Monta indeksiä

- Olemme käsitelleet yksinkertaistettua tilannetta, jossa puun solmuihin ei ole talletettu muuta kuin avaimen arvo

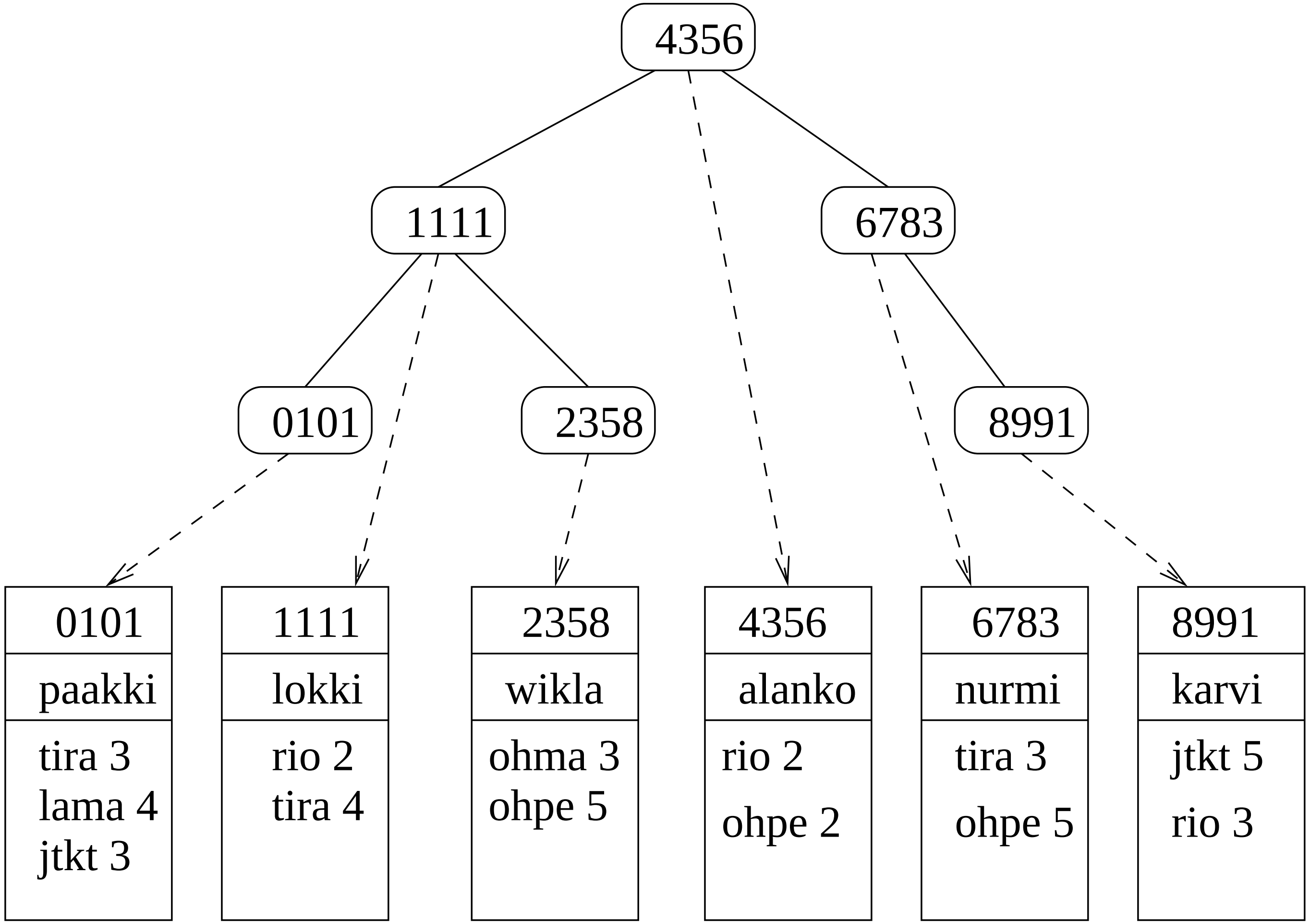
- Jos organisoitavaa dataa on paljon, paras ratkaisu on tallettaa muu data omana olionaan ja lisätä puusolmuihin avaimen lisäksi viite muun datan tallettavaan olioon

- Puusolmu muodostuu tällöin kentistä:

key	talletettu avain
data	viite avaimeen liittyvän tiedon tallettavaan olioon
left	viite vasempaan lapseen
right	viite oikeaan lapseen
parent	viite vanhempaan

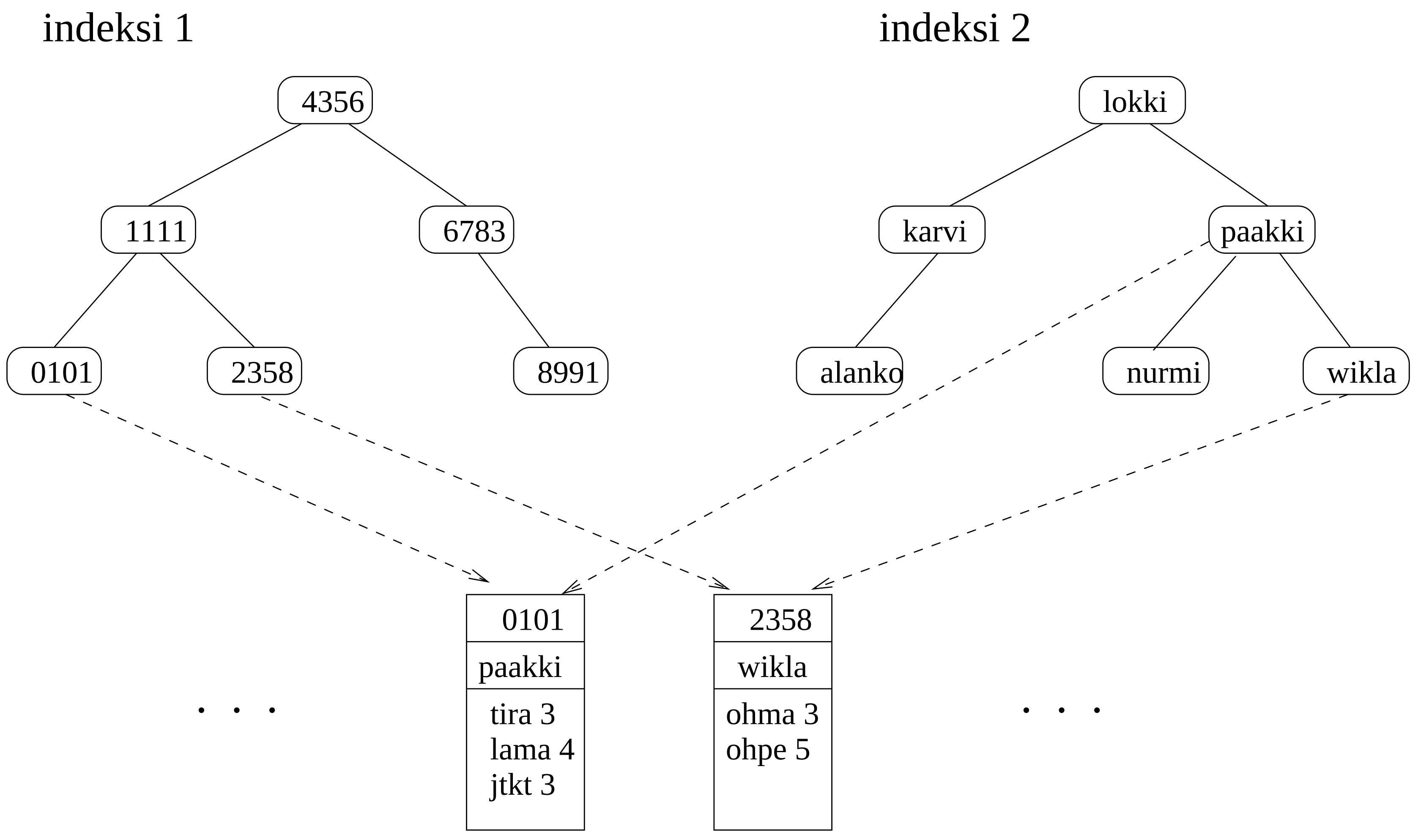
- Näin puu toimii indeksirakenteena, jonka avulla talletettuun tietoon on mahdollista tehdä nopeita hakuja avaimen perusteella

- Esim. opiskelijarekisteri, jossa binäärihakupuu toimii indeksirakenteena opiskelijanumeron suhteen:



- Nyt siis opiskelijan tietojen haku opiskelijanumeron perusteella on nopeaa

- Entä jos haluamme nopeat haut myös nimen perusteella?
- Lisätään samalle datalle toinen indeksirakenne, joka mahdollistaa nopeat haut nimeen perustuen



- Javan valmiista tietorakennetoteutuksista `TreeSet` ja `TreeMap` perustuvat tasapainoisiin binäärihakupuihin

- `TreeSet`:iin talletetaan olioita, joille on määritelty suuruusjärjestys (Tämä tapahtuu joko toteuttamalla ns. `Comparable`-rajapinta tai määrittelemällä järjestyksen antava metodi)

- Oliot on talletettu `TreeSet`:iin niille määritellyssä järjestyksessä, ja olioiden läpikäynti järjestyksessä on nopeaa

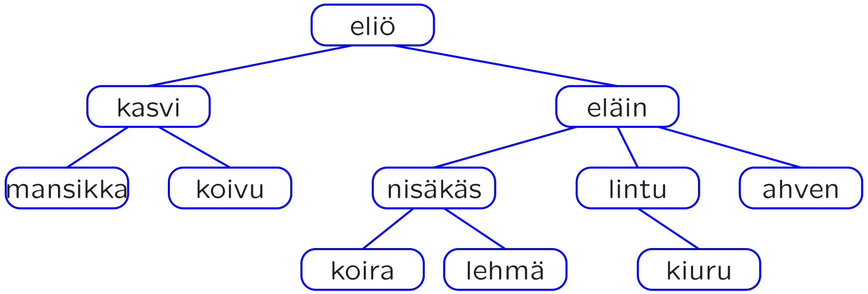
- `TreeSet`:issä olevia olioita ei pysty hakemaan nopeasti mihinkään olion attribuuttiin (esim. opiskelijanumero) perustuen. Nopea, eli $\mathcal{O}(\log n)$ suhteessa talletettujen olioiden määrään n, on ainoastaan testi onko tietty olio `TreeSet`:issä

- `TreeMap` taas toimii edellisten sivujen indeksirakenteiden tapaan, eli `TreeMap`:iin talletetaan avain-dataolio -pareja, ja etsintä avaimeen perustuen on tehokas

- Esim. edellisten sivujen opintorekisteriesimerkin toteutus onnistuisi helposti kahden `TreeMap`:in avulla, toisessa olisi avaimena opiskelijanumero ja toisessa opiskelijan nimi, molemmissa opiskelijan tiedot talletettaisiin erilliseen olioon, joka löytyisi nopeasti sekä nimen että opiskelijanumeron perusteella

Yleisen puun talletus ja läpikäynti

- Kuten jo puuluvun alussa mainittiin, on puille monenlaista käyttöä tietojenkäsittelyssä
- Puita käytetään esim. yleisesti erilaisten hierarkioiden esittämiseen tietojenkäsittelyssä ja muualla:



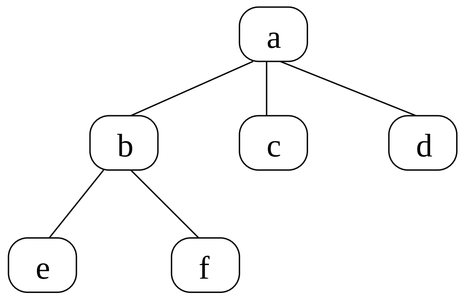
- Muutaman sivun päästä tutustumme puiden käyttöön ongelmanratkaisussa
- Kaikki hyödylliset puut eivät siis suinkaan ole binääripuita tai hakupuita
- Miten voimme tallettaa yleisen puun?

- Jos tiedämme mikä on solmun maksimihaarautumisaste, voimme tallettaa solmuun viitteet kaikkiin mahdollisiin lapsiin

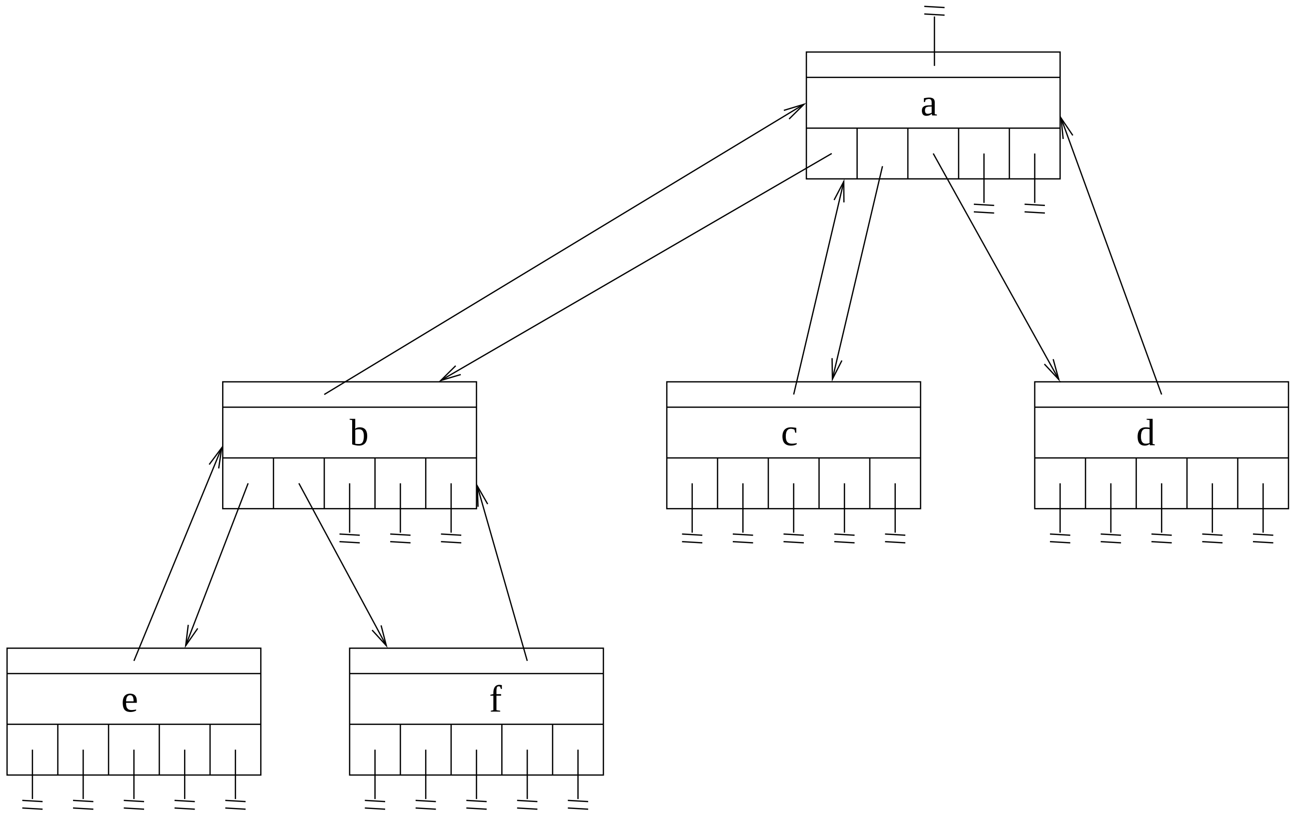
- Eli puusolmu muodostuu tällöin kentistä:

key	talletettu avain
$c1$	viite 1. lapseen
$c2$	viite 2. lapseen
...	
ck	viite k:nteen lapseen
p	viite vanhempaan

- Esimerkki allaolevan puun tallettamisesta seuraavalla sivulla



- Puu talletettuna käyttäen puusolmuja joissa haarautumisaste on 5



- Huomaamme että rakenne tuhlaa paljon muistia tarpeettomiin linkkikenttiin

- Toisaalta voi käydä myös niin että johonkin solmuun tulisikin enemmän lapsia kuin 5

- Esim. Javassa voisimme tallettaa solmun lapset `ArrayList`:iin, joka siis käytännössä on vaihtuvamittainen taulukko

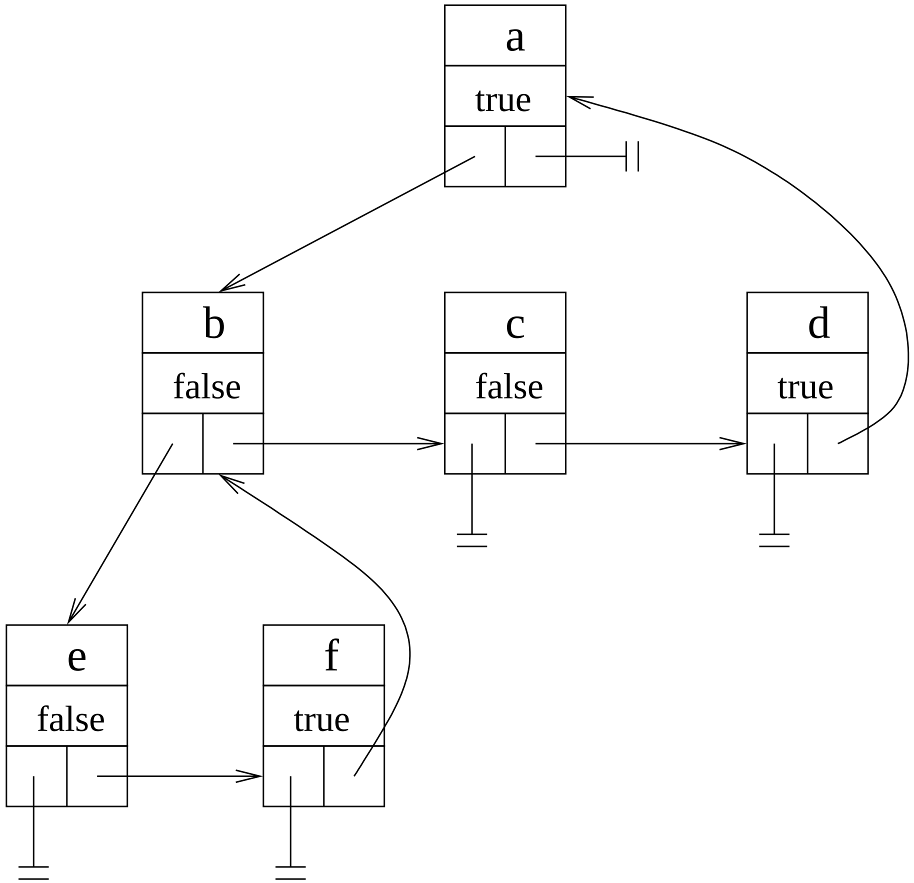
- Näin saataisiin lapsimäärästä joustava, ja turhaa tilaa ei varattaisi kohtuuttoman paljoa

- Muistin käytön kannalta parempi ratkaisu yleisen puun tallettamiseen on kuitenkin seuraava

- Puusolmun kentät

key	talletettu avain
$last$	bitti jonka arvo on true, jos kyseessä on sisaruksista viimeinen
$child$	viite 1. lapseen
$next$	viite seuraavaan sisarukseen (jos $last$ = false), tai vanhempaan (jos $last$ = true)

- Esimerkkipuumme talletettaisiin seuraavasti:



- Muistia ei tuhlaudu turhiin linkkikenttiin ja toisaalta puun haarautumisaste ei ole rajoitettu

- Solmusta x päästään vanhempaan kulkemalla $next$-linkkejä, kunnes on ohitettu sisarus jolla $last$ = true

```
parent(x)
    while x.last == false
        x = x.next
    return x.next
```

- Viite solmun x ensimmäiseen lapseen on helppo selvittää

```
firstchild(x)
    return x.child
```

- Muut lapset saadaan kutsumalla toistuvasti seuraavaa operaatiota parametrina edellinen löydetty lapsi y

```
nextchild(y)
    if y.last == true return NIL
    else return y.next
```

- Viimeisen lapsen jälkeen operaatio palauttaa NIL

- Binäärihakupuun yhteydessä saimme tulostettua puun solmut suuruusjärjestyksessä käymällä puun läpi sisäjärjestyksessä, eli ensin vasen lapsi, sitten solmu itse ja lopulta oikea lapsi

- Yleisten puiden kohdalla mielekkäät läpikäyntitavat ovat esijärjestys ja jälkijärjestys

- Esijärjestyksessä käsittelemme ensin solmun ja tämän jälkeen lapset

- Esimerkkipuumme solmut esijärjestyksessä lueteltuna: a, b, e, f, c, d

- Algoritmina

```
preorder-tree-walk(x)
    print x.key
    y = firstchild(x)
    while y ≠ NIL
        preorder-tree-walk(y)
        y = nextchild(y)
```

- Kutsu **preorder-tree-walk**($T.root$) tulostaa nyt puun sisällön esijärjestyksessä. Huom: operaatio ei toimi tyhjälle puulle!

- Jälkijärjestyksessä käsittelemme ensin lapset ja tämän jälkeen solmun itsensä
- Esimerkkipuumme solmut jälkijärjestyksessä lueteltuna: e, f, b, c, d, a
- Algoritmina

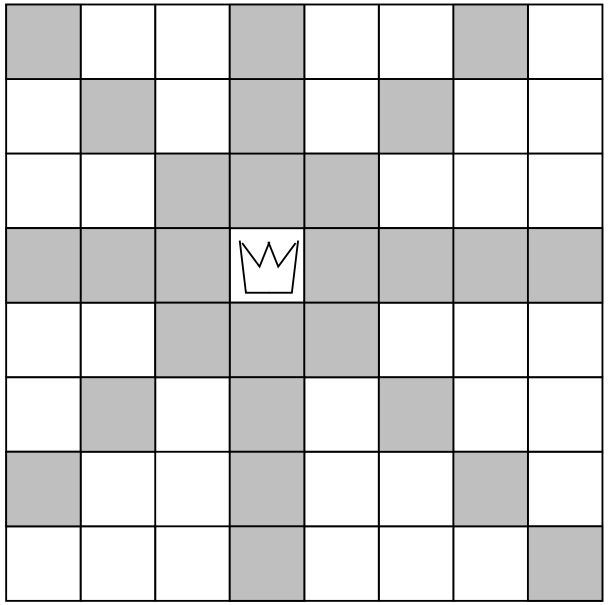
```
postorder-tree-walk(x)
    y = firstchild(x)
    while y ≠ NIL
        postorder-tree-walk(y)
        y = nextchild(y)
    print x.key
```

- Toki muitakin tapoja puun läpikäynnille on, esim. leveyssuuntainen läpikäynti missä puun alkiot käydään läpi taso kerrallaan, alkaen juuresta
- Esimerkkipuumme solmut leveyssuuntaisesti lueteltuna: a, b, c, d, e, f

```
levelorder-tree-walk(x)
    Q = tyhjä solmujono
    enqueue(Q, T.root)
    while not empty(Q)
        x = dequeue(Q)
        print x.key
        y = firstchild(x)
        while y ≠ NIL
            enqueue(Q,y)
            y = nextchild(y)
```

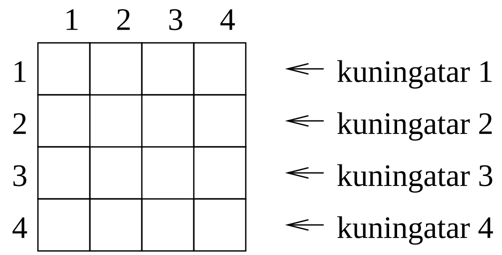
Puut ongelmanratkaisussa: kahdeksan kuningattaren ongelma

- Yksi puiden tärkeistä käyttötavoista on ongelmanratkaisussa tapahtuvan laskennan etenemisen kuvaaminen
- Kahdeksan kuningattaren ongelma: miten voimme sijoittaa shakkilaudalle 8 kuningatarta siten että ne eivät uhkaa toisiaan?
- Kuningatar uhkaa samalla rivillä, sarakkeella sekä diagonaalilla olevia ruutuja

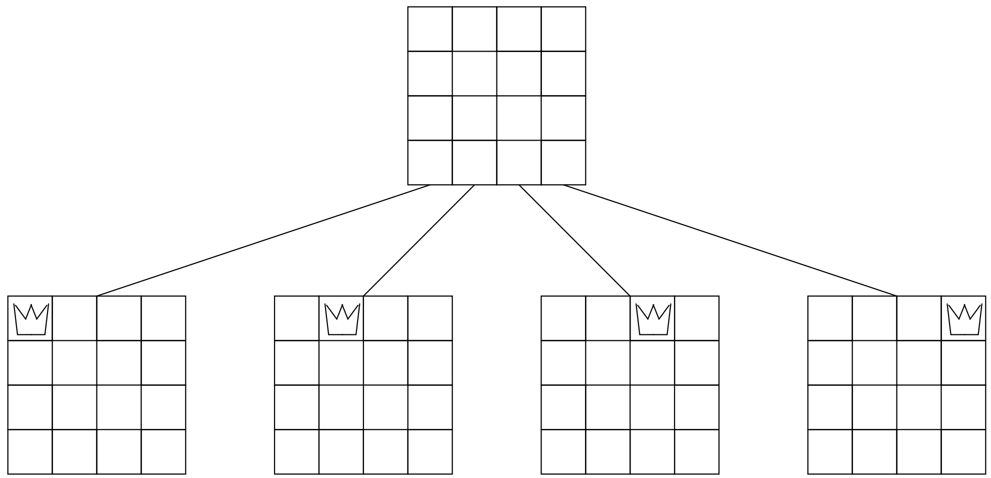


- Yleistetty version ongelmasta: miten saamme sijoitettua n kuningatarta $n \times n$ -kokoiselle shakkilaudalle

