call DFS(G) to compute finishing times $f[u]$ for all u

Compute G^T

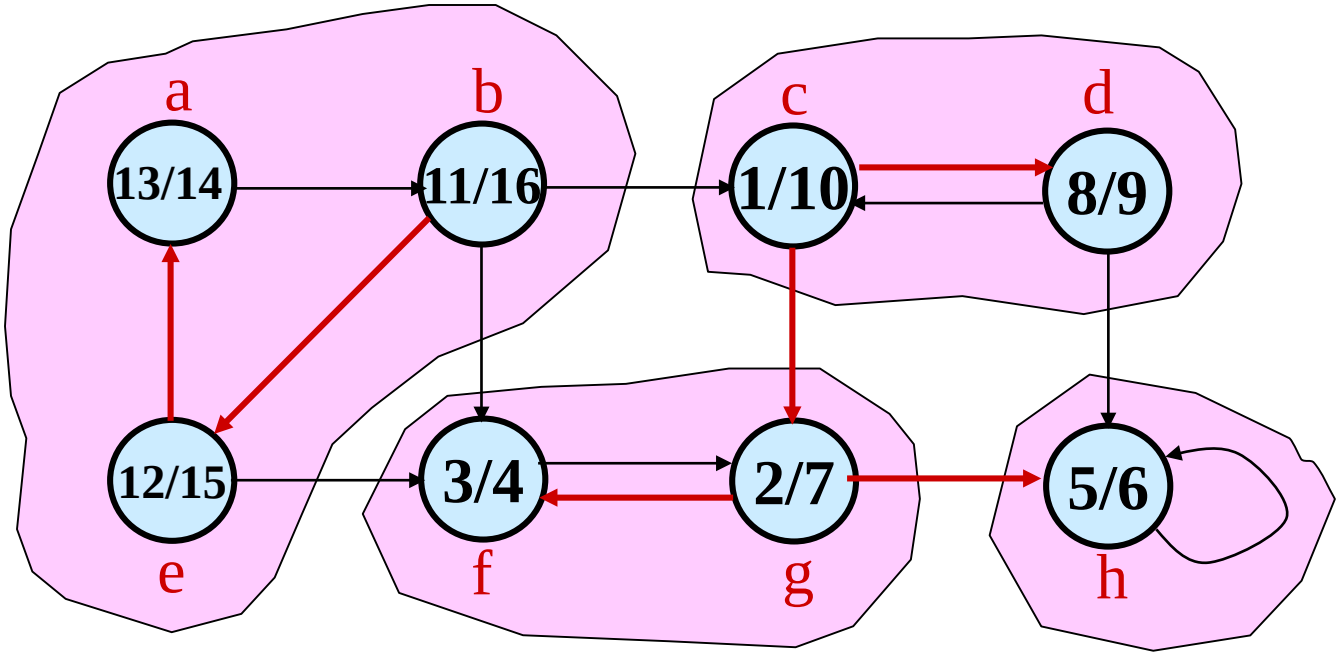
Call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)

Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

- Keerukus: O(V + E), kui graaf on esitatud naabruslistidena
- On olemas SCC algoritme (Tarjani algoritm jt), mis kasutavad ühekordset täiendatud sügavuti otsingut

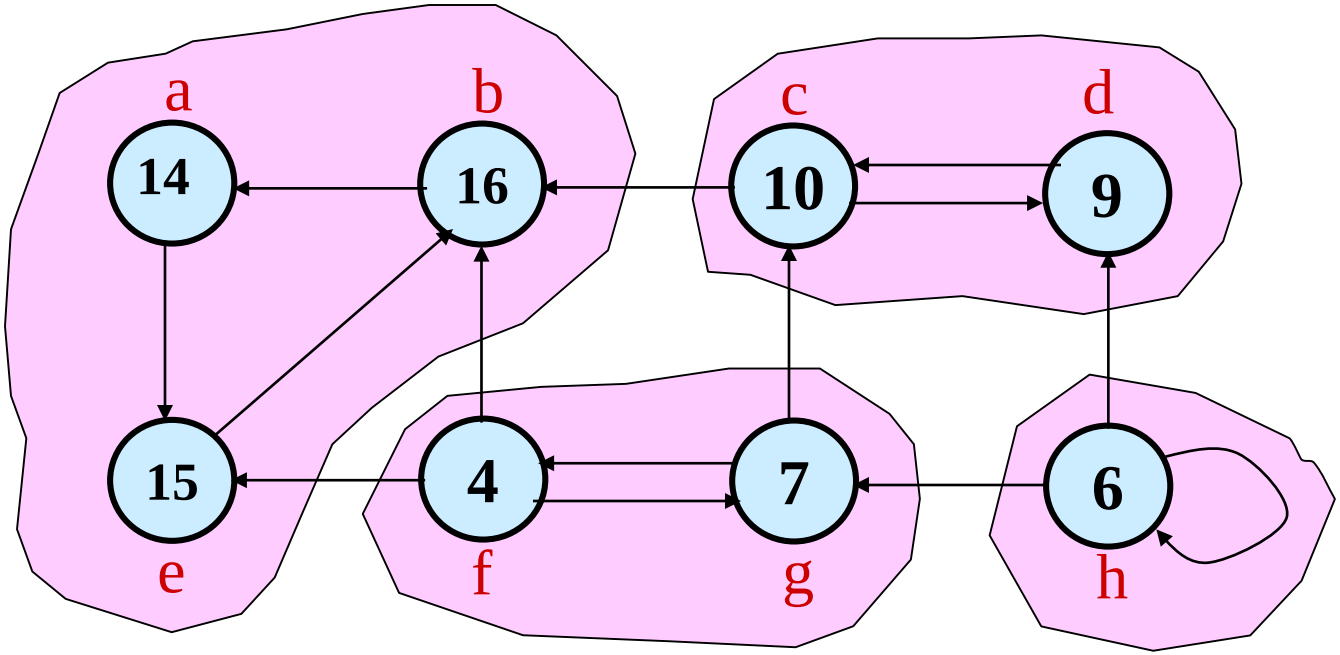
SCC leidmise näide

Leiame sügavuti otsingu puud (metsa) originaalgraafil ja vastavad tippude otsingu lõpetamise ajad. Otsing algas tipust *c*



SCC leidmise näide

Transponeerime graafi
Tippudes on sügavuti otsingu lõpetamise järjekord
Iga SCC piire ületava serva sihttipp on suurema numbriga kui lähtetipp!



SCC leidmise näide

Teeme sügavuti otsingu transponeeritud graafil numbrite kahanemise järjekorras (algab tipust *b*)

{b, a, e} {c, d} {g, f} {h}