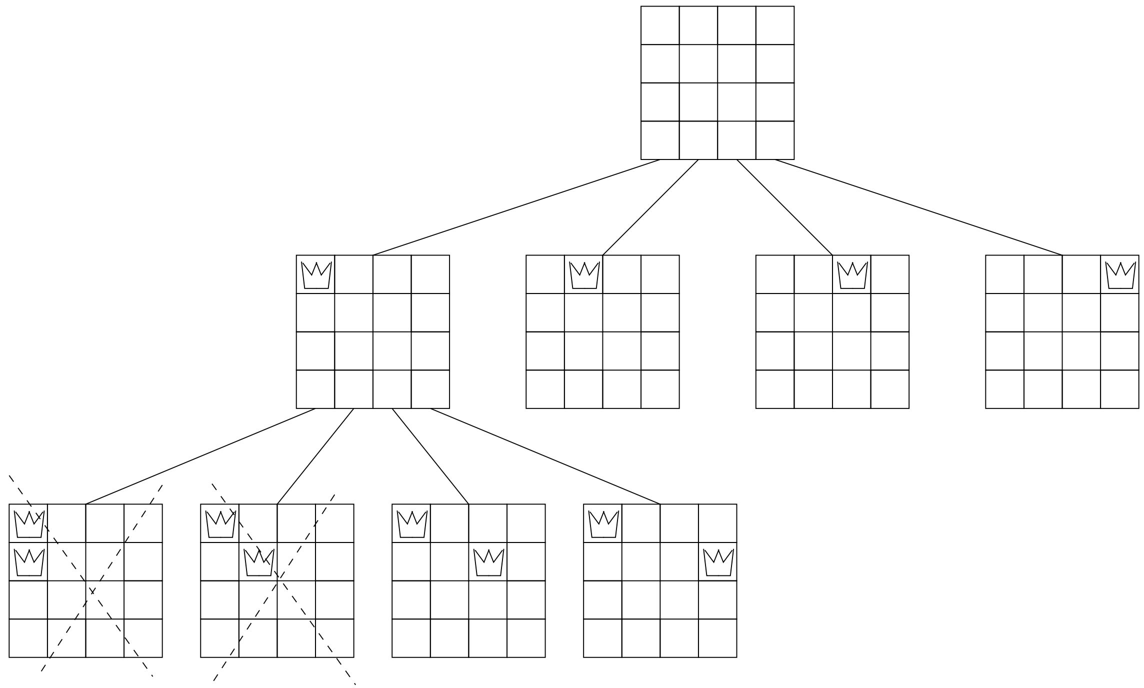
- Tarkastellaan ensin tapausta missä $n = 4$. selvästikin jokaisella rivillä täytyy olla tasan 1 kuningatar:



- Etsitään oikea kuningatarasetelma systemaattisesti
  - aloitetaan tyhjältä laudalta
  - tämän jälkeen asetetaan kuningatar riville 1
  - neljä eri mahdollisuutta:



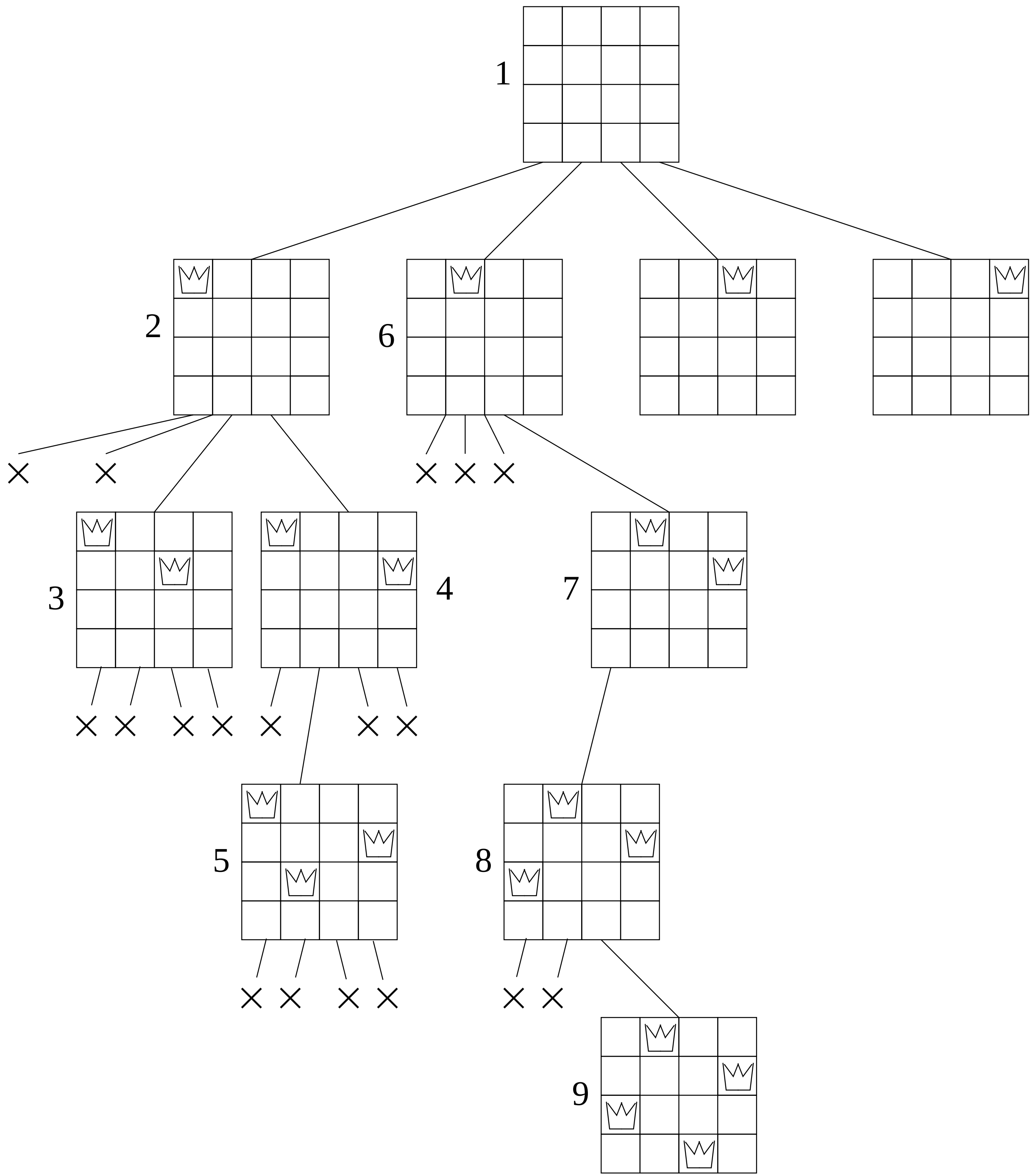
- Seuraavaksi tarkastellaan miten kuningattaret voidaan asettaa riville 2. Aloitetaan vasemmanpuoleisesta 1 rivin valinnasta



- Huomaamme että olemme muodostamassa puuta, joka kuvaa erilaisia ratkaisumahdollisuuksia

- Kaksi vasemmanpuoleisinta yritystä ovat tuhoon tuomittuja, eikä enää kannata tutkia mitä niissä haaroissa tapahtuu

- Seuraavalla sivulla ratkaisun löytymiseen asti piirretty ratkaisupuu


1
2
6
3
4
7
5
8
9

- Kuvassa puun solmut on numeroitu esijärjestyksessä, ja yhdeksäs solmu on siis ratkaisua vastaava pelitilanne

- Kun laudan koko n kasvaa, tulee puusta varsin suuri

- Huomionarvoista on kuitenkin se että koko puun ei tarvitse olla talletettuna muistiin

- Itseasiassa riittää että muistissa on ainoastaan reitti juuresta parhaillaan tutkittavaan solmuun

- Voimme etsiä ratkaisun n:n kuningattaren ongelmaan suorittamalla ratkaisupuun läpikäynnin esijärjestyksessä ilman että ratkaisupuuta on missään vaiheessa olemassa

- Talletetaan pelitilanne $n \times n$ -taulukkoon:
  - oletetaan että pelilautaa esittää $n \times n$ -taulukko $table$
  - jos pelilaudan kohdassa (x, y) on kuningatar, on $table[x, y] =$ true
  - muuten $table[x, y] =$ false

- Oletetaan että käytössä on metodi **check**($table$)
  - metodi palauttaa true jos sen parametrina saama pelitilanne on mahdollinen ratkaisu tai voidaan vielä täydentää ratkaisuksi
  - jos pelilaudalla on toisiaan uhkaavia kuningattaria, operaatio palauttaa false

- Valitaan ensin vakio n, eli pelilaudan koko on $n \times n$

- Aluksi laitetaan $n \times n$ taulukon $table$ kaikkien ruutujen arvoksi false, ja kutsutaan **putqueen**($table$,1)

- **putqueen**(table,row)

```
1   if check(table) == false
2       return
3   if row == n+1
4       print(table)
5       return
6   for x = 1 to n
7       table2 = luoKopio( table )
8       table2[x,row] = true
9       putqueen(table2,row+1)
```

- Operaation toiminta parametreilla ($table$, row):
  - operaatio tarkastaa ensin (rivi 1) edustaako $table$ pelilautaa mikä voi johtaa ratkaisuun tai on jo ratkaisu (rivi 3)
  - jos kyseessä on ratkaisu, tulostetaan pelilauta (rivit 3-5)
  - muussa tapauksessa tutkitaan kaikki tavat asettaa kuningatar riville row
  - luodaan uusi asetelma tauluun $table2$ ja rekursiivinen kutsu (rivi 9) tarkastaa johtaako tämä asetelma ratkaisuun

- Algoritmi käy läpi puun mikä ei ole missään vaiheessa rakennettuna muistiin, tällaista puuta sanotaan implisiittiseksi puuksi

- Jos puu olisi kokonaan muistissa, olisi sen koko valtava: $1 + n + n^2 + n^3 + \ldots + n^n$

- Koska nyt muistissa on korkeintaan puun korkeudellinen (eli n kpl) solmuja, on tilavaativuus $\mathcal{O}(n^3)$, sillä jokainen rekursiokutsu vaatii tilaa shakkilaudan verran eli $\mathcal{O}(n^2)$, tilavaativuus ei siis ole kohtuuton

- Aikavaativuus sen sijaan on suuri, sillä vaikka kaikkia solmuja ei tarvisekaan käydä läpi, kasvaa läpikäytävien solmujen määrä kuitenkin eksponentiaalisesti n:n suhteen

- Tälläisestä implisiittisen puun läpikäyntimenetelmästä käytetään nimitystä peruuttava etsintä (engl. backtracking): umpikujaan jouduttaessa palataan puussa sellaiseen ylempään solmuun, johon vielä liittyy kokeilemattomia vaihtoehtoja

- Esitetty algoritmi kuljettaa muodostettavaa kuningatarasetelmaa rekursiivisten kutsujen parametrina

- Taulukko kopioidaan jokaisen rekursiivisen kutsun yhteydessä rivillä 7

- $n \times n$-kokoisen taulukon kopiointiin kuluu aikaa $O(n^2)$, eli jokainen funktion rungon suoritus kuluttaa taulukkojen kopiointiin aikaa n kertaa $O(n^2)$, eli $O(n^3)$

- Taulukon kuljettaminen parametrina ei ole oikeastaan tarpeen: riittää että pidetään rakennuksen alla oleva kuningatarasetelma globaalina muuttujana olevassa taulukossa

- Oletetaan, että $table$ on kuten edellä, mutta kyseessä on globaali muuttuja, kuningatarasetelma löytyy kutsumalla seuraavaa funktiota parametrilla 1:

putqueenv2(row)
```
1   if check(table) == false
2       return
3   if row == n+1
4       print(table)
5       return
6   for x = 1 to n
7       table[x,row] = true
8       putqueenv2(row+1)
9       table[x,row] = false
```

- Funktiorungon yhden suorituksen aikavaativuus on nyt funktion **check** suoritusaika plus $O(n)$

- Algoritmin kokonaisaikavaativuus on siis solmujen lukumäärä kertaa funktiorungon suoritusaika

- Algoritmin tilavaativuus pienenee, sillä edellisessä versiossa jokainen rekursiivinen kutsu talletti oman kopionsa asetelmasta ja vei tilaa $O(n^2)$ ja koska rekursiivisia kutsuja voi olla kerrallaan menossa n kpl tilavaativuus oli $O(n^3)$

- Uudessa versiossa jokainen rekursiokutsu vie tilaa ainoastaan vakion verran, eli koko algoritmin tilavaativuus on $O(n)$

- Globaalien muuttujien käyttöä ei yleisesti pidetä kovin hyvänä ideana

- Jos käytössä on olio-ohjelmointikieli, voidaan "globaali" muuttuja esittää myös ohjelmistoteknisessä mielessä tyylikkäästi tekemällä globaalisti saatavilla olevasta datasta olion attribuutti

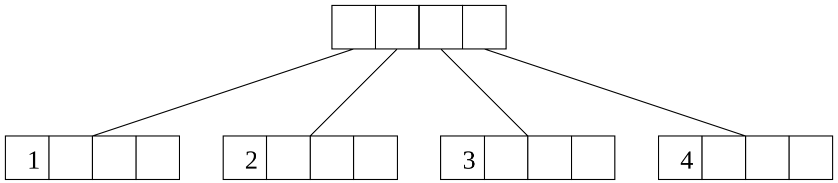
- Java-luonnos ratkaisusta seuraavalla sivulla

```
public class Queens{
   boolean[][] table;
   int n;

   public Queens(int n){ this.n = n; this.table = new boolean[n]; ... }

   private boolean check(){ ... } // ei parametria sillä näkee attribuutin table

   public void putQueen(int row){
       if ( check( ) == false ) return;
       if ( row = n+1 ) { tulosta( this.table ); return; }
       for ( int x=1; x<=n; x++ ) {
           this.table[row][x] = true;
           putQueen(row+1);
           this.table[row][x] = false;
       }
   }
}

public class PaaOhjelma{
   public static void main(String[] args) {
       Queens ongelma = new Queens(8);
       ongelma.putQueen(1);
   }
}
```

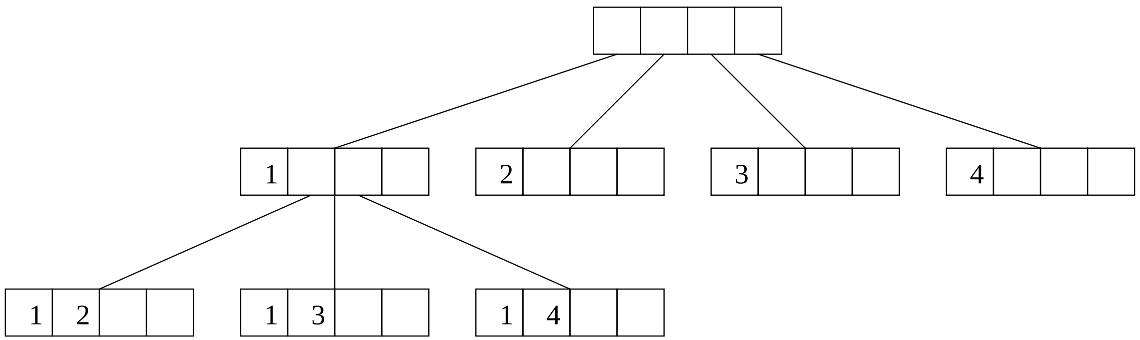
Permutaatioiden generoiminen

- Samaa ratkaisustrategiaa voimme käyttää myös seuraavaan ongelmaan: miten voidaan generoida lukujen $1, 2, \ldots, n$ kaikki permutaatiot?

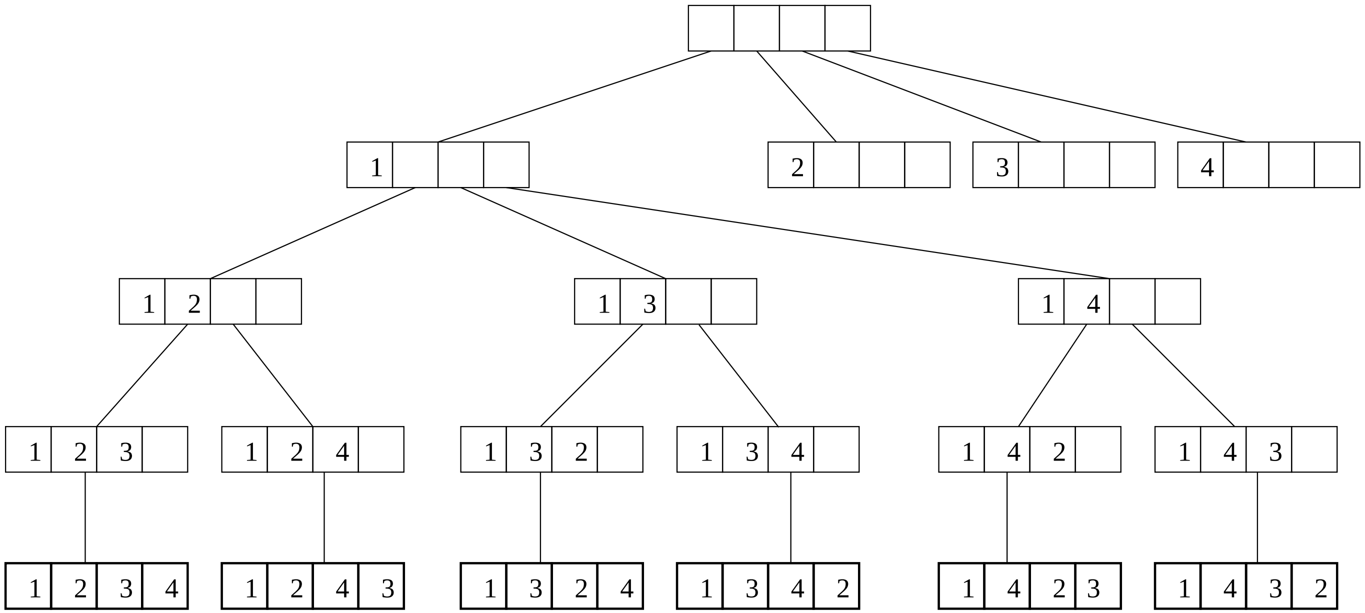
- Tarkastellaan permutaatioita luvuille $1, 2, 3, 4$
  - permutaation ensimmäinen luku voi alkaa mikä tahansa yo. luvuista:



  - vasen haara jatkuisi siten että seuraava numero voi olla joku joukosta $2, 3, 4$, luku 1 on jo käytetty sillä se aloittaa permutaation



- Seuraavassa permutaatiopuu hieman pitemmälle piirrettynä



- Valmiit permutaation löytyvät siis puun lehdistä, ja jos lehdet generoidaan esijärjestyksessä saadaan permutaatiot suuruusjärjestyksessä

- Algoritmi permutaatioiden generoimiseen
  - alustetaan n-paikkainen totuusarvoinen taulukko $used$ siten että jokaisen alkion arvo on false
  - $used$-taulukko kertoo mitkä luvuista on jo käytetty permutaatiossa
  - oletetaan että $table$ on n-paikkainen taulukko minkä alkiot ovat tyyppiä `int`
  - kutsutaan **generate**($table$,$used$,1)

```
generate(table,used,k)
if k == n+1 print(table)
else for i = 1 to n
        if used[i] == false
            used2 = luoKopio( used )
            used2[i] = true
            table[k] = i
            generate(table, used2, k+1)
```

- Algoritmin toimintaidea
  - rivillä 1 tarkistetaan onko permutaatio jo generoitu, jos on niin permutaatio tulostetaan
  - jos permutaatio ei ole vielä valmis, niin jatketaan permutaatiota kaikilla luvuilla jotka eivät vielä ole käytettyjä (rivit 2-3)
  - riveillä 4-6 käyttämätön luku lisätään permutaatioon, merkataan luku käytetyksi (taulukkoon $used2$) ja rekursiivinen kutsu (rivillä 7) jatkaa kyseistä haaraa alaspäin
- Erona kuningatar-ongelmaan siis tällä kertaa on se että puun generoimista ei lopeteta missään vaiheessa sillä haluamme tulostaa kaikki permutaatiot
- Edelleen tilavaativuus on varsin kohtuullinen, $\mathcal{O}(n^2)$, sillä yhden rekursiotason viemä tila on $\mathcal{O}(n)$ ja rekursiotasoja on n kpl
- Puun solmumäärän yläraja on $1 + n + n^2 + \ldots + n^n = \mathcal{O}(n^n)$, yhden solmun käsittely vie aikaa $\mathcal{O}(n^2)$, joten algoritmin aikavaativuus on $\mathcal{O}(n^{n+2})$

- Samoin kuin kuningatarongelmassa, ei nytkään ole välttämätöntä kuljettaa taulukkoa funktiokutsujen parametrina
- Muuttamalla taulukot *table* ja *used* globaaleiksi muuttujiksi, saadaan yhden solmun käsittelyaika lineaariseksi ja koko algoritmin aikavaativuus on $\mathcal{O}(n^{n+1})$
- Edellisellä sivulla todetaan solmujen lukumäärän ylärajan $1+n+n^2+\ldots+n^n$ olevan kertaluokkaa $\mathcal{O}(n^n)$, tämä ei ole välttämättä täysin ilmeistä joten perustellaan miksi on näin
- On selvää, että kun $n \geq 2$, niin $n^{n-1} \leq \frac{1}{2}n^n$, siispä

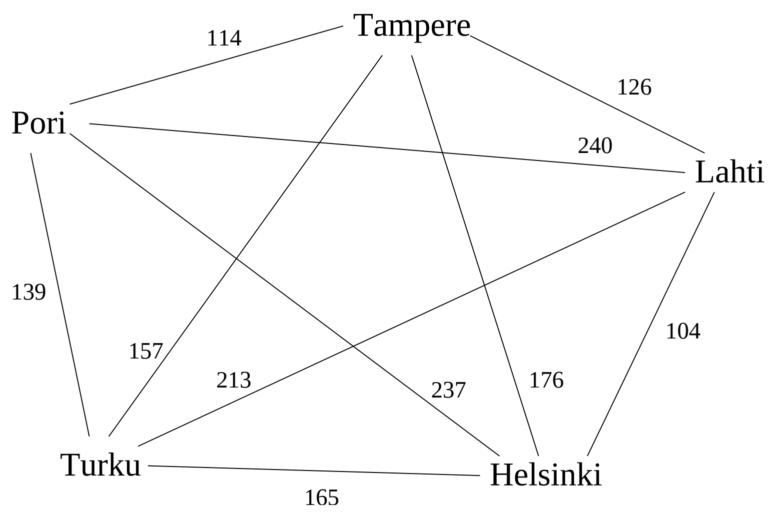
  $1+n+n^2+\ldots+n^{n-1}+n^n \leq 1+n+n^2+\ldots+n^{n-2}+\frac{1}{2}n^n+n^n$

  samoin n^{n-2} on alle puolet n^{n-1}:stä, n^{n-3} n^{n-2}:sta ... eli $n^{n-i} \leq \frac{1}{2^i}n^n$, joten

  $$\begin{aligned} 1+n+n^2+\ldots+n^n &\leq 1+\tfrac{1}{2^n}n^n+\tfrac{1}{2^{n-1}}n^n+\ldots+\tfrac{1}{2^2}n^n+\tfrac{1}{2^1}n^n+\tfrac{1}{2^0}n^n \\ &= 1+n^n(\tfrac{1}{2^n}+\tfrac{1}{2^{n-1}}+\ldots+\tfrac{1}{2^2}+\tfrac{1}{2^1}+\tfrac{1}{2^0}) \\ &= 1+n^n\textstyle\sum_{i=0}^{n}(\frac{1}{2})^i \\ &\leq 1+n^n\textstyle\sum_{i=0}^{\infty}(\frac{1}{2})^i \\ &= 1+n^n\tfrac{1}{1-\frac{1}{2}} = 1+2n^n = \mathcal{O}(n^n) \end{aligned}$$
- Edellä hyödynnettiin kaavaa $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ kun $|x| < 1$.

Kauppamatkustajan ongelma

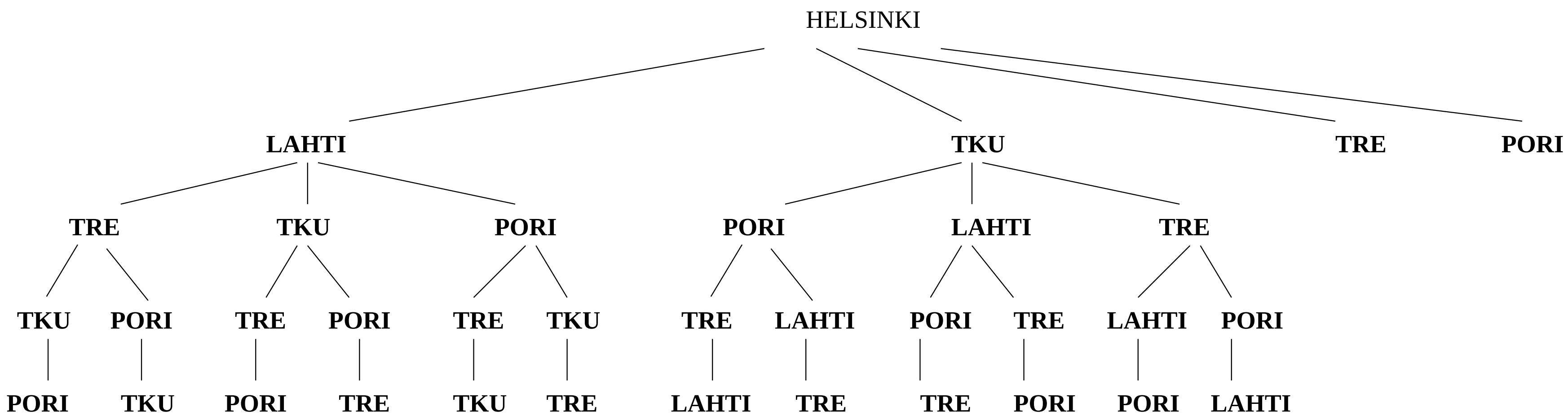
- Helsingissä asuvan kauppamatkustajan täytyy vierailla Lahdessa, Turussa, Porissa ja Tampereella



- Kulujen minimointi bisneksessä on tärkeää: mikä on lyhin reitti joka alkaa Helsingistä ja päättyy Helsinkiin ja sisältää yhden vierailun kussakin kaupungissa?

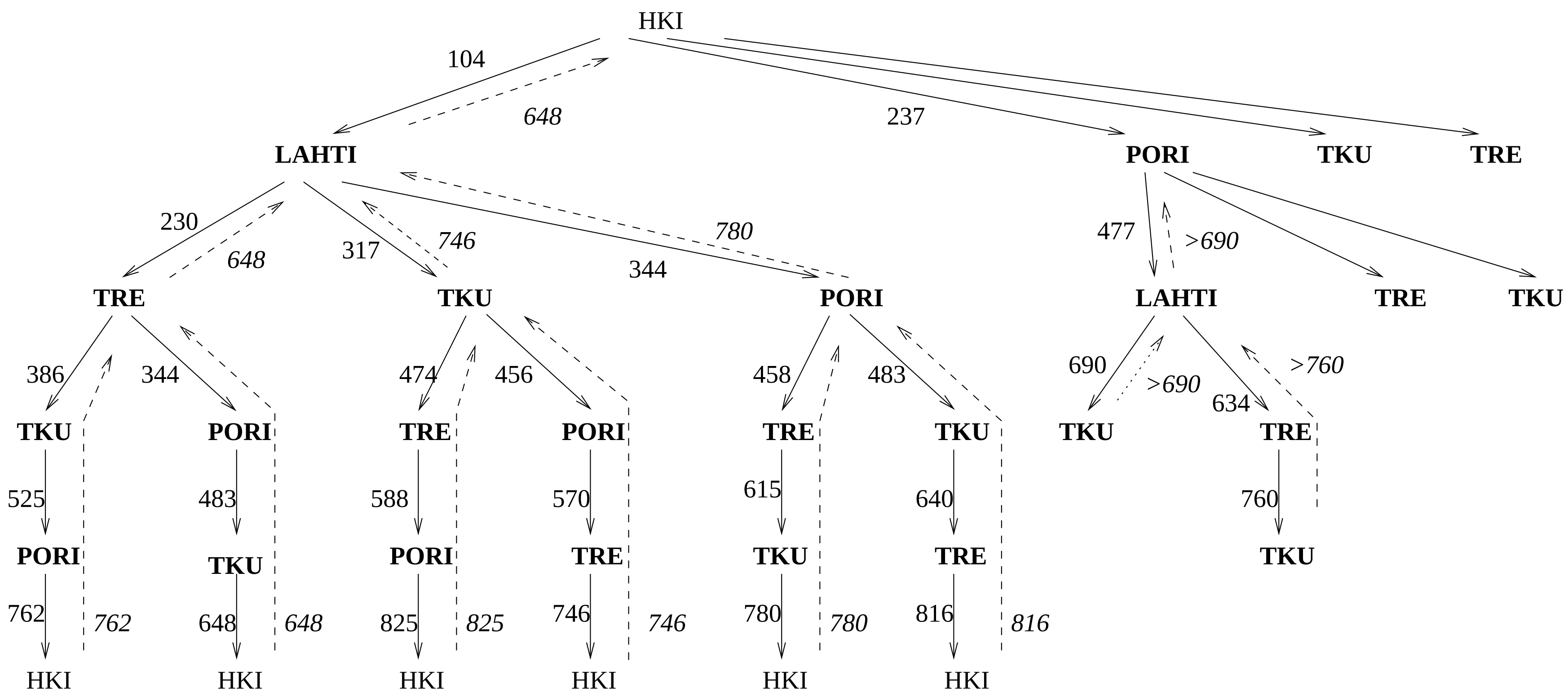
- Huomaamme että mahdolliset reitit ovat kaupunkien jonon Turku, Tampere, Pori, Lahti permutaatiot

- Voimme siis käyttää ratkaisussa samaa periaatetta kuin permutaatioiden tulostuksessa:



- Näin siis saamme systemaattisesti generoitua kaikki mahdolliset reitit

- Reitin pituus kannattaa laskea heti generoinnin yhteydessä:



- Palatessamme etsintäpuuta ylöspäin muistamme mikä oli parhaan kyseistä kautta kulkevan reitin pituus

- Uutta reittiä etsittäessä ei kannata enää jatkaa jos tiedämme että kyseinen reitti tulee joka tapauksessa olemaan pitempi kuin paras aiemmin löydetty reitti

  esim. kuvassa Helsinki $\rightarrow$ Pori $\rightarrow$ Lahti $\rightarrow$ Turku

- Koko etsintäpuun läpikäytyämme saamme tietoon lyhimmän reitin pituuden, samalla toki kannattaa merkitä muistiin minkä kaupunkien kautta reitti kulkee

- Algoritmihahmotelma
  - oletetaan että kaupunkeja on n kappaletta, Helsinki on kaupunki numero 1
  - kaksiulotteinen taulukko $dist$ kertoo kaupunkien välimatkat, esim. $dist$[1,3] sisältää Helsingin ja kaupungin numero 3 välimatkan
  - n-paikkainen totuusarvoinen taulukko $visited$ kertoo missä kaupungeissa on jo vierailtu tutkittavalla polulla
  - alustetaan $visited[i]$ = false jokaiselle i:lle
  - tulos kerätään globaaliin muuttujaan $best$ joka alustetaan arvolla ∞
  - kutsutaan **tsp**(0,$visited$,1,1)

```
tsp(length,visited,current,k)
if k == n
    if length + dist[current,1] < best
        best = length + dist[current,1]
    return
for i = 2 to n
    if visited[i] == false and length + dist[current,i] < best
        visited2 = luoKopio( visited )
        visited2[i] = true
        tsp(length + dist[current,i],visited2,i,k + 1)
return
```

- Algoritmin toimintaidea
  - parametrit:

    $best$ parhaan jo löydetyn reitin pituus

    $length$ kuinka pitkä reitin tähän asti tutkittu osa on

    $visited$ missä kaupungeissa on jo käyty

    $current$ nykyisen kaupungin numero

    k kuinka monessa kaupungissa on jo käyty
  - rivillä 1 huomataan jos koko reitti on jo generoitu ja se on lyhempi kuin paras reitti tähän mennessä: palautetaan tässä tapauksessa reitin pituus (tähän asti käydyn osan pituus + matka Helsinkiin)
  - jos reitti ei ole vielä valmis, niin jatketaan reittiä kaikilla kaupungeilla joissa ei vielä ole käyty (rivit 5-10)
  - rekursiokutsu rivillä 9 tutkii mikä on kaupungilla i jatkuvan reitin pituus
  - uutta reittiä ei tutkita jos se on jo tässä vaiheessa toivottoman pitkä
  - Huomautus: oikeastaan $visited[1]$ ei tule käyttöön ollenkaan

- Käytetty ongelmanratkaisutekniikka muistuttaa läheisesti kuningatarongelmassa käytettyä peruuttavaa etsintää jossa peruutetaan kun törmätään puun haarassa umpikujaan

- Nyt toimimme hieman kehittyneemmin, eli jos huomataan, että joku puun haara ei voi johtaa parempaan ratkaisuun kuin tunnettu paras ratkaisu, jätetään haara tutkimatta.

- Menetelmä kulkee nimellä branch-and-bound

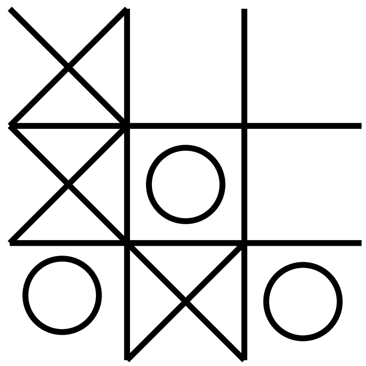
- Tilavaativuus kohtuullinen $\mathcal{O}(n^2)$ sillä yksi rekursiotaso vie tilaa $\mathcal{O}(n)$

- Aikaa algoritmi vie eksponentiaalisesti tutkittavien kaupunkien määrään nähden

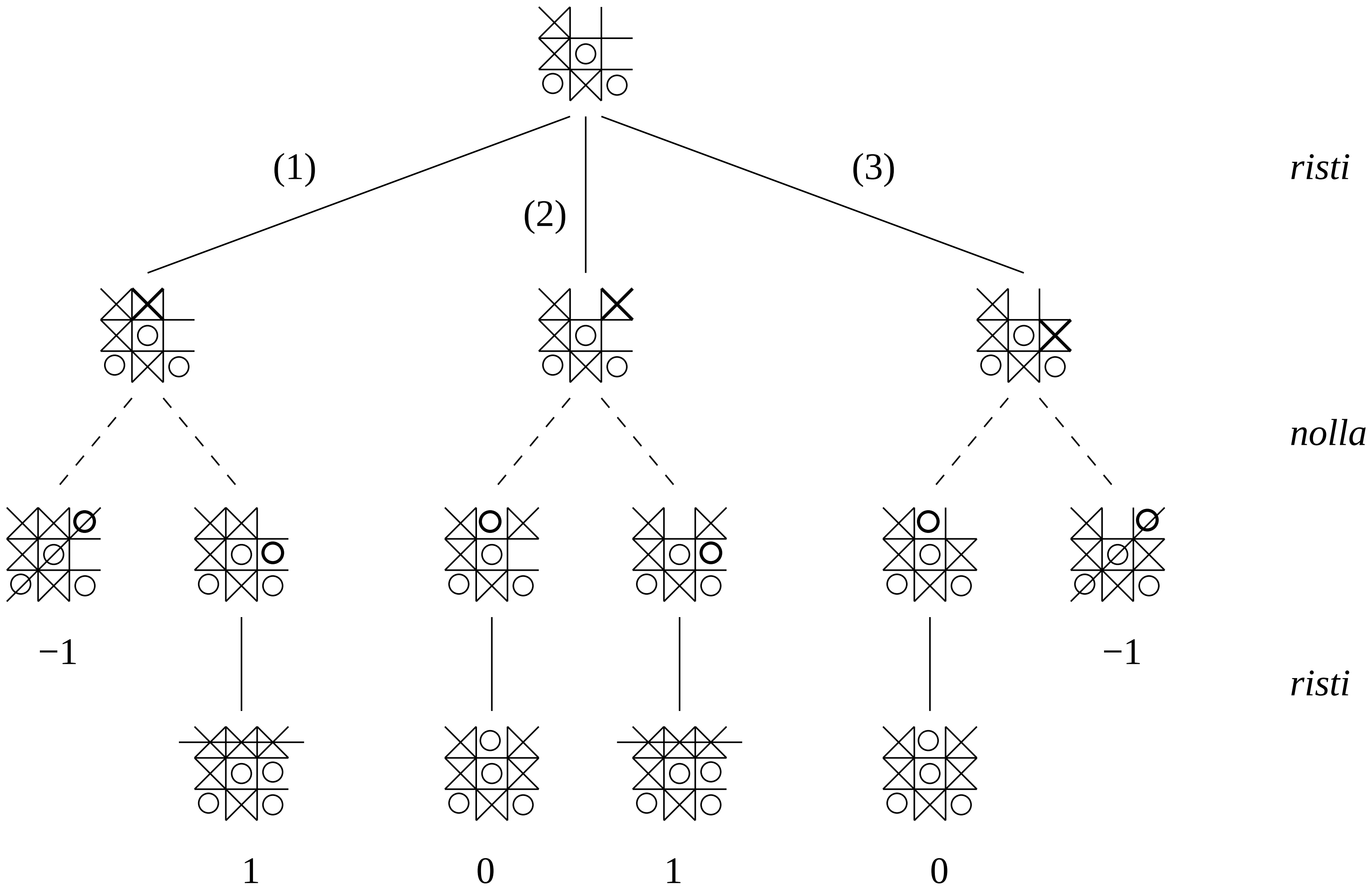
- Huom: esim. reitti Helsinki $\rightarrow$ Lahti $\rightarrow$ Tampere $\rightarrow$ Pori $\rightarrow$ Turku $\rightarrow$ Helsinki on samanpituinen myös päinvastaiseen suuntaan kuljettuna
- Sama pätee jokaiselle reitille, algoritmimme siis oikeastaan tutkii jokaisen erilaisen reitin kahteen kertaan
- Vaikka optimoisimme algoritmia siten että tämä epäkohta poistuisi, pysyy aikavaativuus silti eksponentiaalisena
- Kauppamatkustajan ongelmalle ei tiedetä parempia kuin eksponentiaalisessa ajassa toimivia ratkaisualgoritmeja
- Toisaalta ei ole pystytty todistamaan ettei nopeaa (polynomisessa ajassa toimivaa) algoritmia ole olemassa ...
- Kyseessä on ns. NP-täydellinen ongelma, aiheesta hieman enemmän kurssilla Laskennan mallit

Pelipuu

- Tietokone pelaa risti-nollaa ihmistä vastaan
- On ristin vuoro, mitä tietokoneen kannattaa tehdä seuraavassa tilanteessa?



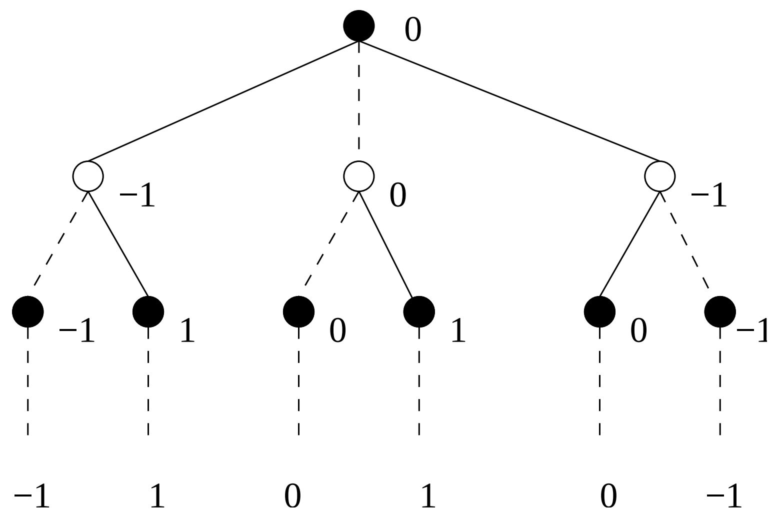
- Tietokone rakentaa päätöksensä tueksi pelipuun, ks. seuraava sivu


(1)
(2)
(3)
risti
nolla
−1
−1
risti
1
0
1
0

- On siis tehtävä valinta kolmen mahdollisen siirron välillä
- Pelipuuhun on kirjattu auki myös kaikki mahdolliset nollaa pelaavan siirrot, eli miten nolla voisi vastata kunkin ristin siirron jälkeen
- Ja edelleen, miten peli voisi jatkua kahden siirtovuoron jälkeen
- Lopputilanteita vastaaviin pelipuun lehtisolmuihin on merkattu tilanteen arvo ristin kannalta: voitto 1, tasapeli 0 ja tappio -1
- Siis minkä siirron tietokone tekee?
  - valinta (1) johtaa lopulta joko nollan tai ristin voittoon
  - valinta (2) johtaa joko tasapeliin tai ristin voittoon
  - valinta (3) johtaa joko tasapeliin tai nollan voittoon

  eli järkevintä valita siirto (2), voittomahdollisuus jää mutta on varmaa ettei ainakaan hävitä
- Strategiana on valita parhaan arvon (voitto 1, tasapeli 0, tappio -1) tuottava haara siten että oletetaan että vastustaja pelaa mahdollisimman hyvin

- Seuraavassa pelipuu piirrettynä hiukan abstraktimmin



- Ristin vuoroa vastaavat solmut ovat mustia ja nollan vuoroa vastaavat valkoisia

- Pelipuu evaluoidaan lähtien lehdistä edeten juureen
  - mustat solmut ovat max-solmuja, ne saavat arvokseen lapsen jolla suurin arvo
  - valkoiset solmut ovat min-solmuja, saaden arvokseen lapsen jolla pienin arvo

- Ristin siirtoa vastaa se lapsi minkä arvon juuren max-solmu perii

- Kuten edellisissä esimerkissämme, ei nytkään ole tarvetta luoda pelipuuta eksplisiittisesti muistiin, riittää generoida yksi polku kerrallaan

- Seuraavassa rekursiiviset operaatiot suorittavat pelipuun evaluoinnin, aluksi kutsutaan operaatiota risti parametrina meneillään olevaa pelitilannetta vastaava solmu

```
risti(v)
if v:llä ei lapsia tai peli jo ohi
    if ristillä kolmen suora return 1
    if nollalla kolmen suora return -1
    return 0
mybest = −∞
for kaikilla v:n lapsilla w eli pelitilanteilla mihin nykyisestä asetelmasta päästään
    newval = nolla(w)
    if newval > mybest mybest = newval
return mybest
```

```
nolla(v)
if v:llä ei lapsia tai peli jo ohi
    if ristillä kolmen suora return 1
    if nollalla kolmen suora return -1
    return 0
myworst = ∞
for kaikilla v:n lapsilla w eli pelitilanteilla mihin nykyisestä asetelmasta päästään
    newval = risti(w)
    if newval < myworst myworst = newval
return myworst
```

- Rivillä 1 siis huomaamme jos siirtoja ei enää ole ja palautamme pelitilannetta vastaavan arvon (rivit 2-4)

- Jos peli jatkuu vielä, käymme läpi kaikki mahdolliset siirrot (rivi 6) ja evaluoimme miten peli etenee tämän siirron seurauksena (rivi 7)

- **risti**-operaatio palauttaa parhaan lapsensa arvon ja **nolla** palauttaa huonoimman lapsensa arvon

- Esitetty pelipuun evaluointimenetelmä kulkee kirjallisuudessa nimellä min-max-algoritmi

- Risti-nollassa pelipuut ovat vielä kohtuullisen kokoisia, eli siirron valinta vie kohtuullisen ajan (huom: jotkut puun haarat ovat symmetrisiä eikä "samanlaisista" tarvitse tutkia kuin yksi vaihtoehto)

- Useimmissa peleissä, kuten esim. shakissa tilanne on aivan toinen, pelipuut ovat niin suuria, että niiden läpikäynti kokonaisuudessaan on mahdotonta
- Tällöin paras mitä voidaan tehdä, on generoida pelitilanteita tiettyyn syvyyteen asti
- Jos pelipuuta ei voida rakentaa valmiisiin tilanteisiin (voitto, häviö, tasapeli) asti, ongelmaksi nouseekin se mikä on pelipuun lehtisolmuissa olevien pelitilanteiden arvo
- Tähän on toki mahdollista kehitellä erilaisia arviointitapoja (jäljellä olevat omat/vastustajan nappulat, asetelma laudalla, y.m.)

7. Keko

- Tarkastellaan vielä yhtä tapaa toteuttaa sivulla 159 määritelty tietotyyppi joukko

- Tällä kertaa emme kuitenkaan toteuta normaalia operaatiorepertoaaria, olemme kiinnostuneita ainoastaan kolmesta operaatiosta:

  - **heap-insert**(A, k) lisää joukkoon avaimen k
  - **heap-min**(A) palauttaa joukon pienimmän avaimen arvon
  - **heap-del-min**(A) poistaa ja palauttaa joukosta pienimmän avaimen

- Nämä operaatiot tarjoavaa kekoa sanotaan minimikeoksi (engl. minimum heap)

- Toinen vaihtoehto, eli maksimikeko (engl. maximum heap) tarjoaa operaatiot:

  - **heap-insert**(A, k) lisää joukkoon avaimen k
  - **heap-max**(A) palauttaa joukon suurimman avaimen arvon
  - **heap-del-max**(A) poistaa ja palauttaa joukosta suurimman avaimen

- Näitä kolmea operaatiota sanotaan (maksimi/minimi) keko-operaatioiksi

- Keskitymme tässä luvussa ensisijaisesti maksimikekoon ja sen toteutukseen

- Huom: Keoksi kutsutaan myös muistialuetta, josta suoritettavien ohjelmien ajonaikainen muistinvaraaminen tapahtuu. Tällä tietojenkäsittelyn "toisella" keolla ei ole mitään tekemistä tietorakenteen keon kanssa

- Pystyisimme luonnollisesti toteuttamaan operaatiot käyttäen jo tuntemiamme tietorakenteita:
  - käyttämällä järjestämättömän listan **insert**, **delete** ja **max**-operaatioita **heap-insert** olisi vakioaikainen mutta **heap-del-max** veisi aikaa $\mathcal{O}(n)$
  - käyttämällä järjestetyn rengaslistan **insert**, **delete** ja **max**-operaatioita **heap-insert** veisi aikaa $\mathcal{O}(n)$ ja **heap-del-max** olisi $\mathcal{O}(1)$
  - käyttämällä AVL-puun **insert**, **delete** ja **max**-operaatioita sekä **heap-insert** että **heap-del-max** veisivät aikaa $\mathcal{O}(\log n)$

- Seuraavassa esittämämme keko-tietotyypin toteutuksessa sekä **heap-del-max** (tai minimikeossa **heap-del-min**) että **heap-insert** toimivat ajassa $\mathcal{O}(\log n)$ ja **heap-max** (tai minimikeossa **heap-min**) toimii vakioajassa $\mathcal{O}(1)$

- Herää kysymys, mihin tarvitsemme näin spesialisoitunutta tietorakennetta?
- Eikö riitä, että käyttäisimme muokattua tasapainoitettua hakupuuta, sillä näin saavutettu keko-operaatioiden vaativuus olisi $\mathcal{O}$-analyysin mielessä sama kuin kohta esitettävällä varsinaisella kekototeutuksella?
- Tulemme kurssin aikana näkemään algoritmeja, jotka käyttävät aputietorakenteenaan kekoa ja näiden algoritmien tehokkaan toteutuksen kannalta keko-operaatioiden tehokkuus on oleellinen
- Vaikka keko ei tuokaan kertaluokkaparannusta operaatioihin, on se toteutukseltaan hyvin kevyt, ja käytännössä esim. AVL-puuhun perustuvaa toteutusta huomattavasti nopeampi
- Keko on ohjelmoijalle kiitollinen tietorakenne siinä mielessä, että toteutus on hyvin yksinkertainen toisin kuin esim. tasapainoisten hakupuiden toteutukset
- Keosta on erilaisia versioita, kuten binomikeko ja Fibonacci-keko (joita emme käsittele täällä)

- Minimikeon avulla saamme toteutettua tehokkaasti prioriteettijonon:
  - **heap-insert** vie jonottajan jonoon, avain vastaa jonottajan prioriteettia (mitä pienempi numeroarvo sitä korkeampi prioriteetti)
  - **heap-del-min** ottaa jonosta seuraavaksi palveltavaksi korkeimman prioriteetin omaavan jonottajan
- Keon avulla saamme myös toteutettua tehokkaan kekojärjestämisalgoritmin
- Myös verkkoalgoritmien yhteydessä (luvussa 8) löydämme käyttöä keolle

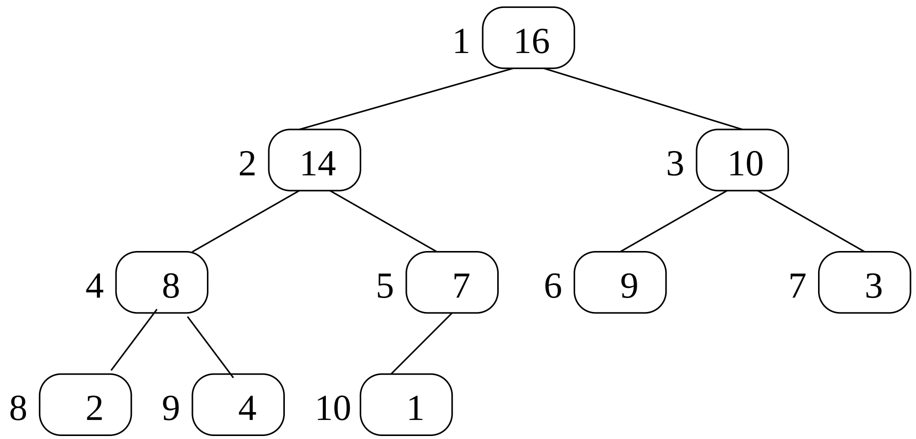
Maksimikeon toteuttaminen

- Keko kannattaa ajatella binääripuuna, joka on talletettu muistiin taulukkona
- Binääripuu on maksimikeko jos

  (K1) kaikki lehdet ovat kahdella vierekkäisellä tasolla k ja $k+1$ siten että tason $k+1$ lehdet ovat niin vasemmalla kuin mahdollista ja kaikilla k:ta ylempien tasojen solmuilla on kaksi lasta

  (K2) jokaiseen solmuun talletettu arvo on suurempi tai yhtäsuuri kuin solmun lapsiin talletetut arvot

- Seuraava binääripuu on maksimikeko



- Keko-ominaisuuden (k1) ansiosta puu voidaan esittää taulukkona, missä solmut on lueteltu tasoittain vasemmalta oikealle

- Yllä oleva puu taulukkoesityksenä:

1	2	3	4	5	6	7	8	9	10	11	12
16	14	10	8	7	9	3	2	4	1		
juuri	*taso 1*		*taso 2*				*taso 3*				

- Käytännössä keko kannattaa aina tallentaa taulukkoa käyttäen

- Kekotaulukkoon A liittyy kaksi attribuuttia
  - $A.length$ kertoo taulukon koon
  - $A.heap\text{-}size$ kertoo montako taulukon paikkaa (alusta alkaen) kuuluu kekoon
  - Huom: Taulukossa A voi siis kohdan $heap\text{-}size$ jälkeen olla "kekoon kuulumattomia" lukuja

- Keon juurialkio on talletettu taulukon ensimmäiseen paikkaan $A[1]$

- Taulukon kohtaan i talletetun solmun vanhemman sekä lapset tallettavat taulukon indeksit saadaan selville seuraavilla apuoperaatioilla:

  parent(i)
      return $\lfloor i/2 \rfloor$

  left(i)
      return 2i

  right(i)
      return 2i+1

- Vaikka varsinaisia viitteitä ei ole, on keossa liikkuminen todella helppoa,
  - tarkastellaan edellisen sivun esimerkkitapausta
  - juuren $A[1]$ vasen lapsi on paikassa $A[2 * 1] = A[2]$ ja oikea lapsi paikassa $A[2 * 1 + 1] = A[3]$
  - solmun $A[5]$ vanhempi on $A[\lfloor 5/2 \rfloor] = 2$, vasen lapsi $A[10]$ mutta koska $\mathit{heap\text{-}size}[A] = 10$, niin oikeaa lasta ei ole

- Voimme lausua kekoehdon (K2) nyt seuraavasti:

  kaikille $1 < i \leq \mathit{heap\text{-}size}$ pätee $A[\mathit{parent}(i)] \geq A[i]$

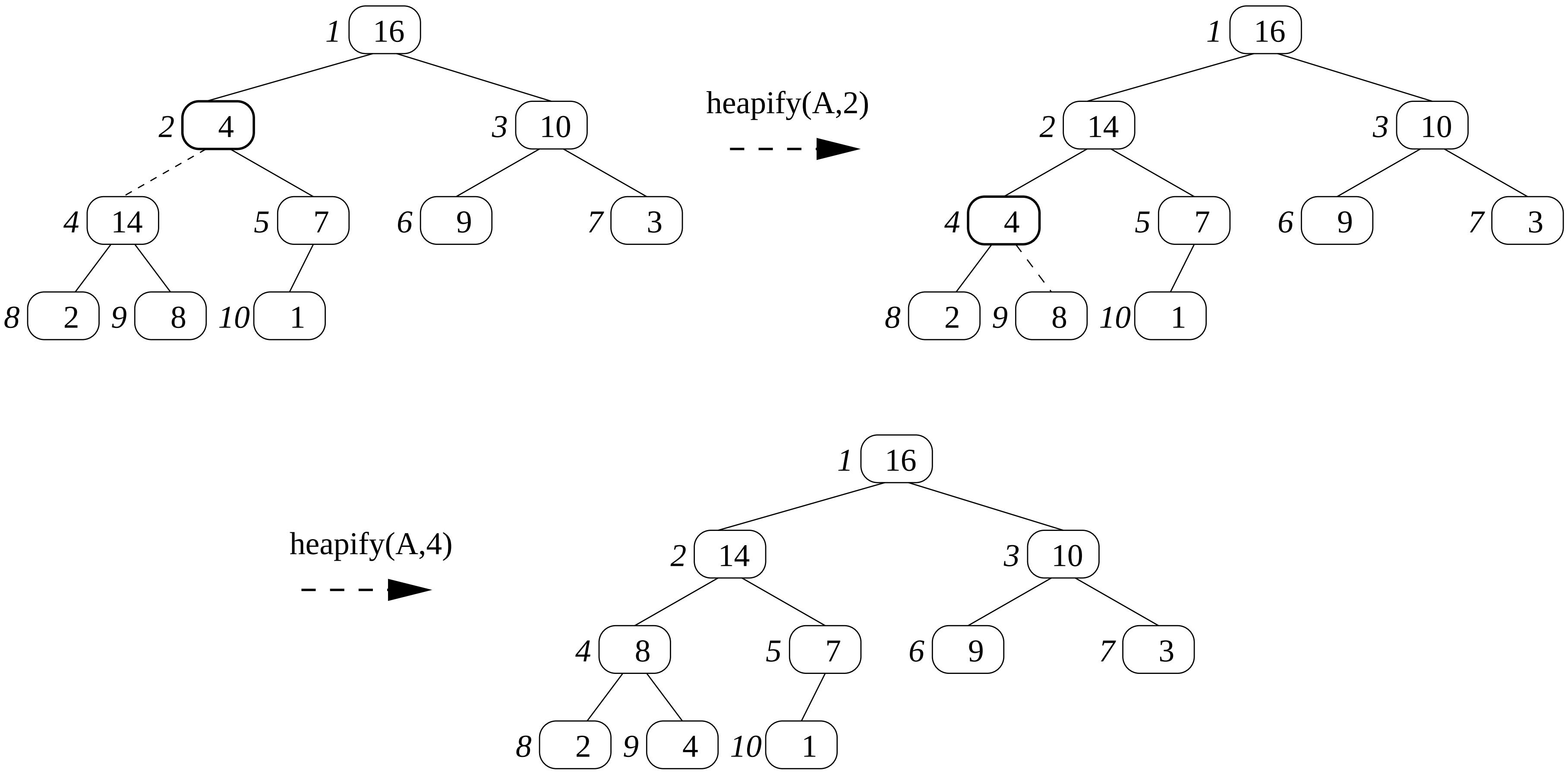
- Ennen varsinaisia keko-operaatioita, toteutetaan tärkeä apuoperaatio **heapify**
- Operaatio korjaa kekoehdon, jos se on rikki solmun i kohdalla
- Oletus on, että solmun i vasen ja oikea alipuu toteuttavat kekoehdon
- Kutsun **heapify**(A, i) jälkeen koko solmusta i lähtevä alipuu toteuttaa kekoehdon
- Toimintaperiaate on seuraava:
  - Jos $A[i]$ on pienempi kuin toinen lapsistaan, vaihdetaan se suuremman lapsen kanssa
  - Jatketaan rekursiivisesti alas muuttuneesta lapsesta

- **heapify**:n parametreina ovat taulukko A ja indeksi i
  - oletuksena siis on että $\mathit{left}(i)$ ja $\mathit{right}(i)$ viittaavat jo kekoja olevien alipuiden $A[\mathit{left}(i)]$ ja $A[\mathit{right}(i)]$ juuriin
  - operaatio kuljettaa alkiota $A[i]$ alaspäin, kunnes alipuusta jonka juurena $A[i]$ on tulee keko

```
heapify(A,i)
1   l = left(i)
2   r = right(i)
3   if r ≤ A.heap-size
4       if A[l] > A[r] largest = l
5       else largest = r
6       if A[i] < A[largest]
7           vaihda A[i] ja A[largest]
8           heapify(A,largest)
9   elsif l == A.heap-size and A[i]<A[l]
10          vaihda A[i] ja A[l]
```

- Jos molemmat lapset ovat olemassa (rivi 3), vaihdetaan tarvittaessa $A[i]$:n arvo lapsista suuremman arvoon ja kutsutaan lapselle **heapify**-operaatiota
- Jos vain vasen lapsi on olemassa ja tämän arvo suurempi kuin $A[i]$:n, vaihdetaan arvot keskenään (rivit 9 ja 10)

- Seuraava kuvasarja valottaa operaation toimintaa



- **heapify**-operaation suoritusaika riippuu ainoastaan puun korkeudesta, rekursiivisia kutsuja tehdään pahimmassa tapauksessa puun korkeuden verran

- n-alkioisen keon korkeus selvästi $\mathcal{O}(\log n)$ sillä keko on lähes täydellinen binääripuu

- n alkiota sisältävälle keolle tehdyn **heapify**-operaation pahimman tapauksen aikavaativuus on siis $\mathcal{O}(\log n)$

- Rekursion takia operaation tilavaativuus on $\mathcal{O}(\log n)$

- Operaatio on helppo kirjoittaa myös ilman rekursiota, jolloin se toimii vakiotilassa

- Kekoehdosta (K2) seuraa suoraan että keon maksimialkio on talletettu paikkaan $A[1]$

- Operaatio **heap-max** siis on triviaali ja vie vakioajan

  heap-max(A)
      return A[1]

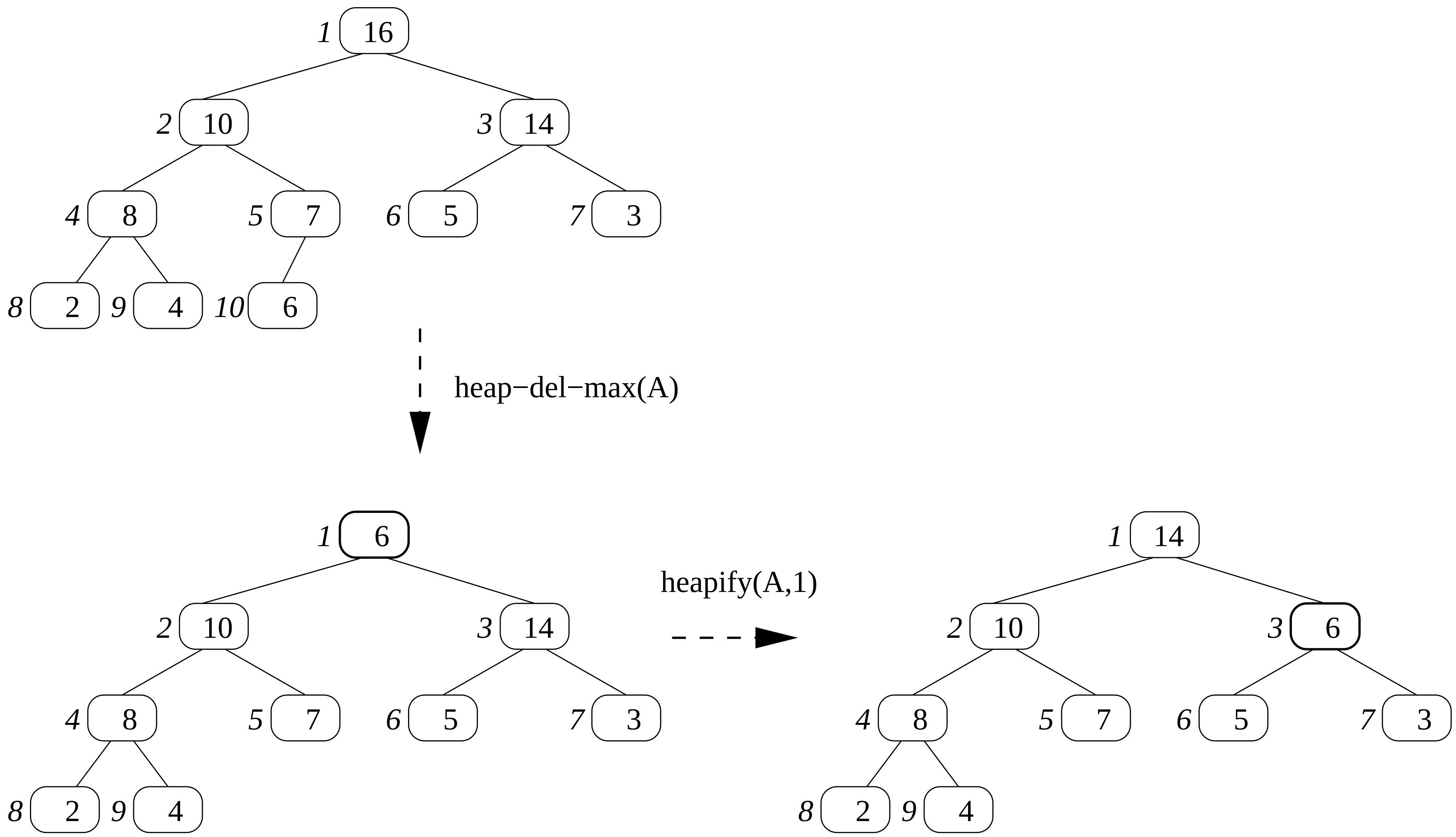
- Alkion poistaminen keosta

```
heap-del-max(A)
1   max = A[1]
2   A[1] = A[A.heap-size]
3   A.heap-size = A.heap-size -1
4   heapify(A,1)
5   return max
```

- Toimintaidea
  - operaatio palauttaa kohdassa $A[1]$ olleen avaimen
  - keon viimeisessä paikassa oleva alkio $A[A.\mathit{heap\text{-}size}]$ viedään poistetun alkion tilalle ja keon kokoa pienennetään yhdellä (rivit 2 ja 3)
  - keko on muuten kunnossa mutta kohtaan $A[1]$ siirretty avain saattaa rikkoa keko-ominaisuuden, kutsutaan **heapify** operaatiota korjaamaan tilanne

- Operaation aikavaativuus sama kuin **heapify**:llä, eli $\mathcal{O}(\log n)$

- Esimerkki **heap-del-max**-operaation toiminnasta

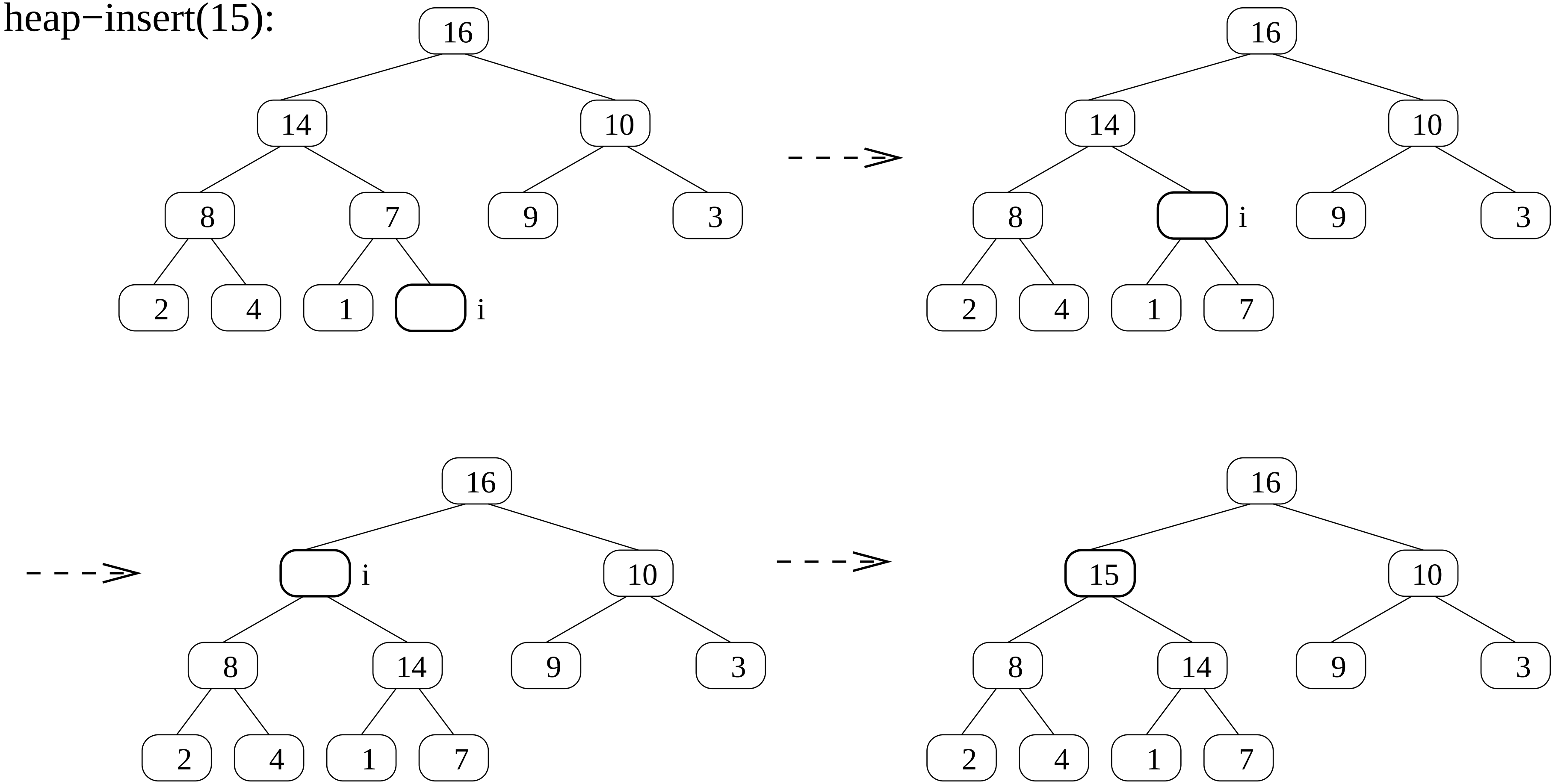


- Alkion lisääminen kekoon

```
heap-insert(A,k)
A.heap-size = A.heap-size+1
i = A.heap-size
while i>1 and A[parent(i)] < k
    A[i] = A[parent(i)]
    i = parent(i)
A[i] = k
```

- Toimintaidea
  - kasvatetaan keon kokoa yhdellä solmulla eli tehdään paikka uudelle avaimelle
  - kuljetaan nyt keon uudesta solmusta ylöspäin ja siirretään arvoja samaan aikaan yhtä alemmas niin kauan kunnes uudelle alkiolle löydetään paikka joka ei riko keko-ominaisuutta (K2)
- Pahimmassa tapauksessa lisättävä avain viedään puun juureen ja näin käydessä puun korkeudellisen verran avaimia on valutettu alaspäin
- Operaation aikavaativuus siis on $\mathcal{O}(\log n)$
- Seuraavalla sivulla esimerkki operaation toiminnasta

heap−insert(15):

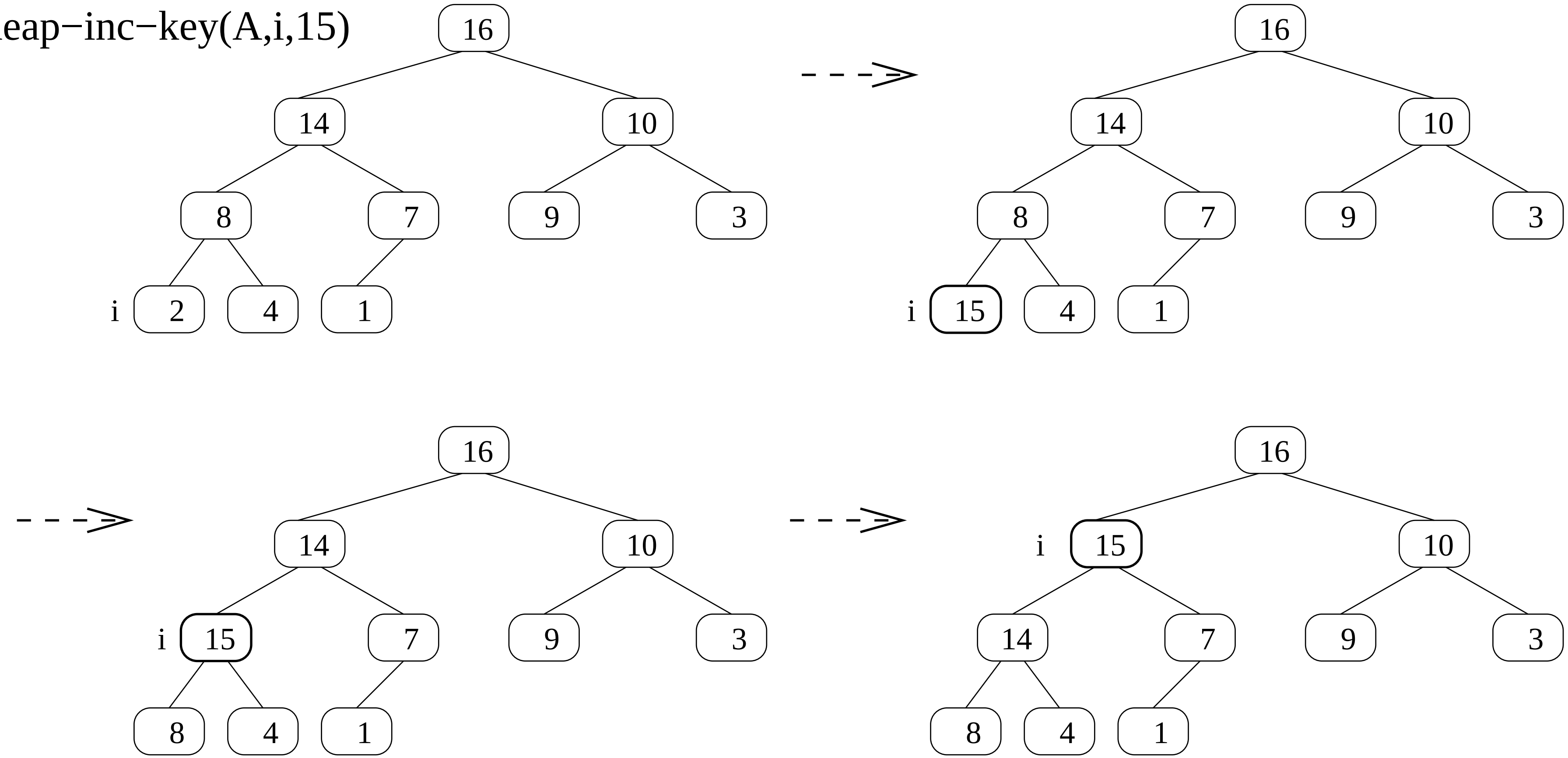


- Jotkut sovellukset tarvitsevat keko-operaatiota, joka kasvattaa annetussa indeksissä olevan avaimen arvoa

```
heap-inc-key(A,i,newk)
1  if newk > A[i]
2      A[i] = newk
3      while i>1 and A[parent(i)] < A[i]
4          vaihda A[i] ja A[parent[i]]
5          i = parent(i)
```

- Toimintaidea
  - jos yritetään pienentää avaimen arvoa, operaatio ei tee mitään
  - kopioidaan keon kohtaan i uusi avaimen arvo (rivi 2)
  - jos kasvatettu avain rikkoo keko-ominaisuuden (K2), vaihdetaan sen arvo vanhemman kanssa niin monta kertaa kunnes oikea paikka löytyy (rivit 3-5)
- Pahimmassa tapauksessa lehdessä olevaa avainta muutetaan ja muutettu avain joudutaan kuljettamaan aina puun juureen saakka
- Operaation aikavaatività on $\mathcal{O}(\log n)$
- Seuraavalla sivulla esimerkki operaation toiminnasta

heap−inc−key(A,i,15)



- Vastaavasti voidaan pienentää annetussa indeksissä olevan avaimen arvoa

  heap-dec-key(A,i,newk)

  ```
  if newk < A[i]
      A[i] = newk
      heapify(A,i)
  ```

- Nyt voidaan siis vaihtaa avainta indeksissä i joko **heap-inc-key**$(A, i, newk)$:lla tai **heap-dec-key**$(A, i, newk)$, sen mukaan onko $newk$ pienempi tai suurempi kuin $A[i]$

Kekojärjestäminen

- Oletetaan että A on n-paikkainen kokonaislukutaulukko
- Seuraava algoritmi järjestää A:n alkiot suuruusjärjestykseen käyttäen kekoa H aputietorakenteena

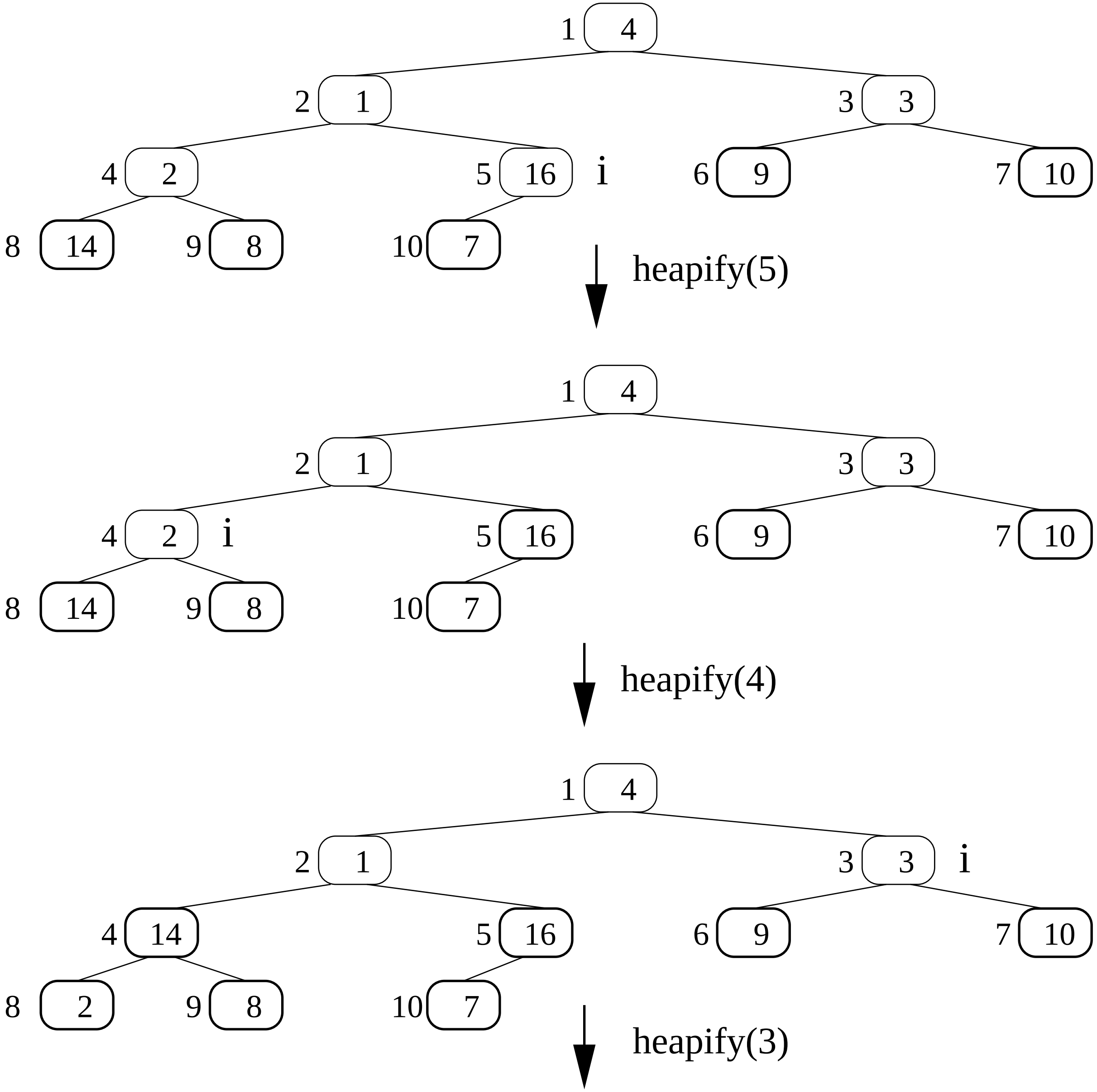
```
sort-with-heap(A,n)
for i = 1 to n
    heap-insert(H,A[i])
for i = n downto 1
    A[i] = heap-del-max(H)
```

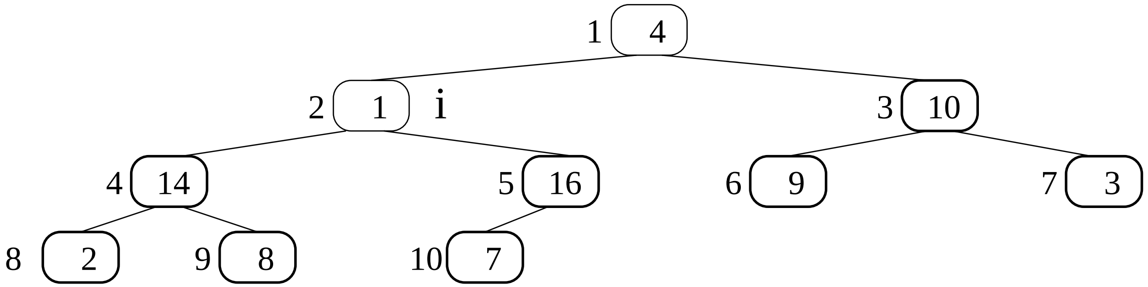
- Algoritmin aikavaativuus on $\mathcal{O}(n \log n)$, sillä **heap-insert** ja **heap-del-max** vievät $\mathcal{O}(\log n)$ ja molempia kutsutaan n kertaa

- Mutta voimme toimia fiksummin: operaation **heapify** avulla on helppo rakentaa mistä tahansa taulukosta A keko:
  - lehdet, eli paikossa $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ olevat yhden alkion alipuut ovat jo kekoja
  - kutsutaan **heapify** lapselliselle kekosolmulle alkaen $A[\lfloor n/2 \rfloor]$:sta aina juureen $A[1]$ asti
  - näin taulukko A muuttuu keoksi
  - for-silmukan invariantti on siis: kaikilla $i < j \leq A.\mathit{length}$ solmusta j lähtevä alipuu toteuttaa kekoehdon

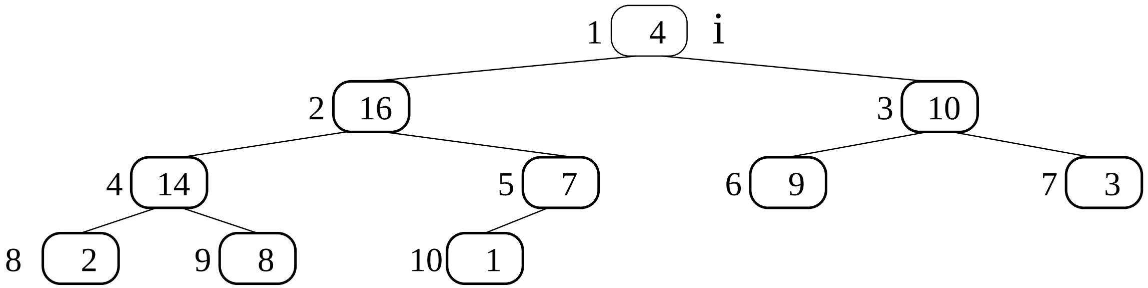
```
build-heap(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    heapify(A,i)
```

	1	2	3	4	5	6	7	8	9	10
alussa	4	1	3	2	16	9	10	14	8	7

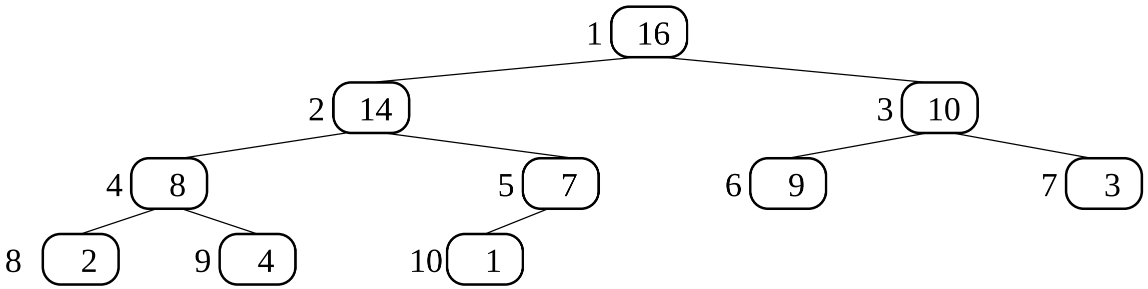




heapify(2); heapify(5)



heapify(1); heapify(2); heapify(4)



tuloksena

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

- **Kekojärjestäminen** tapahtuu seuraavasti

```
heap-sort(A)
    build-heap(A)
    for i = A.length downto 2
        vaihda A[1] ja A[i]
        A.heap-size = A.heap-size −1
        heapify(A,1)
```

- Toimintaidea
  - aineistosta tehdään ensin maksimikeko, näin suurin alkio on kohdassa $A[1]$
  - vaihdetaan keskenään keon ensimmäinen ja viimeinen alkio
  - näin saamme yhden alkion vietyä taulukon loppuun "oikealle" paikalleen
  - pienentämällä keon kokoa yhdellä huolehditaan vielä että viimeinen alkio ei enää kuulu kekoon
  - paikkaan $A[1]$ viety alkio saattaa rikkoa keko-ominaisuuden
  - huolehditaan vielä että keko-ominaisuus säilyy kutsumalla **heapify**$(A, 1)$
  - toistetaan samaa niin kauan kun keossa on alkiota

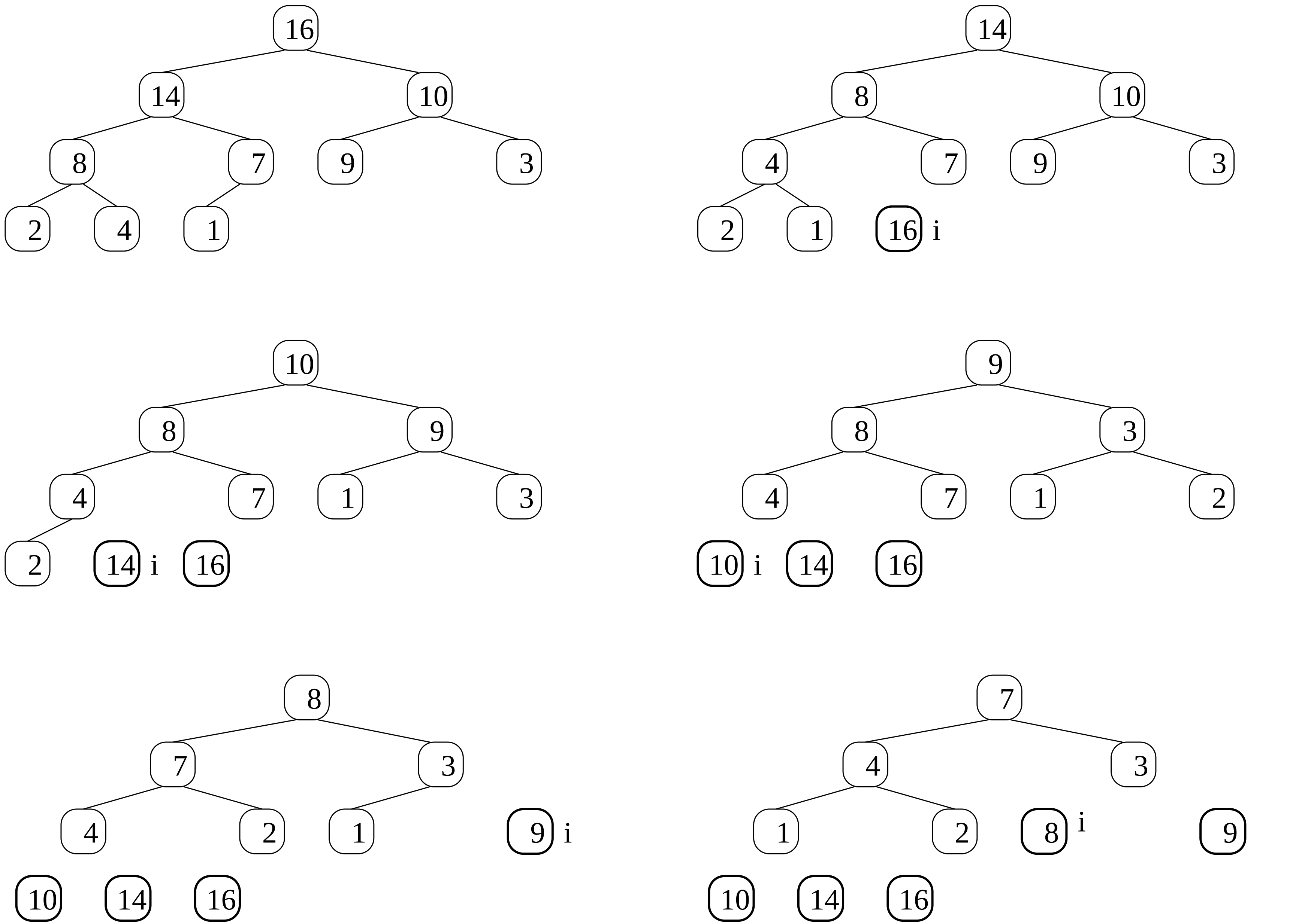
- **Kekojärjestämisen** aikavaativuus
  - **heapify**:n aikavaatimus on $\mathcal{O}(\log n)$ keolle jossa n alkiota
  - **build-heap**-operaatiossa kutsutaan $n/2$ kertaa **heapify** keolle, jossa on korkeintaan n alkiota, siis **build-heap** käyttää aikaa korkeintaan $\mathcal{O}(n \log n)$
  - **heap-sortissa** kutsutaan vielä $n - 1$ kertaa **heapify**-operaatiota
  - kokonaisuudessaan **kekojärjestämisen** aikavaativuus on siis $\mathcal{O}(n \log n)$
- Tilavaativuus
  - **heapify** kutsuu rekursiivisesti itseään pahimmillaan keon korkeudellisen verran, operaatio on kuitenkin helppo toteuttaa myös ilman rekursiota jolloin sen tilantarve on vakio
  - muutkaan **kekojärjestämisen** toimet eivät aputilaa tarvitse, siis tilavaativuus $\mathcal{O}(1)$

- Tarkempi analyysi paljastaa että operaation **build-heap** aikavaativuus onkin oikeastaan vain $\mathcal{O}(n)$
  - Operaation **heapify**(A, i) aikavaativuus on $\mathcal{O}(h(i))$, missä $h(i)$ on solmun i korkeus puussa
  - Keon korkeus on selvästi $k = \lfloor \log_2 n \rfloor$
  - Keon tasolla j on korkeintaan 2^j solmua ja näiden korkeus on $k - j$ tai $k - j - 1$, eli korkeintaan $k - j$
  - Operaation **build-heap** aikavaativuus on siis $\mathcal{O}(\sum_i h(i))$, mutta edellisen perusteella

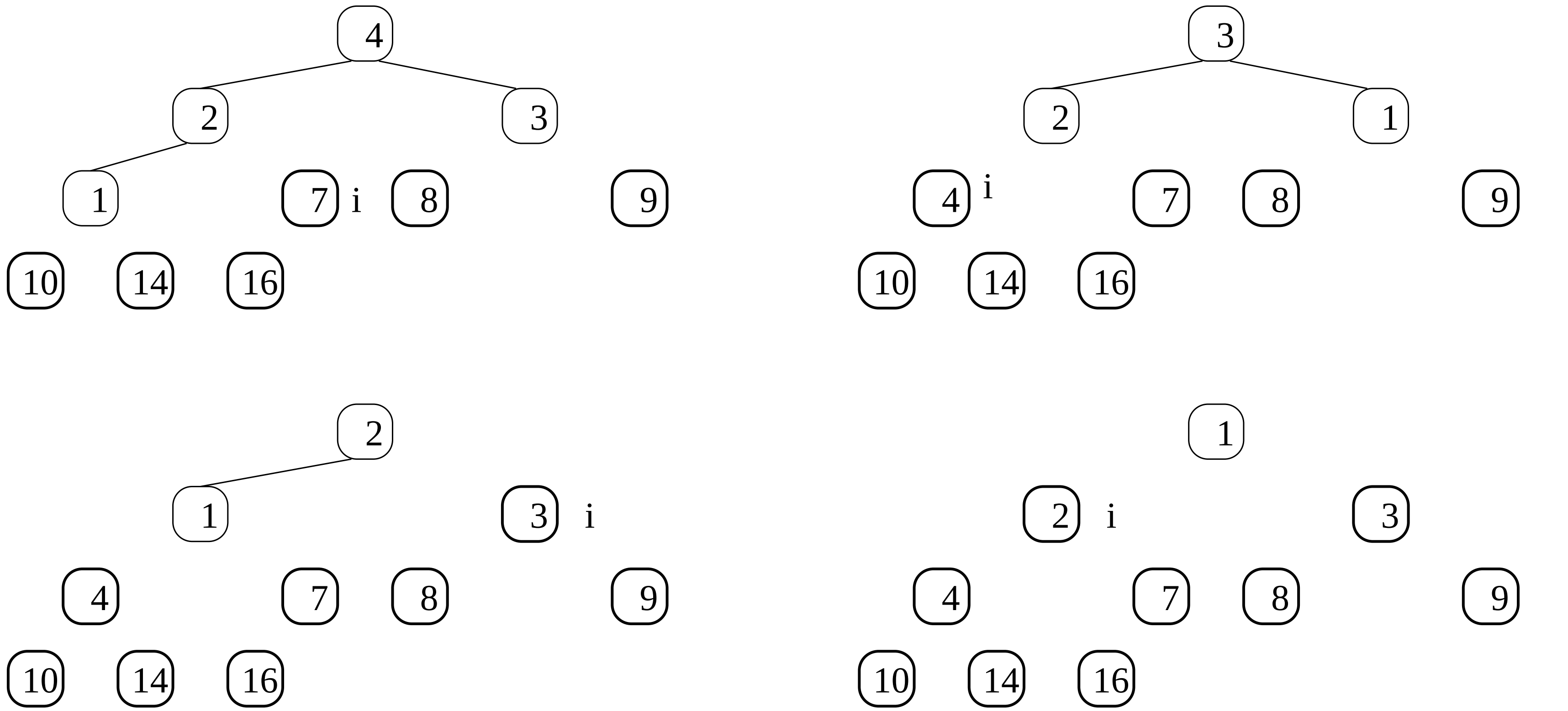
$$\sum_i h(i) \leq \sum_{j=0}^{k} 2^j (k - j) = \sum_{j=0}^{k} 2^{k-j} \cdot j = 2^k \sum_{j=0}^{k} j(\frac{1}{2})^j < 2^k \sum_{j=0}^{\infty} j(\frac{1}{2})^j \leq 2n,$$

    koska $2^k \leq n$ ja kaavakirjasta näemme, että $\sum_{j=0}^{\infty} j(\frac{1}{2})^j = 2$

- Seuraavilla sivuilla on esimerkki **kekojärjestämisestä**



16
14
10
8
7
9
3
2
4
1
14
8
10
4
7
9
3
2
1
16
i
10
8
9
4
7
1
3
2
14
i
16
9
8
3
4
7
1
2
10
i
14
16
8
7
3
4
2
1
9
i
10
14
16
7
4
3
1
2
8
i
9
10
14
16



tuloksena	1	2	3	4	7	8	9	10	14	16

- Mistä johtuu, että **pikajärjestäminen** toimii käytännössä useimmiten nopeammin kuin **kekojärjestäminen**?

- **Kekojärjestäminen** vaihtaa hyvin usein täysin eri puolilla järjestettävää taulukkoa olevien alkioiden arvoja, **pikajärjestäminen** taas pysyttelee useimmiten pitemmän aikaa pienemmässä osassa taulukkoa

- Tällä on suuri merkitys käytännössä, sillä jos muistiviittaukset keskittyvät tietyllä ajanjaksolla pieneen osaan taulukkoa, on todennäköisempää että taulukon tarvittava osa löytyy välimuistista

- Välimuistiin tehtävien muistihakujen viemä aika on merkittävästi pienempi verrattuna siihen, että tieto jouduttaisiin hakemaan keskusmuistista

- Asian merkitys korostuu vielä enemmän, jos koko järjestettävä taulukko ei mahdu kerralla keskusmuistiin, vaan sijaitsee osittain kiintolevyn swap-osiossa

- Käytännössä on myös osoittautunut, että **kekojärjestäminen** suorittaa keskimäärin monta kertaa enemmän vertailu- ja sijoitusoperaatioita kuin **pikajärjestäminen**

- Ei ole mitenkään ilmeistä mistä tämä seikka johtuu.

Keko käytännössä

- Monissa käytännön sovelluksissa, esim. käytettäessä kekoa prioriteettijonona, keottavat alkiot sisältävät muitakin attribuutteja kuin pelkän avaimen

- Tällöin itse kekoon ei välttämättä kannata tallettaa muuta kuin avaimet sekä viitteet avaimeen liittyvään muun datan tallettavaan olioon

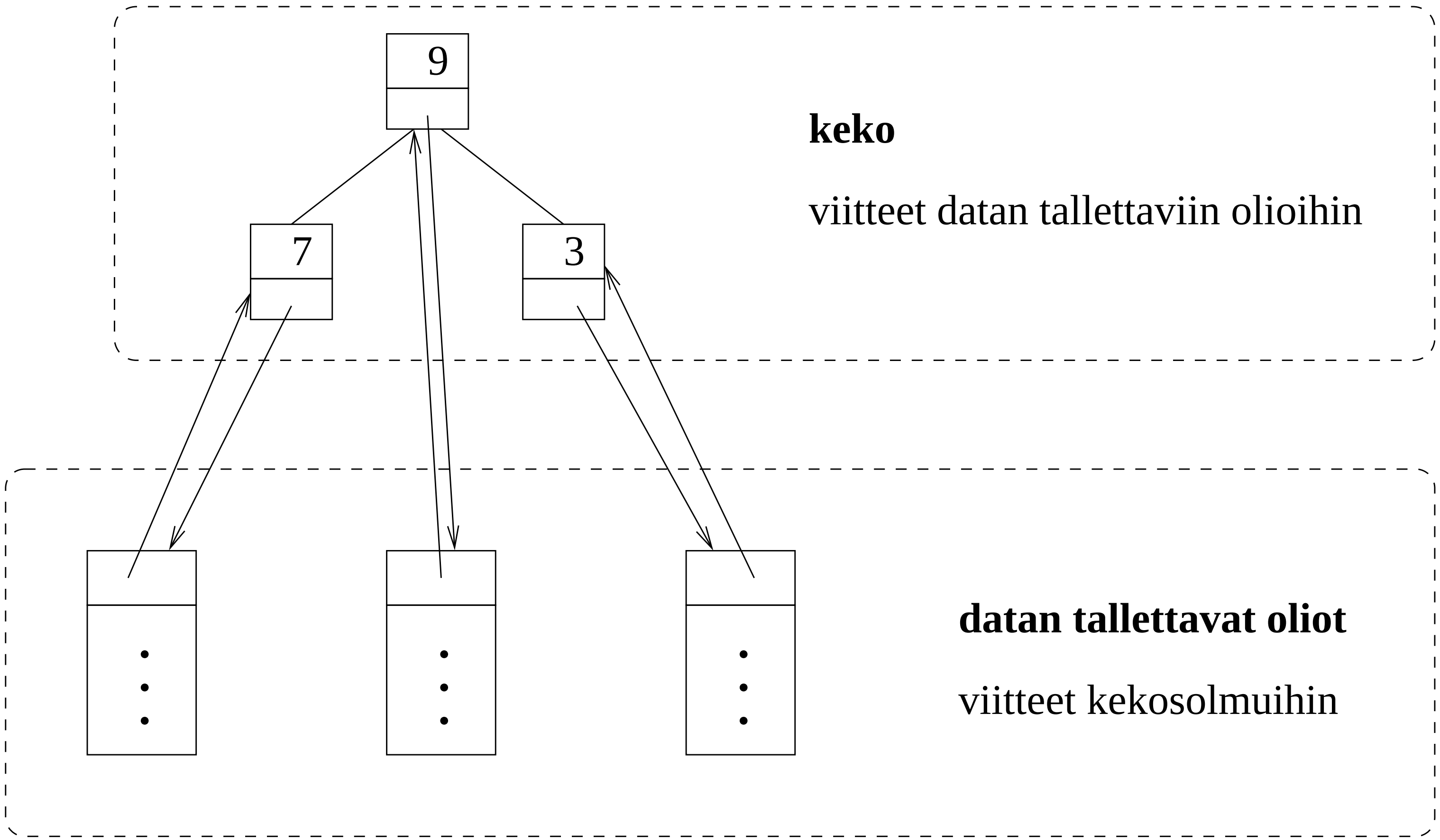
- Tällaisessa käytännön tilanteessa keko-operaatioiden parametrit kannattanee valita seuraavasti

- **heap-insert**(A, x, k)	lisää kekoon olion x, jolla avain k
- **heap-max**(A)	palauttaa viitteen olioon jolla on avaimena keon maksimiarvo
- **heap-del-max**(A)	palauttaa viitteen olioon jolla on avaimena keon maksimiarvo ja poistaa olion liittyvän avaimen keosta
- **heap-inc-key**$(x, newk)$	kasvattaa olion x avainta antaen sille uuden arvon $newk$
- **heap-dec-key**$(x, newk)$	pienentää olion x avainta antaen sille uuden arvon $newk$

- Jotta operaatio **heap-inc-key** saadaan toteutettua tehokkaasti datan talletettavissa olioissa on myös oltava viite vastaavaan kekoalkioon

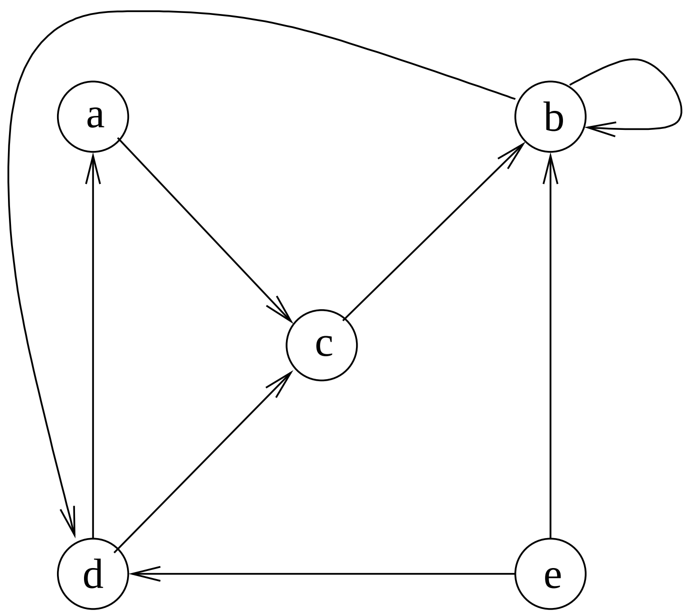
- Käytännössä viitteet siis ovat kekotaulukon indeksejä eli kokonaislukuja

- Muistin organisointi näyttää esim. seuraavalta:

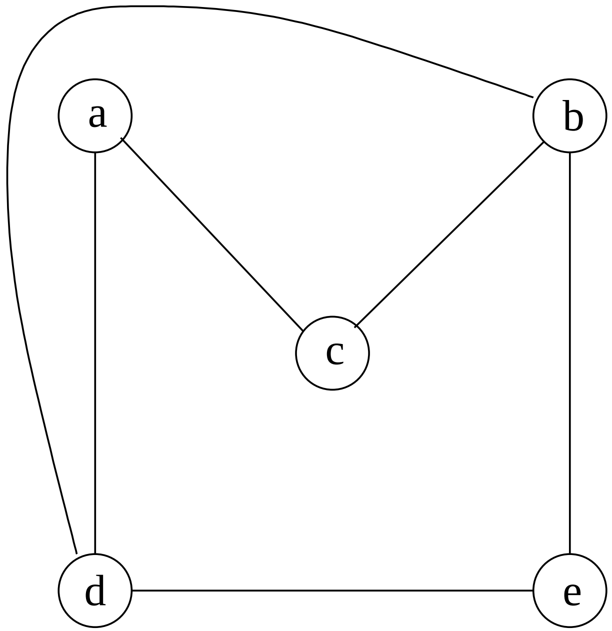


8. Verkot

- Verkko (engl. graph) koostuu solmuista (engl. vertex, node) ja niitä yhdistävistä kaarista (engl. edge)
- Verkkoja on kahta päätyyppiä
- Suunnatuissa verkoissa (engl. directed graph) kaarilla (engl. edge, arc) on suunta
- Esim:



- Suuntaamattomien verkkojen (engl. undirected graph) kaarilla ei ole suuntaa
- Esim:



- Verkoilla on paljon sovelluksia tietojenkäsittelyssä (ja muualla)
- Tutustutaan ensin verkon käsitteistöön, sen jälkeen katsotaan muutamia verkkojen sovelluksia ja tutustutaan tyypillisimpiin verkkoalgoritmeihin

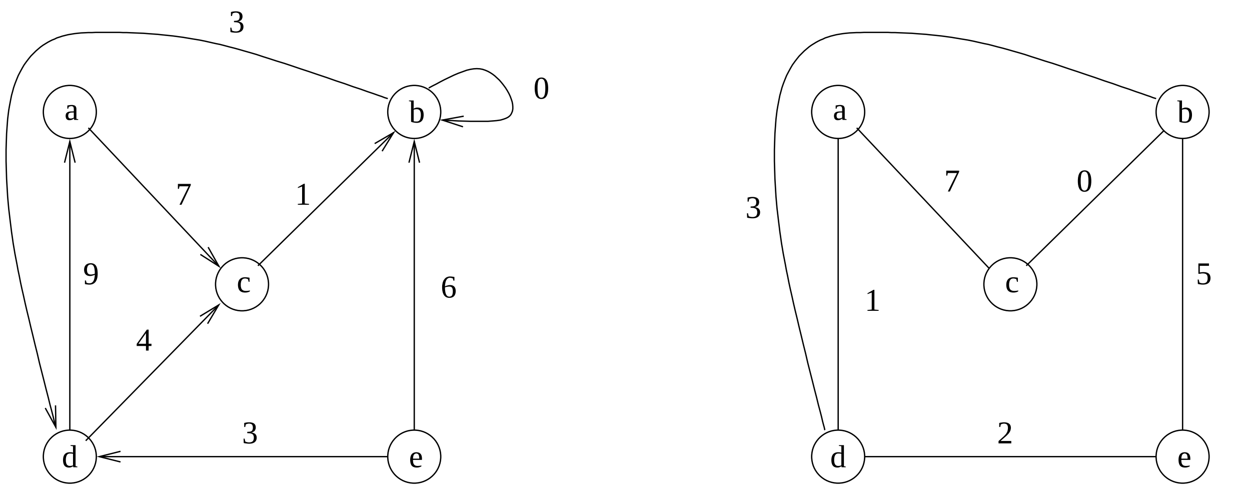
Käsitteistö

- Formaalisti verkko G esitetään parina (V, E), missä
  - V on solmujen joukko
  - E on kaarien joukko, $E \subseteq V \times V$
- Merkinnät: G niin kuin graph, V vertex ja E edge
- Kaaret ovat siis pareja (u, v) missä u ja v ovat solmuja
- Suunnatussa verkossa $(u, v) \in E$ jos solmusta u on kaari solmuun v
  - tällöin u on kaaren lähtösolmu ja v kaaren maalisolmu
  - solmua v sanotaan solmun u vierussolmuksi (engl. adjacent vertex)
  - suunnatun verkon kaarista käytetään usein myös merkintää $u \rightarrow v$
- Sivun 424 suunnatussa verkossa siis esim. solmun e vierussolmuja ovat b ja d sillä $e \rightarrow b$ ja $e \rightarrow d$

  solmun b vierussolmuja ovat d ja solmu itse, sillä $b \rightarrow d$ ja $b \rightarrow b$

- Esim: sivun 424 kuvan suunnatun verkon formaali määritelmä:
  - $V = \{a, b, c, d, e\}$
  - $E = \{(a, c), (b, b), (b, d), (c, b), (d, a), (d, c), (e, b), (e, d)\}$
- Edellä mainittiin että verkon kaaret muodostavat joukon, eli kahden solmun välillä ei määritelmän mukaan voi olla kahta samaan suuntaan kulkevaa kaarta
  - joissakin sovelluksissa tilanne poikkeaa tästä, ja on mielekästä sallia, että kahden solmun välillä on useita kaaria (ns. multiverkko)
- Suunnatun verkon solmujen u ja v välillä voi sen sijaan olla kaaret molempiin suuntiin $u \to v$ ja $v \to u$
- Suuntaamattomassa verkossa kaarten joukko E on symmetrinen (engl. symmetric), eli jos $(u, v) \in E$ niin myös $(v, u) \in E$
  - merkitsemme myös suuntaamattoman verkon kaaria joskus $u \to v$
  - jos $(u, v) \in E$ sanotaan että solmut u ja v ovat vierekkäisiä, (engl. adjacent) eli v on u:n vierussolmu ja u on v:n vierussolmu
- Esim: sivun 425 suuntaamaton verkko formaalisti määriteltynä:
  - $V = \{a, b, c, d, e\}$
  - $E = \{(a, c), (c, a), (b, d), (d, b), (c, b), (b, c), (d, a), (a, d), (e, b), (b, e), (e, d), (d, e)\}$

- Usein verkon kaariin liitetään paino (engl. weight)



- Oletetaan että kaaripainot ovat kokonaislukuja
- Kaaripainon käsite määritellään funktiona $w : E \rightarrow \{0, 1, 2 \ldots\}$
- Eli funktio w liittää jokaiseen kaareen painon, esim. kuvan suunnatussa verkossa $w(a, c) = 7$, $w(e, d) = 3$ jne.
- Painotetun verkon kaarista $(u, v) \in E$ käytetään myös merkintää $u \stackrel{w(u,v)}{\rightarrow} v$, eli esimerkissämme on kaari $a \stackrel{7}{\rightarrow} c$
- Kaaripainoilla voidaan ilmaista esim. solmujen välisiä etäisyyksiä, niiden välisen yhteyden hintaa ym., painon ei siis välttämättä tarvitse olla kokonaisluku

- Solmujono $v_1, v_2, \ldots, v_n$ on polku (engl. path) solmusta v_1 solmuun v_n jos $v_1 \to v_2, v_2 \to v_3, \ldots, v_{n-1} \to v_n$

- Jos solmusta u on polku solmuun v käytetään merkintää $u \leadsto v$

- Jos $u \leadsto v$ sanotaan että solmu v on saavutettavissa (engl. reachable) solmusta u

- Polun pituus on polkuun liittyvien kaarien lukumäärä (polun $u \to u$ pituus on 0)

- Painotetussa verkossa polun paino on polun kaarien yhteenlaskettu paino
  - Huom: Painotetussa verkossa polun painoa kutsutaan usein myös polun pituudeksi (ja näin mekin teemme)

- Polku on yksinkertainen (engl. simple), jos kukin solmu esiintyy polussa vain kerran, paitsi viimeinen ja ensimmäinen saavat olla sama solmu

- Yksinkertainen polku on sykli eli kehä (engl. cycle) jos viimeinen ja ensimmäinen solmu ovat samat

- Sivun 428 suunnatun painotetun verkon polkuja:
  - $e \xrightarrow{3} d \xrightarrow{4} c \xrightarrow{1} b$ on yksinkertainen syklitön polku jonka pituus on 3 ja paino 8
  - $d \xrightarrow{9} a \xrightarrow{7} c \xrightarrow{1} b \xrightarrow{3} d$ on sykli jonka pituus on 4 ja paino 20
  - $c \xrightarrow{1} b \xrightarrow{0} b \xrightarrow{0} b \xrightarrow{3} d$ on polku jonka pituus 4, paino 4 ja joka sisältää kaksi sykliä
- Suunnattu verkko on syklitön (engl. acyclic) jos se ei sisällä yhtään sykliä
- Verkon ei välttämättä tarvitse olla yhtenäinen, eli verkko voi koostua useista erillisistä osista
- Suuntaamaton verkko on yhtenäinen (engl. connected), jos $u \leadsto v$ kaikilla $u, v \in V$
- Suunnattu verkko on vahvasti yhtenäinen (engl. strongly connected), jos $u \leadsto v$ kaikilla $u, v \in V$. ''Vahva'' korostaa, että kaarten suuntauksia on kunnioitettava

- Huom: Englannin kielessä syklittömästä suunnatusta verkosta käytetään lyhennettä DAG (directed acyclic graph)

- Vapaa puu (engl. free tree) on suuntaamaton verkko, joka on sekä syklitön että yhtenäinen. Puussa minkä tahansa kahden solmun välillä on tasan yksi yksinkertainen polku

- Juurellinen puu (engl. rooted tree) on usein luontevaa esittää muodostamalla vastaava vapaa puu ja suuntaamalla sitten kaaret juurta kohti

Esimerkkejä verkoista ja verkko-ongelmista

- **Tietokoneverkon yhtenäisyys**:
  solmut: tietokoneita
  kaaret: tietoliikenneyhteyksiä; ei suuntausta, ei yleensä painoja
  ongelma: mitkä yhteydet ovat sellaisia, että niiden katkeaminen jakaisi verkon kahteen toisistaan eristettyyn osaan

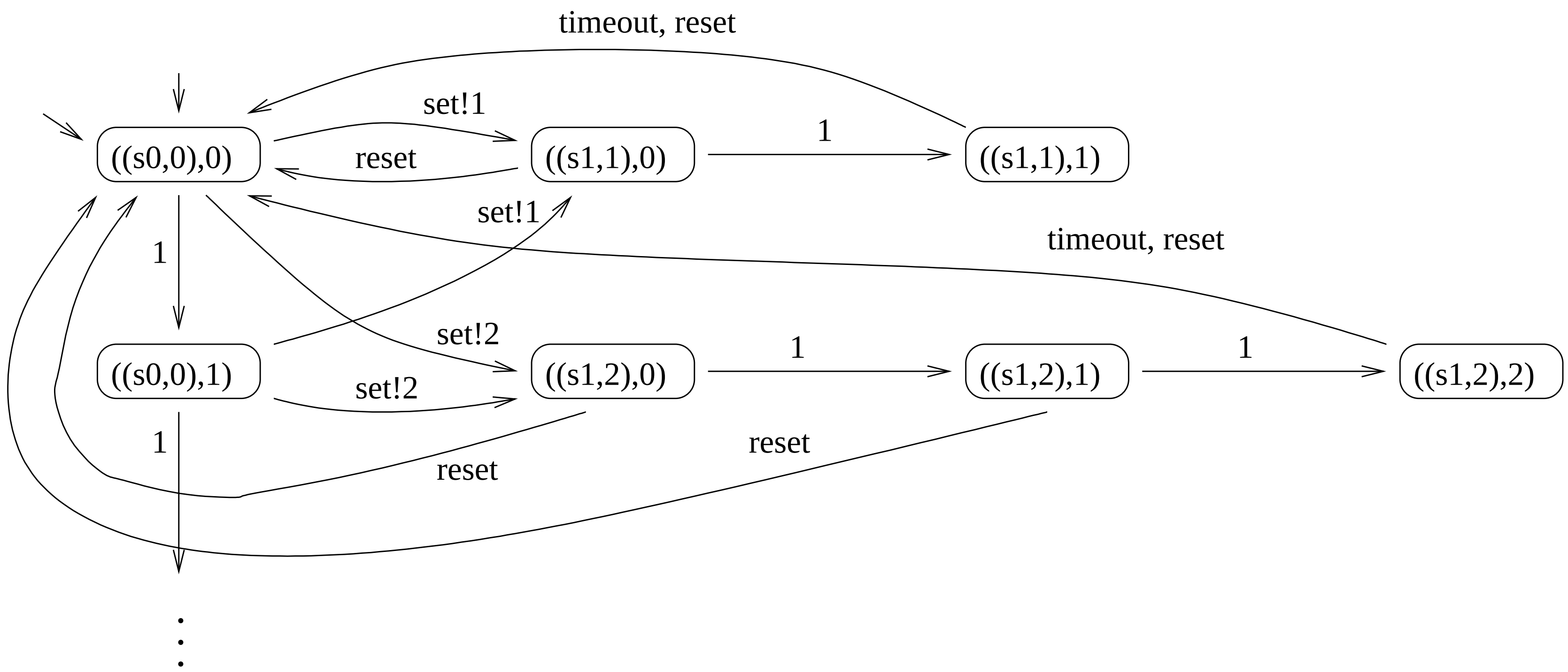
- **Robotin navigointi:**
  solmut: sopivalla tarkkuustasolla esitettyjä maantieteellisiä sijainteja
  kaaret: tunnettuja väyliä; ei suuntausta, ei painoja
  ongelma: ohjaa robotti paikasta A paikkaan B

- **Maantieverkosto:**
  solmut: kaupunkeja
  kaaret: maanteitä; ei suuntausta
  painot: kaupunkien välisiä etäisyyksiä
  ongelma: mikä on lyhin reitti kaupungista A kaupunkiin B

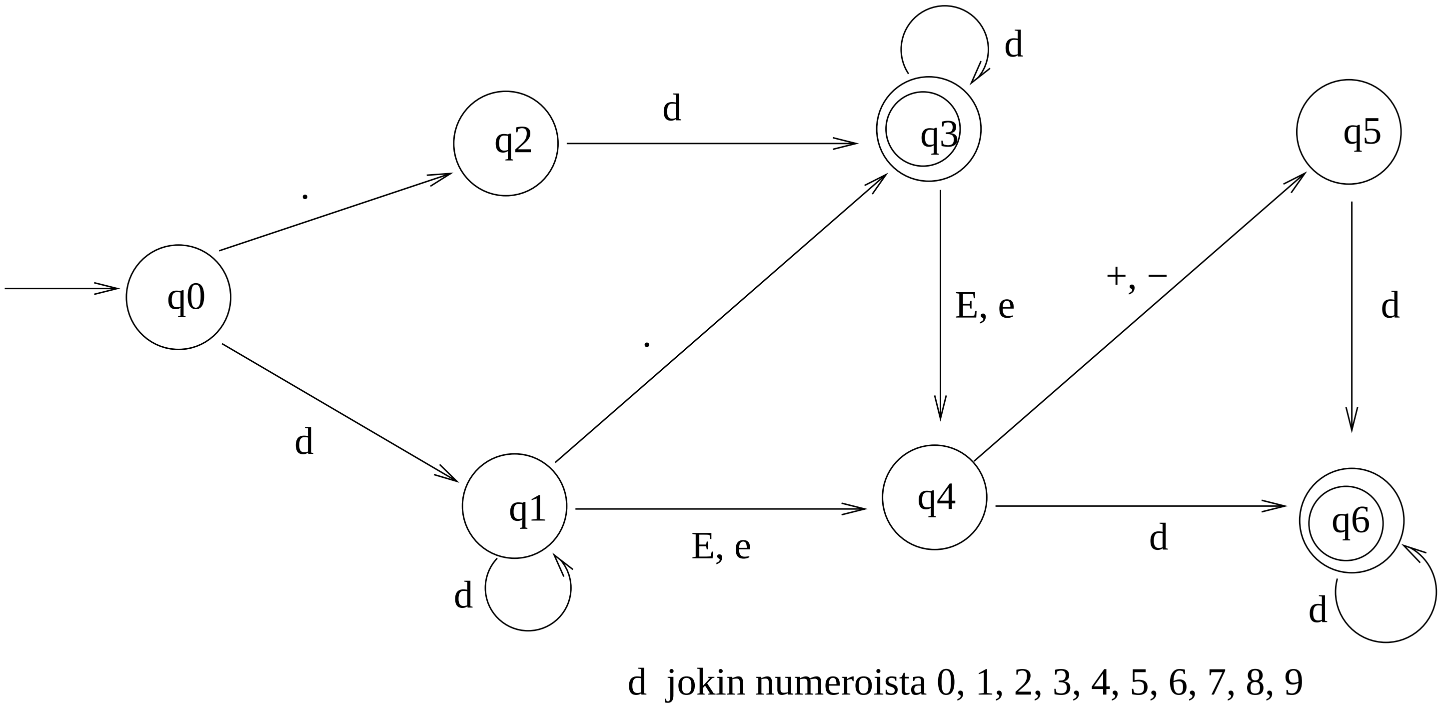
- **Logistiikkaverkosto**:
  solmut: varastoja
  kaaret: olemassaolevia kuljetusreittejä; ei suuntausta
  painot: useasta tavaralajista tieto, kuinka paljon sitä voidaan tietyssä ajassa kuljettaa mitäkin reittiä pitkin
  ongelma: miten saadaan halutut määrät tavaroita kulkemaan eri varastojen välillä

- **Tilasiirtymäverkko:**
  solmut: reaaliaikaisen järjestelmän tiloja
  kaaret: tilojen välisiä siirtymiä; suunnattu
  painot: mikä ulkoinen tapahtuma aiheuttaa minkäkin tilasiirtymän
  ongelma: voiko jokin tapahtumajono johtaa johonkin epätoivottuun tilaan tai tapahtumajonoon

- Esimerkki tilasiirtymäverkosta



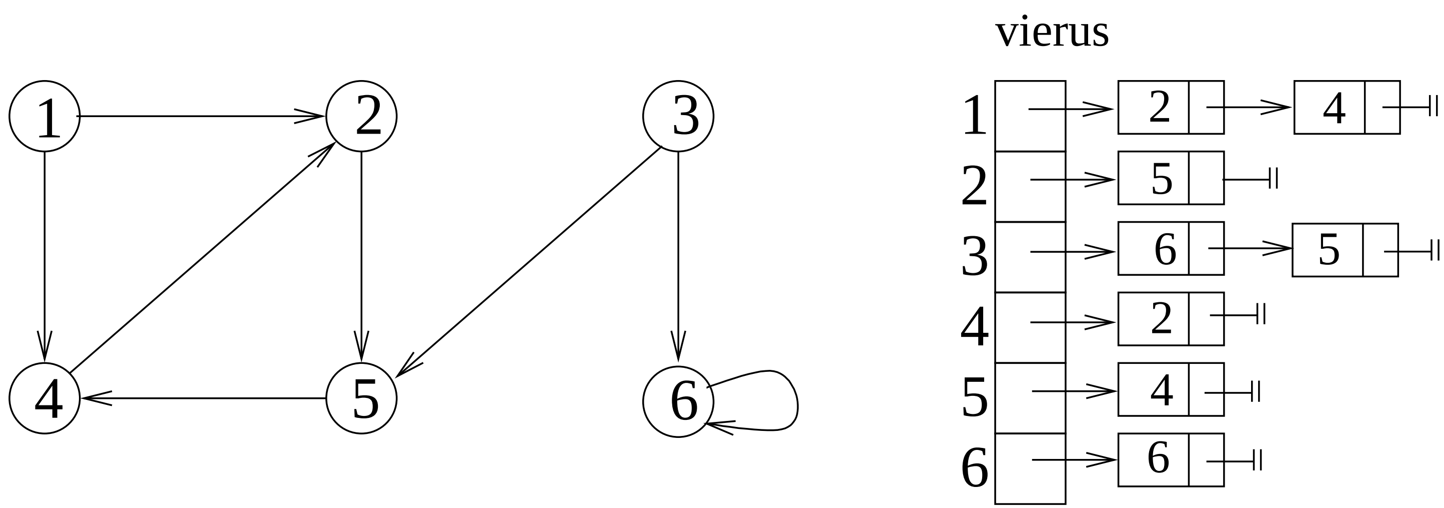
- **Äärellinen automaatti** (kurssilla Laskennan mallit):
  - solmut: abstraktin automaatin laskennan tilanteita
  - kaaret: abstraktin automaatin laskenta-askelia
  - painot: merkkejä
  - ongelma: Voiko tilasta A kulkea tilaan B siten, että kuljettujen kaarten painoista muodostuu haluttu merkkijono
- Esimerkki äärellisestä automaatista, joka määrittelee etumerkittömän Javan float -tyyppisen vakion syntaktisesti oikean muodon



Verkkojen tallettaminen

- Tarkastellaan seuraavassa tapoja verkon $G = (V, E)$ esittämiselle tietokoneohjelmassa
- Merkitään solmujen lukumäärää $|V|$ ja kaarien lukumäärää $|E|$
- Vaihtoehtoisia talletustapoja on kaksi:
  - vieruslistat (engl. adjacency lists)
  - vierusmatriisit (engl. adjacency matrices)
  - on myös tilanteita, joissa verkko on mielekkäämpi tallettaa jossain muussa muodossa tai verkkoa ei edes kannata tallentaa etukäteen
- Vieruslistaesityksessä verkko $G = (V, E)$ esitetään taulukkona $vierus$ joka sisältää $|V|$ kappaletta linkitettyjä listoja, yhden kullekin verkon solmulle
- Jokaiselle solmulle $u \in V$ lista $vierus[u]$ sisältää kaikki ne solmut joihin u:sta on kaari

- Esim: suunnattu verkko ja sen vieruslistaesitys

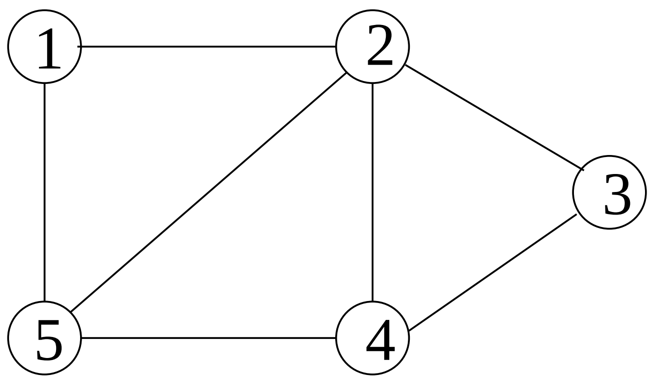


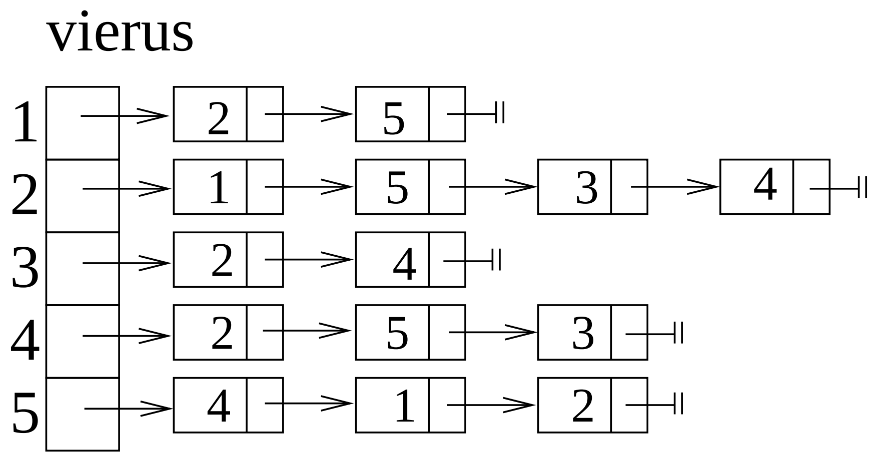
- Suunnatun verkon vieruslistojen yhteenlaskettu pituus on $|E|$ sillä jokainen kaari on talletettu kertaalleen yhteen vieruslistoista

- Koko vieruslistaesitys vie suunnattujen verkkojen tapauksessa tilaa $\mathcal{O}(|E| + |V|)$, sillä kaarien lisäksi varataan luonnollisesti tila taulukolle *vierus*

- Javassa vieruslistat voitaisiin esittää taulukollisena LinkedList-olioita

- Esim: suuntaamaton verkko ja sen vieruslistaesitys

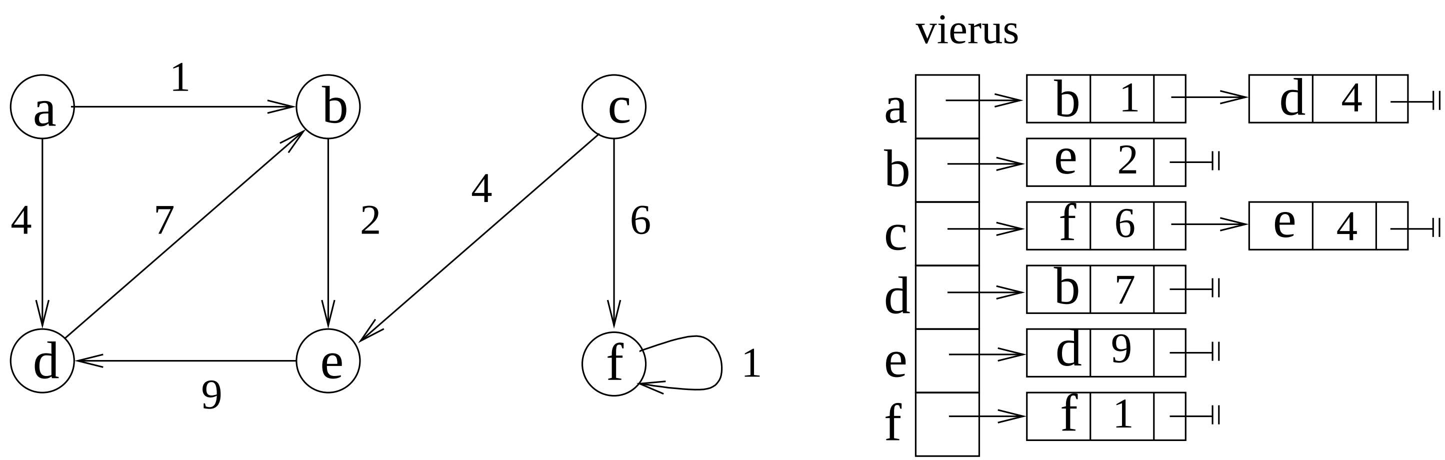




- Suuntaamattoman verkon vieruslistojen yhteenlaskettu pituus $|E|$ on kaksi kertaa kaarien lukumäärä, sillä jokainen kaari on talletettu kahteen vieruslistaan

- Koko vieruslistaesitys vie suuntaamattomien verkkojen tapauksessa tilaa $\mathcal{O}(|V| + 2 * |E|) = \mathcal{O}(|V| + |E|)$

- Myös kaarien painot voidaan tallentaa vieruslistarakenteeseen:



- Vieruslistaesityksen hyvä puoli on siis kohtuullinen tilantarve joka on $\mathcal{O}(|E| + |V|)$, eli lineaarinen suhteessa solmujen ja kaarten määrään

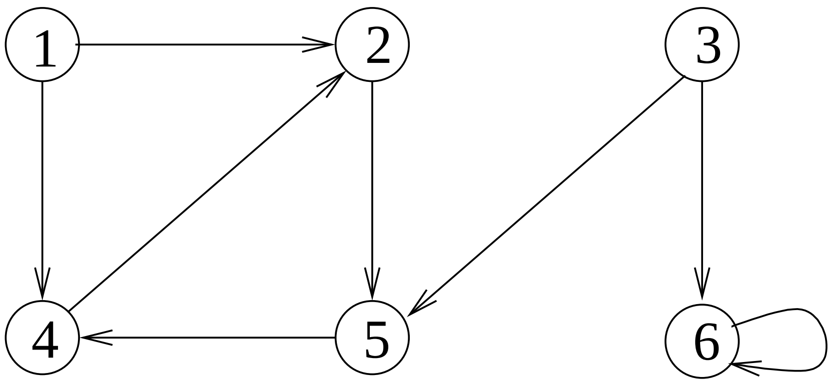
- Huononа puolena taas se että tieto onko verkossa kaarta $u \to v$ ei ole suoraan saatavilla, vaan vaatii vieruslistan $vierus[u]$ läpikäynnin

- Pahimmillaan tämä operaatio vie aikaa $\Omega(|V|)$ sillä solmusta u voi olla pahimmassa tapauksessa kaari kaikkiin verkon solmuihin

- Verkon $G = (V, E)$ vierusmatriisiesityksessä oletetaan että solmut on numeroitu, eli esim: $V = \{1, 2, \ldots, n\}$

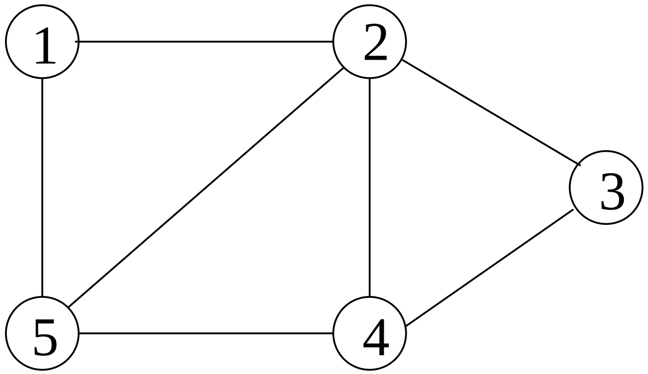
- Vierusmatriisi on $n \times n$-matriisi A, missä $A[i, j] = 1$ jos $(i, j) \in E$ ja muuten $A[i, j] = 0$

- Esimerkki suunnatusta verkosta:



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

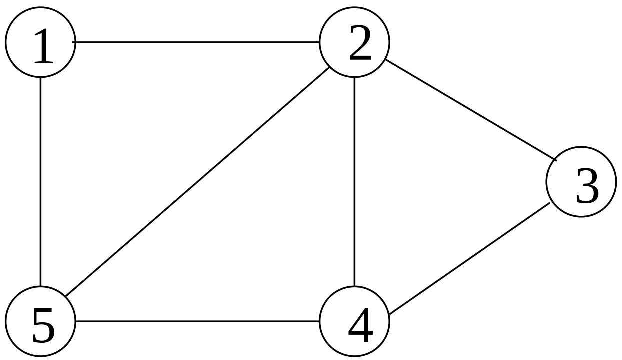
- Esimerkki suuntaamattomasta verkosta:



	1	2	3	4	5
1	0	**1**	0	0	**1**
2	**1**	0	**1**	**1**	**1**
3	0	**1**	0	**1**	0
4	0	**1**	**1**	0	**1**
5	**1**	**1**	0	**1**	0

- Suuntaamattoman verkon tapauksessa jokainen kaari on rekisteröity kahteen kertaan vierusmatriisiin, esim. koska $(2, 5) \in E$, niin $A[2, 5] = 1$ ja $A[5, 2] = 1$

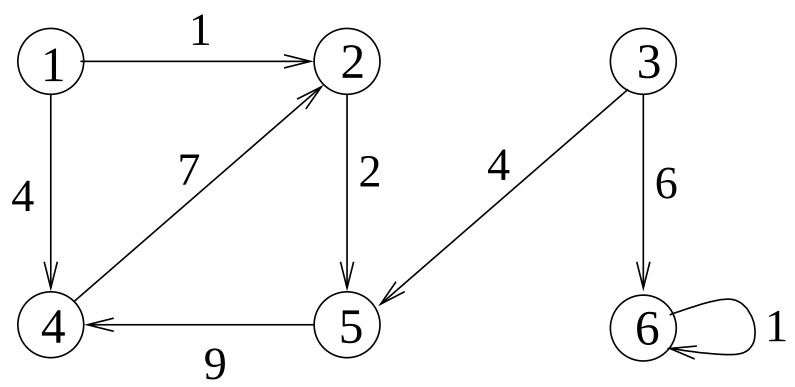
- Suuntaamattoman verkon tapauksessa riittäisikin siirtymämatriisista puolikas:



1	2	3	4	5	
0	**1**	0	0	**1**	1
	0	**1**	**1**	**1**	2
		0	**1**	0	3
			0	**1**	4
				0	5

- Joskus on vieläpä käytössä rajoitus että suuntaamattomassa verkossa kaaret muotoa (i, i) eivät ole sallittuja, jos näin on, ei vierusmatriisin diagonaalia (eli alkioita $A[1,1], A[2,2], \ldots$) myöskään tarvita

- Kaaripainojen tallettaminen vierusmatriisiin on vaivatonta:



	1	2	3	4	5	6
1	∞	**1**	∞	**4**	∞	∞
2	∞	∞	∞	∞	**2**	∞
3	∞	∞	∞	∞	**4**	**6**
4	∞	**7**	∞	∞	∞	∞
5	∞	∞	∞	**9**	∞	∞
6	∞	∞	∞	∞	∞	**1**

- Painotetun verkon vierusmatriisissa periaatteena siis on asettaa $A[i,j] = w(i,j)$ jos $(i,j) \in E$ ja muuten $A[i,j] = \infty$ tai $A[i,j] = 0$
  - jos kaaren $i \xrightarrow{x} j$ paino kuvaa reitin i:stä j:hin pituutta tai kustannusta, on ääretön luonnollinen valinta olemattomien kaarien merkintään
  - jos kaari taas kuvaa reitin kapasiteettia, on luonnollinen valinta olemattomien kaarien merkintään nolla

- Hyvänä puolena vierusmatriisissa on se että tietyn kaaren olemassaolo selviää matriisista vakioajassa

- Toisaalta solmun kaikkien kaarien selvittämiseen kuluu aikaa aina $\mathcal{O}(|V|)$ vaikka kaaria olisikin vain yksi

- Toinen huono puoli vierusmatriisissa on tilan tarve, matriisin koko on kaarien lukumäärästä riippumatta aina $|V| \times |V|$

- Verkkoa sanotaan harvaksi jos kaaria on suhteellisen vähän, esim. vain kaksi kertaa solmujen määrä
  - Kaarien määrä on tällöin $|E| = \mathcal{O}(|V|)$, ja vieruslistana esitetty verkko vie tilaa $\mathcal{O}(|E| + |V|) = \mathcal{O}(|V|)$ kun taas vierusmatriisiesityksen tilantarve on tähän verrattuna neliöinen $\mathcal{O}(|V| \times |V|)$

- Isot harvat verkot siis kannattanee tallentaa vieruslistoja käyttäen

- Osa verkkoalgoritmeista tosin olettaa että verkko on talletettu esim. käyttäen vierusmatriiseja, eli verkon talletusmuoto riippuu paljolti myös verkon käyttötarkoituksesta

Verkon muunlaiset esitystavat

- Vieruslista- ja vierusmatriisiesitys olettavat, että verkon solmut ja kaaret ovat eksplisiittisesti generoitu koneen muistiin

- Usein tämä ei ole tarpeen tai tarkoituksenmukaista

- Verkosta voi olla olemassa jokin muunlainen esitysmuoto ja verkko "verkkona" on olemassa vain ohjelmoijan taustalla olevana mentaalisena mallina

- Esim. tehtävänä on etsiä löytyykö merkkeinä kuvatusta labyrintista X:llä merkitystä kohdasta reittiä ulos

```
####.####
#..#....#
#.#.###.#
#....X..#
#########
```

- Lienee mielekästä tulkita mahdolliset sijaintipaikat eli pisteet solmuiksi
  - jokaisen vierekkäisen pistettä edustavan solmun välille tulee kaari
  - labyrintin seinät eli #-merkit taas ovat kaarettomia kohtia
- Labyrintti kannattanee tallettaa koneen muistiin kaksiulotteisena taulukkona jossa esim. 1 merkitsee seinätöntä kohtaa ja 0 seinää:

```
000010000
011011110
010100010
011111110
000000000
```

- Solmut vastaavat nyt taulukon kohtia, esim. yläreunan aukko labyrintista ulos on taulukon kohta lab[0,4]. Ensimmäinen indeksi siis tarkoittaa riviä ja toinen saraketta
- Kaarien olemassaolo selviää nyt suoraan taulukosta, eli esim
  - kohdasta lab[3,5] (paikka mistä aloitetaan, eli missä on alussa X) on kaari kohtaan lab[3,4] ja lab[3,6] koska molemmissa kohdissa taulukossa on 1
  - kohdasta lab[1,1] kaari kohtiin lab[1,2] ja lab[2,1]
- Verkkoa ei siis kannattane esittää vieruslista- tai vierusmatriisiesityksenä koska taulukosta lab selviää kaikki kaaria koskeva informaatio

Verkon läpikäynti

- Tyypillistä verkkoa käytettäessä on että halutaan kulkea verkossa systemaattisesti vieraillen kaikissa solmuissa tai ainakin kaikissa tietystä solmusta saavutettavissa olevissa solmuissa
- Solmujen läpikäyntiin on kaksi perusstrategiaa:
  - leveyssuuntainen läpikäynti (engl. bredth-first search), ja
  - syvyyssuuntainen läpikäynti (engl. depth-first search)
- Algoritmit tuottavat myös erilaista lisäinformaatiota verkon rakenteesta, joten niitä käytetään muissakin verkkoalgoritmeissa esiproseduurina tai rakennusalustana
- Toimivat sekä suunnatuille että suuntaamattomille verkoille (mutta tulosten tulkinta voi poiketa)

Leveyssuuntainen läpikäynti

- Verkon $G = (V, E)$ leveyssuuntaisessa läpikäynnissä (engl. breadth-first search) tutkitaan mitkä verkon solmuista ovat saavutettavissa annetusta aloitussolmusta $s \in V$

- Läpikäynti etenee uusiin solmuihin "taso kerrallaan", eli ensin etsitään mitkä solmut saavutetaan s:stä yhden pituista polkua käyttäen, tämän jälkeen edetään solmuihin mitkä ovat saavutettavissa s:stä kahden mittaista polkua käyttäen, jne

- Algoritmin sivutuotteena selvitetään mikä on polun pituus aloitussolmusta s kuhunkin läpikäynnin aikana löydettyyn solmuun v. Tieto talletetaan taulukkoon $distance$

- Toisena sivutuotteena algoritmi muodostaa verkkoa läpikäydessään leveyssuuntaispuuta (engl. bredth-first tree)
  - puu kertoo mitä reittiä läpikäynti on edennyt kuhunkin solmuun v
  - algoritmin suorituksen jälkeen puun polku solmusta s solmuun v vastaa verkon lyhintä polkua $s \leadsto v$
  - puun kaaret talletetaan taulukkoon $tree$, ja taulukon alkio $tree[v]$ kertoo mistä solmusta lyhin polku solmuun v saapuu

- Algoritmin kirjanpitoa varten verkon solmuihin tarvitaan vielä kolmaskin taulukko, $color$, jossa pidetään kirjaa solmujen "väreistä"

  taulukon alkio $color[v]$ kertoo onko läpikäynti jo löytänyt solmun v

  – solmut joita läpikäynti ei ole vielä löytänyt ovat valkoisia, eli niille $color[v] =$ white

  – jos läpikäynti on ehtinyt solmuun v, mutta solmusta u edelleen vieviä kaaria ei ole vielä käsitelty, niin solmun väriksi asetetaan harmaa $color[v] =$ gray

  – kun solmusta v lähtevät kaaret on käsitelty (jolloin koko solmu v on käsitelty), asetetaan sen väriksi musta, eli $color[v] =$ black

- Aluksi kaikille paitsi aloitussolmulle s merkitään $color[v] =$ white

- Aloitussolmulle s merkitään $color[s] =$ gray

- Algoritmi käyttää aputietorakenteenaan jonoa Q joka on aluksi tyhjä; jonossa ovat tietyllä hetkellä ne solmut jotka läpikäynti on jo löytänyt, mutta joiden naapurisolmuja ei vielä ole käsitelty

- Rivien 1-7 alustusvaiheen jälkeen aloitussolmu s on merkitty harmaaksi ja laitettu jonoon

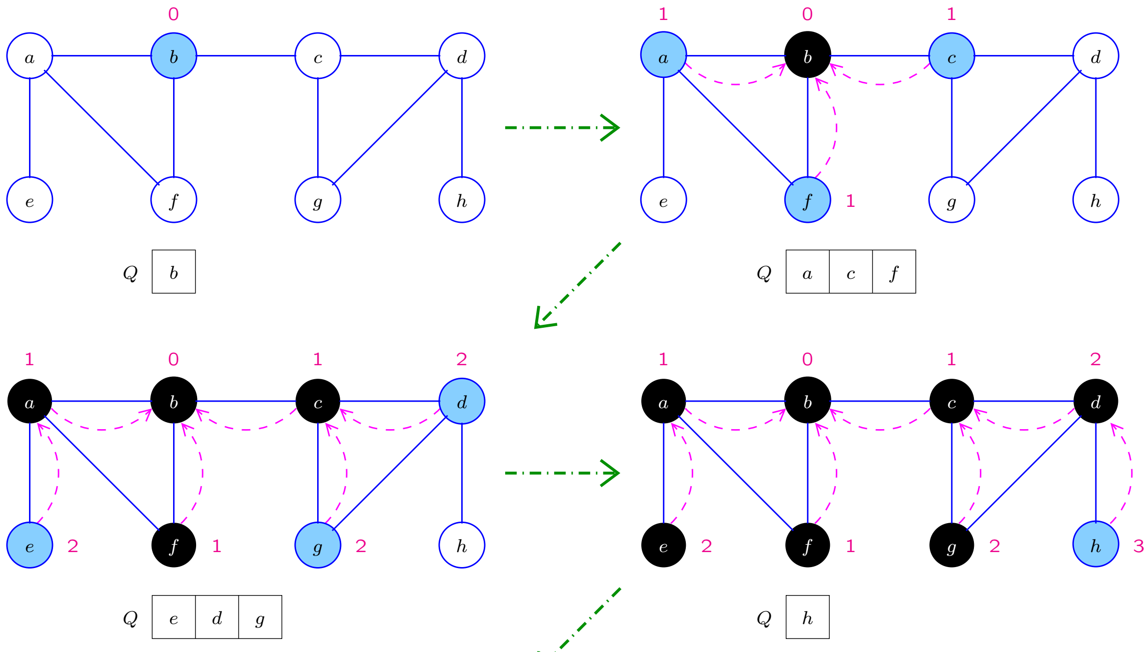
- Rivien 8-16 toimintaperiaate
  - aloitussolmu s otetaan jonosta, ja kaikki sen vierussolmut laitetaan jonoon
  - vierussolmujen etäisyydeksi päivitetään 1, niitä leveyssuuntaispuussa edeltäväksi solmuksi asetetaan s ja merkitään solmut harmaiksi
  - solmu s on nyt käsitelty ja se muuttuu mustaksi
  - tämän jälkeen niin kauan kun jonossa on solmuja, otetaan käsittelyyn jonon alussa oleva solmu u
  - laitetaan jonoon ne u:n vierussolmut, joita etsintä ei ole vielä kohdannut, eli joille $color[v] =$ white ja
  - päivitetään jonoon laitettujen etäisyys- ja leveyssuuntaispuutietoa sekä merkitään ne harmaiksi
  - kun solmu u on käsitelty valmiiksi se merkitään mustaksi

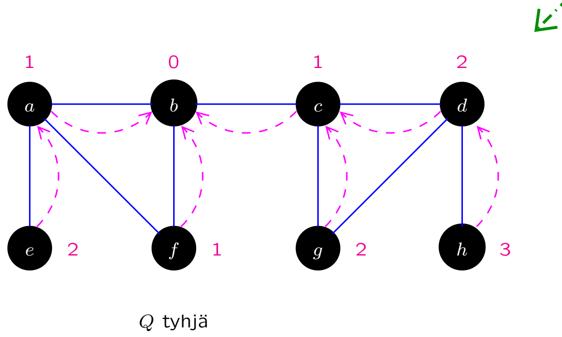
- Algoritmi

```
BFS(G,s)
for jokaiselle solmulle u ∈ V
    color[u] = white
    distance[u] = ∞
    tree[u] = NIL
color[s] = gray
distance[s] = 0
enqueue(Q,s)
while ( not empty(Q) )
    u = dequeue(Q)
    for jokaiselle solmulle v ∈ vierus[u]     // kaikille u:n vierussolmuille v
        if color[v]==white                     // solmua v ei vielä löydetty
            color[v] = gray
            distance[v] = distance[u]+1
            tree[v] = u
            enqueue(Q,v)
    color[u] = black
```

- Algoritmi toimii sekä suunnatuilla että suuntaamattomilla verkoilla

- Esimerkki algoritmin toiminnasta seuraavalla sivulla (harmaat solmut ovat kuvassa sinisiä)





Suuntaamattoman verkon leveyssuuntainen läpikäynti

vieruslistojen oletettu olevan aakkosjärjestyksessä

- Algoritmin suorituksen jälkeen lyhin polku $s \leadsto v$ saadaan selville seuraavasti:
  - $tree[v]$ kertoo minkä solmun kautta lyhin polku $s \leadsto v$ saapuu solmuun v
  - solmuun $tree[v]$ lyhin polku saapuu solmun $tree[tree[v]]$ kautta, jne
  - laitetaan pinoon $tree[v]$, $tree[tree[v]]$, $tree[tree[tree[v]]]$ ja tulostetaan pinon sisältö
  - näin saadaan tulostettua polulla koko polku $s \leadsto v$ alusta loppuun

- Algoritmina:

```
shortest-path(G,v)
u = tree[v]
while u ≠ s
    push(S,u)
    u = tree[u]
print( "lyhin polku solmusta s solmuun v")
while not empty(S)
    u = pop(S)
    print(u)
```

- Leveyssuuntaisen läpikäynnin aikavaativuus:
  - alustukseen (rivit 1-6) kuluu aikaa $\mathcal{O}(|V|)$
  - koska jonoon laitettava solmu värjätään harmaaksi eikä väri enää muutu takaisin valkoiseksi, takaa rivin 11 testi että jokainen solmu laitetaan jonoon vain kerran
  - jokainen solmu siis myös poistetaan jonosta korkeintaan kerran
  - enqueue ja dequeue-operaatiot voidaan toteuttaa ajassa $\mathcal{O}(1)$, eli kokonaisuudessaan jono-operaatioihin kuluu aikaa $\mathcal{O}(|V|)$
  - kunkin solmun vieruslista käydään läpi ainoastaan silloin kuin solmu poistetaan jonosta, eli korkeintaan kerran
  - vieruslistojen yhteispituus on $\mathcal{O}(|E|)$, eli yhteensä vieruslistojen läpikäyntiin käytetään aikaa korkeintaan $\mathcal{O}(|E|)$
- Kokonaisuudessaan aikaa siis kuluu $\mathcal{O}(|V| + |V| + |E|)$ eli $\mathcal{O}(|V| + |E|)$
- Tilavaativuus algoritmilla on $\mathcal{O}(|V|)$ sillä pahimmassa tapauksessa aloitussolmusta on kaari kaikkiin verkon solmuihin, ja tässä tapauksessa jono Q tulisi sisältämään kaikki verkon solmut; myös aputaulukot $color$, $tree$ ja $distance$ kuluttavat tilaa $\mathcal{O}(|V|)$

Leveyssuuntaisen läpikäynnin oikeellisuus

- Tarkastelemme nyt algoritmin oikeellisuuden osoittamista esimerkkinä ajattelutavasta, josta on apua hankalampien algoritmien ymmärtämisessä
- Väitämme, että algoritmin suorituksen jälkeen kaikilla solmuilla u pätee, että
  - jos u on saavutettavissa solmusta s, niin u on musta ja $distance[u]$ on lyhimmän polun pituus $s \rightsquigarrow u$
  - muuten u on valkoinen ja $distance[u] = \infty$.

- Värityksiä koskeva osa algoritmin toiminnasta on helppo todeta oikeaksi
- Algoritmista nähdään suoraan, että while-silmukassa pätee invariantti
  - jos solmu on valkoinen, se ei ole käynytkään jonossa
  - jos solmu on harmaa, se on parhaillaan jonossa
  - jos solmu on musta, se on poistettu jonosta
  - jos solmu on musta, sen viereiset solmut ovat harmaita tai mustia
- Erityisesti algoritmin päättyessä
  - harmaita solmuja ei ole, koska jono on tyhjä
  - lähtösolmu on musta
  - jos $s \leadsto u$ ja u olisi valkoinen, niin polku hyppäisi jossain kohdassa mustasta valkoiseksi

  $\Rightarrow$ kaikki saavutettavissa olevat solmut ovat mustia

- Tehdään toisaalta vastaoletus, että algoritmi värittää harmaaksi ainakin yhden solmun, joka ei ole saavutettavissa solmusta s

- Olkoon v näistä algoritmin suoritusjärjestyksessä ensimmäinen harmaaksi väritettävä

- Ennen kuin v voidaan värittää harmaaksi, algoritmin on pitänyt värittää harmaaksi jokin solmu u, jolla $v \in vierus[u]$

- Koska oletuksen mukaan v oli ensimmäinen ''väärin'' väritetty, solmu u on saavutettavissa

- Mutta oletuksen $v \in vierus[u]$ mukaan myös v on nyt saavutettavissa; ristiriita

- Siis solmu väritetään algoritmin kuluessa harmaaksi, jos ja vain jos se on saavutettavissa solmusta s

- Koska kaikki harmaat solmut tulevat mustiksi ennen suorituksen loppua, niin värien osalta algoritmi toimii väitetyllä tavalla

- Polkujen pituuksien tarkastelemiseksi olkoon $D(u)$ lyhimmän polun pituus lähtösolmusta solmuun u

- Lisäksi merkitään $D(u) = \infty$, jos polkua ei ole

- Väitämme siis, että lopuksi $\mathit{distance}[u] = D(u)$ kaikilla u

- On helppo nähdä, että algoritmi säilyttää invariantin $\mathit{distance}[u] \geq D(u)$ kaikilla u

- Aluksi $\mathit{distance}[u] = \infty$ ja invariantti selvästi pätee

- Kun algoritmi myöhemmin päivittää $\mathit{distance}[v] = \mathit{distance}[u] + 1$, niin $(u, v) \in E$

- Tällöin $D(v) \leq D(u) + 1$, ja yhtäsuuruus pätee, jos jokin lyhin polku $s \leadsto v$ kulkee solmun u kautta

- Jos siis $\mathit{distance}[u] \geq D(u)$, niin $\mathit{distance}[v] = \mathit{distance}[u] + 1 \geq D(u) + 1 \geq D(v)$

- Tämä perustuu siihen, että algoritmin laskema arvo $\mathit{distance}[u]$ on jonkin polun pituus $s \leadsto u$

- Ongelmaksi jää osoittaa, että algoritmi todella löytää lyhimmän polun
- Algoritmin **BFS** tapauksessa tämä on melko suoraviivaista, koska (kuten kohta perustellaan) algoritmi löytää solmut arvon $D(u)$ mukaan kasvavassa järjestyksessä
- Myöhemmin kohtaamme vastaavan tilanteen painotetuissa verkoissa, jolloin dynamiikka on monimutkaisempi

- Analysoimme arvojen $distance[u]$ laskemista jakamalla suorituksen vaiheisiin: Vaihe k päättyy, kun viimeisen kerran muutetaan mustaksi solmu u, jolla $distance[u] = k - 1$

- Lisäksi sovimme, että vaihe 0 koostuu alustuksista ennen while-silmukan alkua

  Todistamme induktiolla arvon k suhteen, että vaiheen k päättyessä

  - jos $D(u) < k$, niin u on musta (eli poistunut jonosta)
  - jos $D(u) = k$, niin u on harmaa (eli parhaillaan jonossa)
  - jos $D(u) > k$, niin u on valkoinen (eli ei ole vielä ollut jonossa)
  - jos u on harmaa tai musta, niin $distance[u] = D(u)$

- Alustusten jälkeen väite selvästi pätee

- Oletetaan nyt, että väite pätee vaiheen k päättyessä (induktio-oletus)
- Siis jonossa on tasan ne solmut u, joilla $D(u) = k$
- Määritelmän mukaan $D(v) = k + 1$, jos ja vain jos
  - $v \in vierus[u]$ jollain u, jolla $D(u) = k$, ja
  - $v \notin vierus[w]$, jos $D(w) < k$
- Täsmälleen nämä solmut viedään jonoon vaiheen $k + 1$ aikana:
  - kaikki ehdon $D(u) = k$ täyttävät vieruslistat $vierus[u]$ käydään läpi
  - jos $v \in vierus[w]$ missä $D(w) < k$, niin induktio-oletuksen mukaan w on käynyt jonossa aiemmilla kierroksilla, jolloin v on muutettu harmaaksi
- Siis ehto pätee vaiheen $k + 1$ jälkeen

- Algoritmin päättyessä edellä todetun mukaan
  - kaikki lähtösolmusta saavutettavat solmut ovat mustia ja
  - kaikille mustille solmuille u pätee $distance[u] = D(u)$

- Toisaalta muut kuin saavutettavat solmut u ovat valkoisia, ja niillä on voimassa alkuasetus $distance[u] = \infty$

- Siis lopuksi $distance[u] = D(u)$ kaikilla u, kuten haluttiin

- Tarkempi tarkastelu osoittaa, että algoritmin toiminnan kannalta emme oikeastaan tarvitse harmaaksi merkitsemistä, vaan voisimme merkitä solmut suoraan mustiksi, kun niihin tullaan

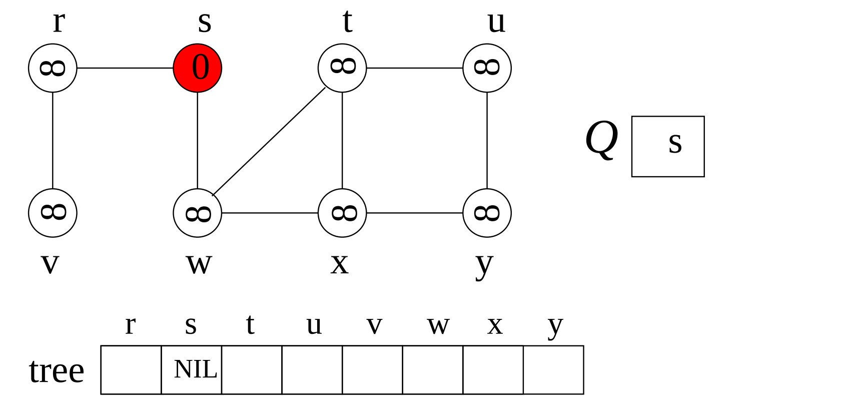
- Algoritmi olisi nyt tällainen:

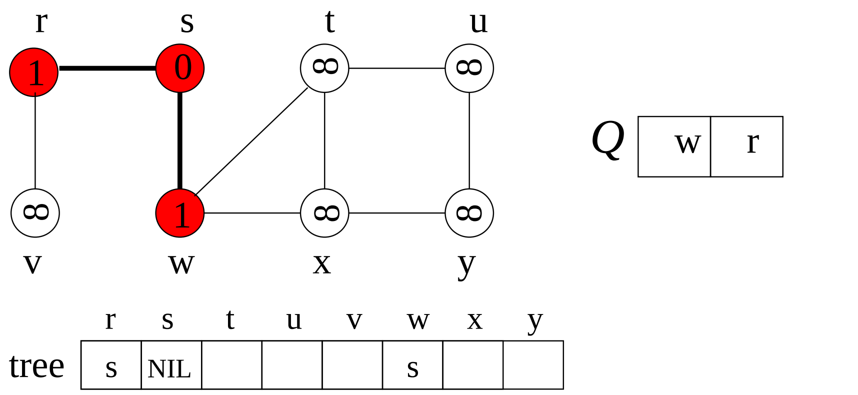
BFS2(G,s)

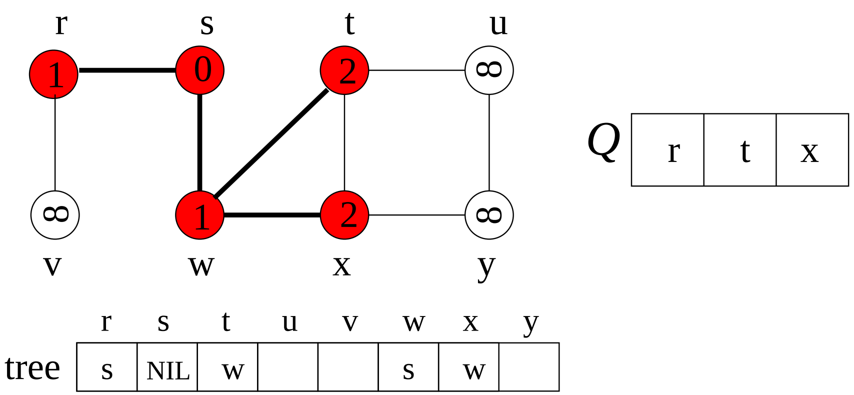
```
for jokaiselle solmulle u ∈ V
    color[u] = white
    distance[u] = ∞
    tree[u] = NIL
color[s] = black
distance[s] = 0
enqueue(Q,s)
while ( not empty(Q) )
    u = dequeue(Q)
    for jokaiselle solmulle v ∈ vierus[u]     // kaikille u:n vierussolmuille v
        if color[v]==white                    // solmua v ei vielä löydetty
            color[v] = black
            distance[v] = distance[u]+1
            tree[v] = u
            enqueue(Q,v)
```

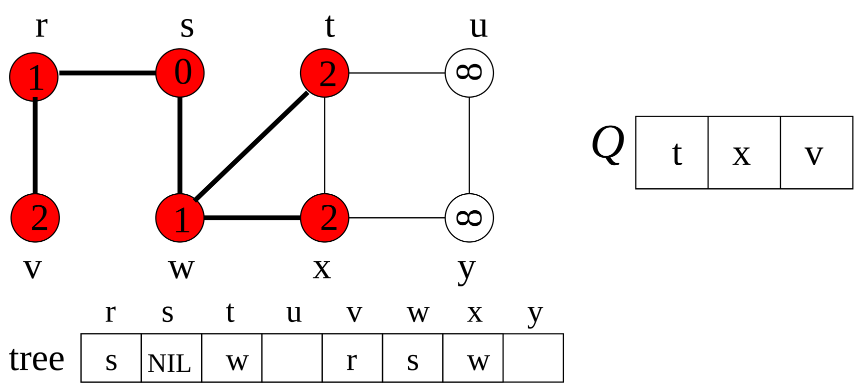
- Käydään vielä yksi esimerkki läpi, käyttäen tätä algoritmia (mustat solmut ovat värillisillä sivuilla punaisia)

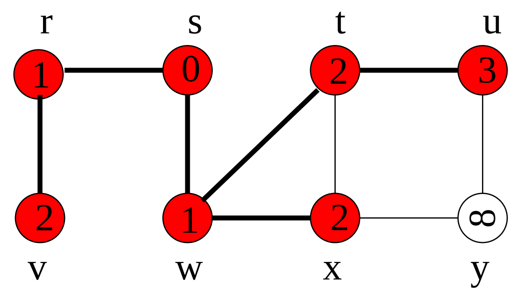
- Taulukon *tree* alkio *tree*[v] siis kertoo mistä solmusta lyhin polku lähtösolmusta solmuun v saapuu





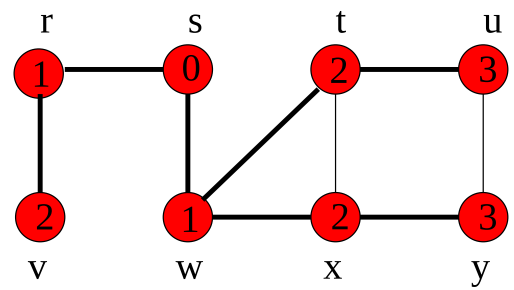






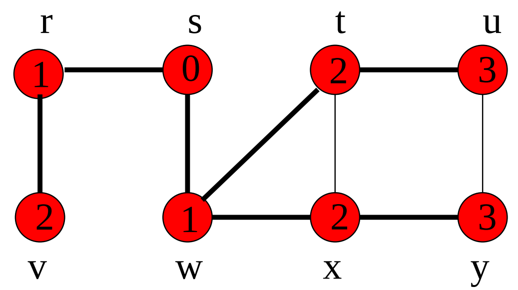
Q: x | v | u

	r	s	t	u	v	w	x	y
tree	s	NIL	w	t	r	s	w	

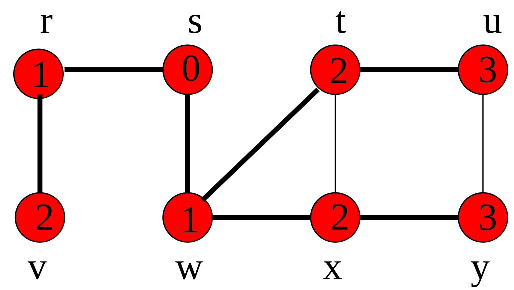


Q: v | u | y

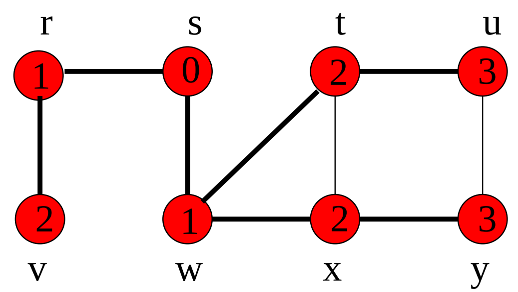
	r	s	t	u	v	w	x	y
tree	s	NIL	w	t	r	s	w	x



Q: u | y



Q: y



Q

Syvyyssuuntainen läpikäynti

- Toinen verkkojen läpikäyntitavoista on siis syvyyssuuntainen läpikäynti
- Strategiana on nyt edetä aloitussolmusta s yhtä polkua niin pitkälle kuin mahdollista
- Kun tullaan solmuun josta ei enää päästä uusiin, vielä tutkimattomiin solmuihin, peruutetaan tutkitulla polulla lähimpään sellaiseen solmuun josta lähtee vielä tutkimaton haara
- Näin löydetään kaikki solmusta s saavutettavissa olevat solmut
- Syvyyssuuntainen etsintä saadaan aikaan korvaamalla leveyssuuntaisen etsinnän jono pinolla
- Jos halutaan käydä läpi kaikki verkon solmut ja verkossa on solmuja jotka eivät ole saavutettavissa solmusta s, valitaan yksi saavuttamattomissa olevista solmuista ja käynnistetään uusi läpikäynti

- Myös syvyyssuuntainen läpikäynti värjää solmuja samalla tavalla kuin leveyssuuntaisessa läpikäynnissä:
  - solmut joita ei ole löydetty ovat valkoisia
  - kun solmu löydetään, se asetetaan harmaaksi
  - kun solmun kaikkien vierussolmujen käsittelystä on palattu, tulee solmusta musta
- Harmaa väri siis tarkoittaa, että solmu on jo löydetty, mutta sen käsittely ei ole vielä kokonaisuudessaan ohi

- Seuraavassa algoritmi, joka selvittää aloitussolmusta s saavutettavissa olevat solmut

```
DFS(G,s)
for jokaiselle solmulle u ∈ V
    color[u] = white
DFS-visit(G,s)

DFS-visit(G,u)
color[u] = gray
for jokaiselle solmulle v ∈ vierus[u]     // kaikille u:n vierussolmuille v
    if color[v]==white                    // solmua v ei vielä löydetty
            DFS-visit(G,v)
color[u] = black
```

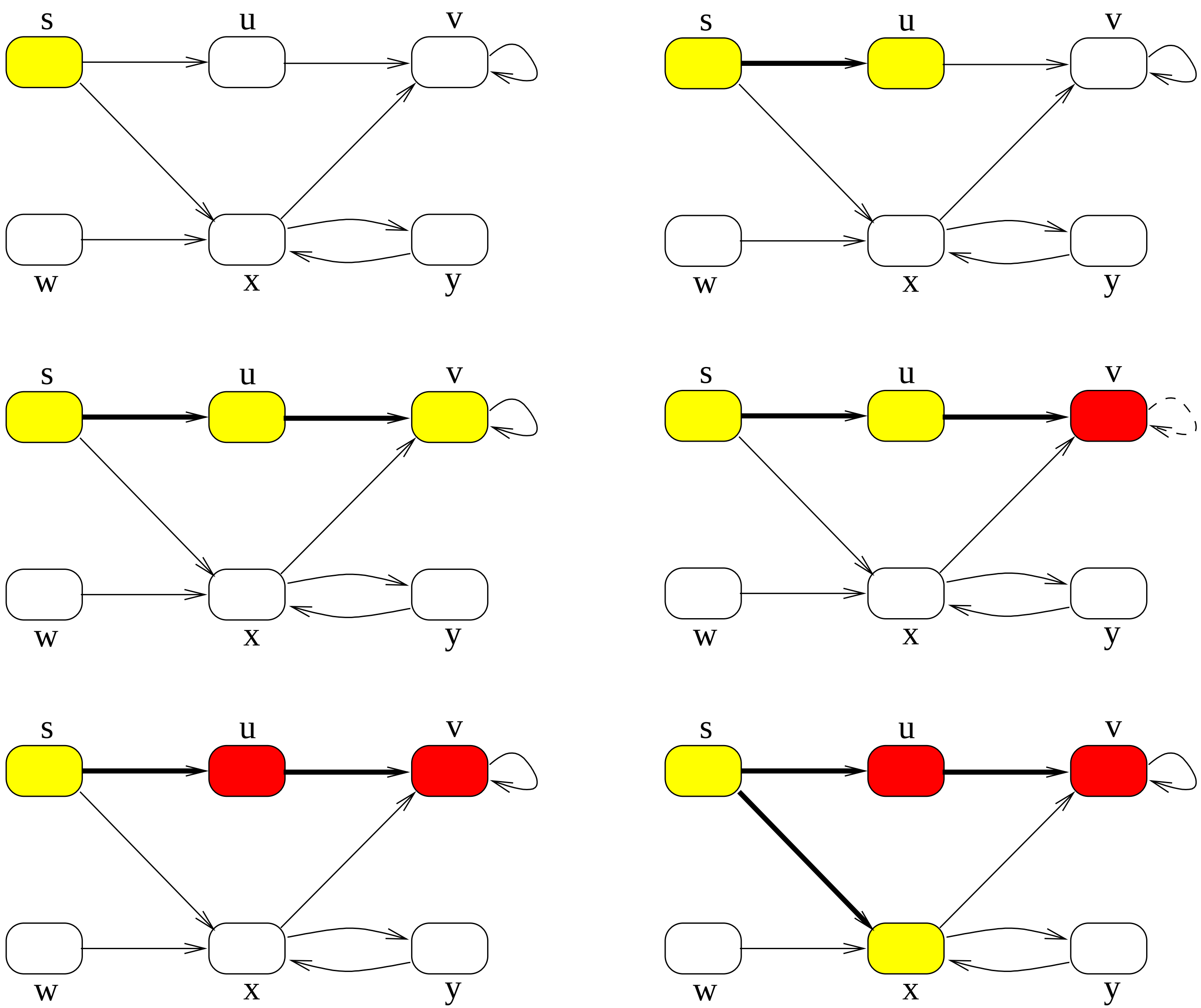
- Harmaa ei ole tässäkään tarpeen algoritmin toiminnan kannalta, eli solmu voitaisiin värjätä heti mustaksi

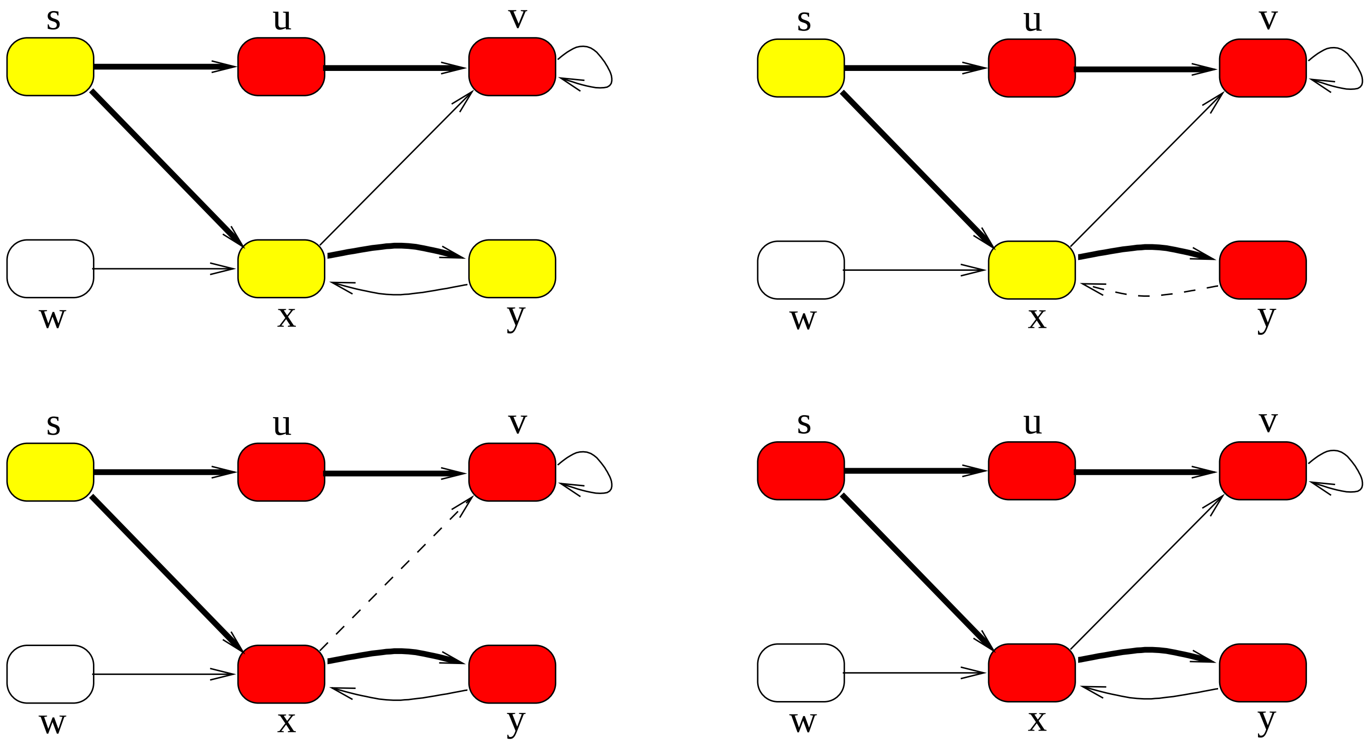
- Tarvitsemme harmaata väriä kohta esitettävässä syvyyssuuntaisen läpikäynnin sovelluksessa (syklittömyyden tarkastus), joten jätämme sen algoritmiin

- Tulemme taas osoittamaan tarkasti algoritmin oikeellisuuden. Sitä varten tarvitsemme lisämuuttujia, mutta palaamme siihen myöhemmin

- Toimintaperiaate:
  - alustusvaiheessa kaikki solmut merkataan löytymättömiksi eli valkoisiksi
  - läpikäynti aloitetaan kutsumalla **DFS-visit** aloitussolmulle s
  - kun läpikäynti etenee solmuun, merkataan että solmu on löydetty ja että sen käsittely on kesken eli solmu muuttuu harmaaksi (rivi 4)
  - jokaiselle solmun vierussolmulle jota ei ole vielä löydetty, eli valkoisille solmuille kutsutaan rekursiivisesti **DFS-visit**:iä (rivit 5-7)
  - kun kaikki vierussolmut on käsitelty, merkataan solmu käsitellyksi eli mustaksi (rivi 8) ja rekursiivinen funktio päättyy

- Esimerkki algoritmin toiminnasta seuraavalla sivulla. Värillisessä kuvassa harmaat solmut ovat keltaisia ja mustat punaisia


s u v
w x y
s u v
w x y
s u v
w x y
s u v
w x y
s u v
w x y
s u v
w x y


s
u
v
w
x
y
s
u
v
w
x
y
s
u
v
w
x
y
s
u
v
w
x
y

- Algoritmi siis merkkaa ensin $color[v]$ = gray ja lopulta $color[v]$ = black kaikille aloitussolmusta s saavutettavissa oleville solmuille
  - solmu v pysyy harmaana niin kauan kuin haku etenee solmuissa, jotka ovat v:stä saavutettavissa
  - kun kaikki v:stä saavutettavat solmut on löydetty ja käsitelty, värjätään v mustaksi
- Kaaret joita pitkin läpikäynti on edennyt, eli syvyyssuuntaispuun kaaret on kuvassa paksunnettu
- Syvyyssuuntainen läpikäynti siis löysi solmut seuraavassa järjestyksessä: s, u, v, x, y
- Solmua w ei saavuteta aloitussolmusta, eli lopussa edelleen $color[w]$ = white
- Koko verkolle tehtävä syvyyssuuntainen läpikäynti tapahtuu seuraavasti:

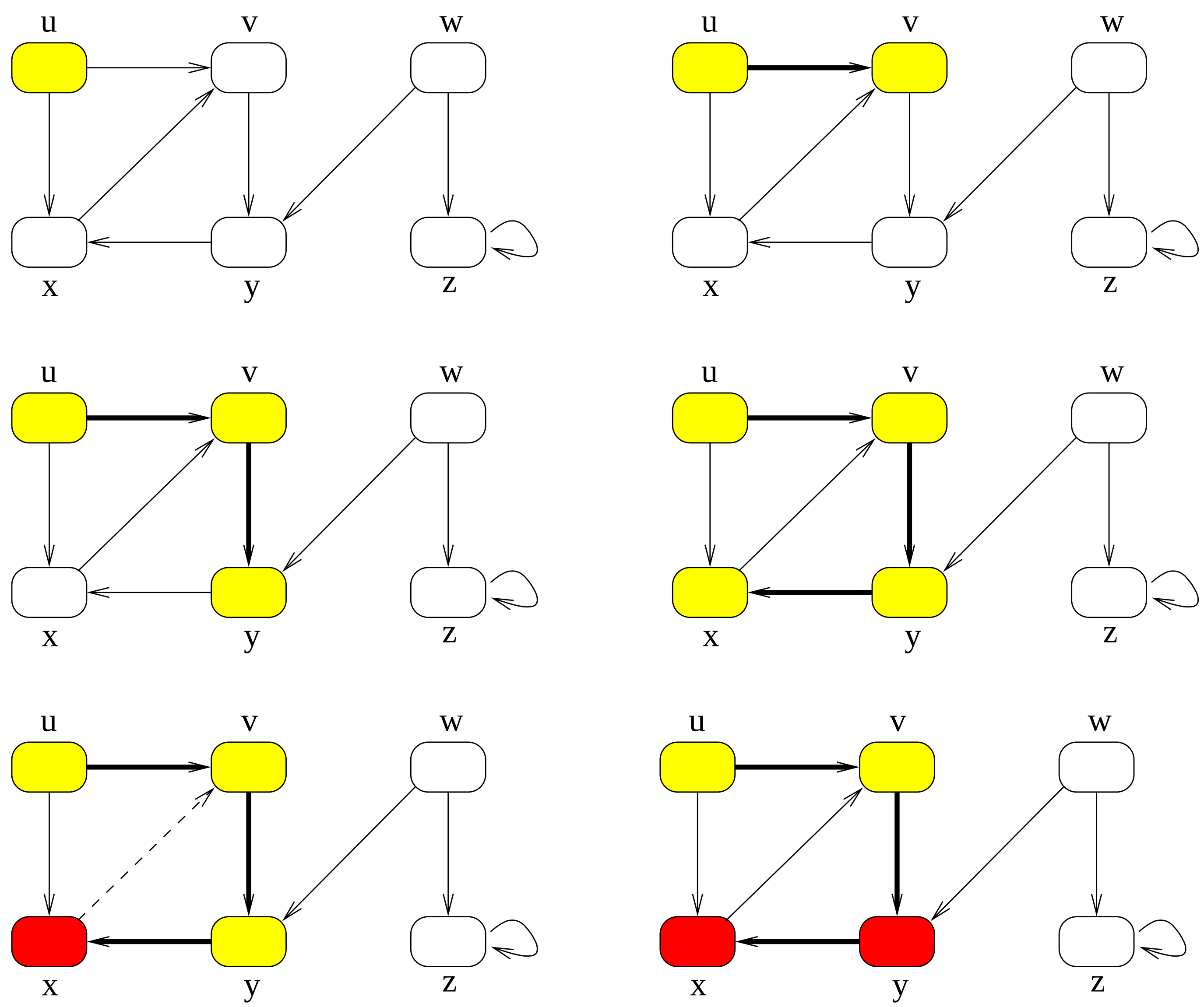
```
DFS-all(G)
for jokaiselle solmulle u ∈ V
    color[u] = white
for jokaiselle solmulle u ∈ V
    if color[u]==white
        DFS-visit(G,u)
```

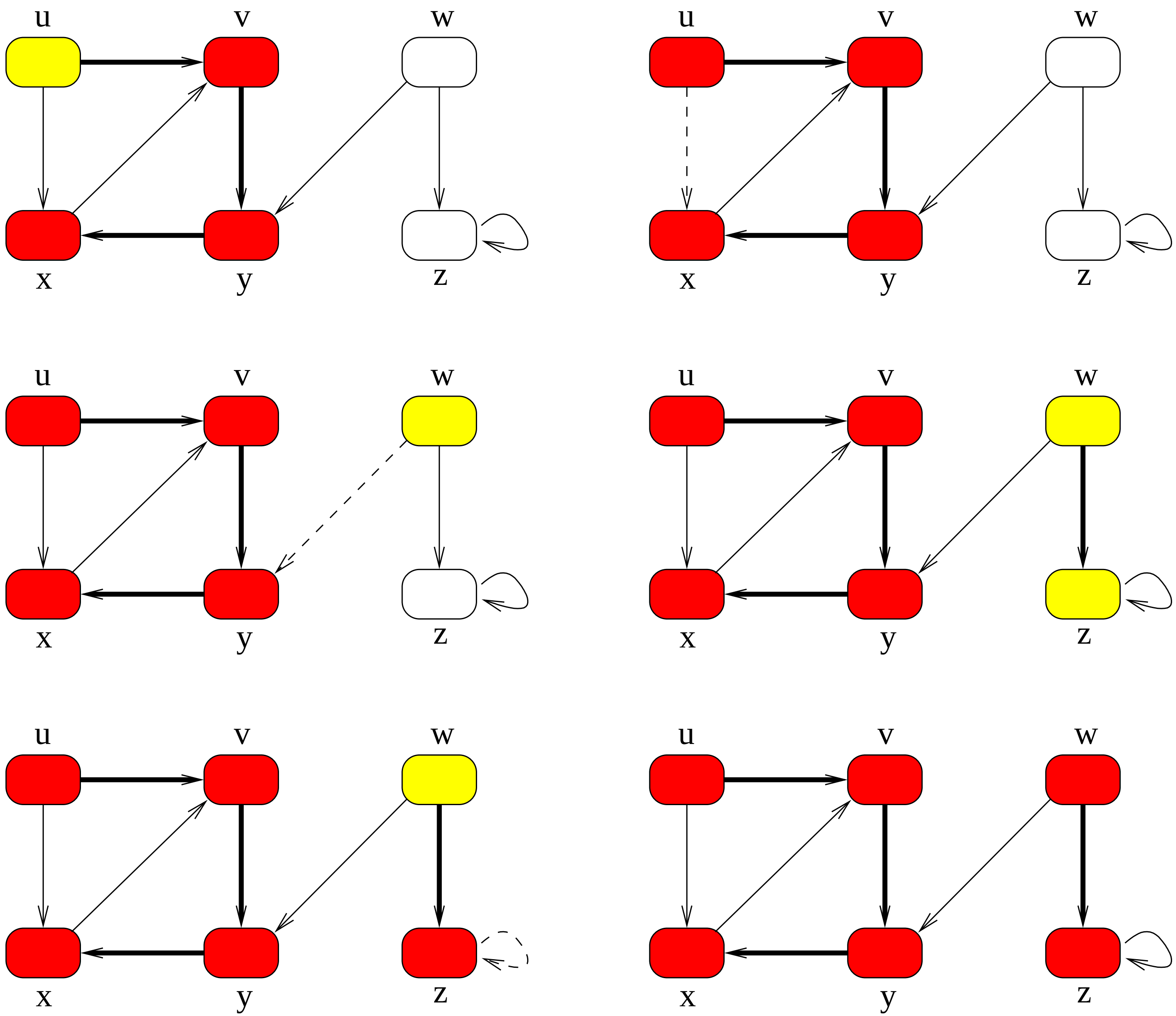
- Koko verkolle tehtävän syvyyssuuntaisen läpikäynnin aikana muodostuu verkon syvyyssuuntainen metsä (engl. depth-first forest). Tämä on kokoelma syvyyssuuntaispuita (engl. depth-first tree). Kukin puu joka koostuu niistä kaarista, joita pitkin läpikäynti eteni aiemmin löytymättömiin solmuihin

- Syvyyssuuntainen metsä ei ole yksikäsitteinen, vaan riippuu valitusta solmujen ja vieruslistojen järjestyksestä

- Kuten leveyssuuntaisen läpikäynnin yhteydessä, syvyyssuuntaispuun kaaret olisi tarvittaessa helppo kirjata algoritmin yhteydessä esim. erilliseen taulukkoon $tree$, johon asetettaisiin $tree[v] = u$ jos läpikäynti eteni solmusta u solmuun v

- Seuraavassa esimerkki algoritmin toiminnasta:



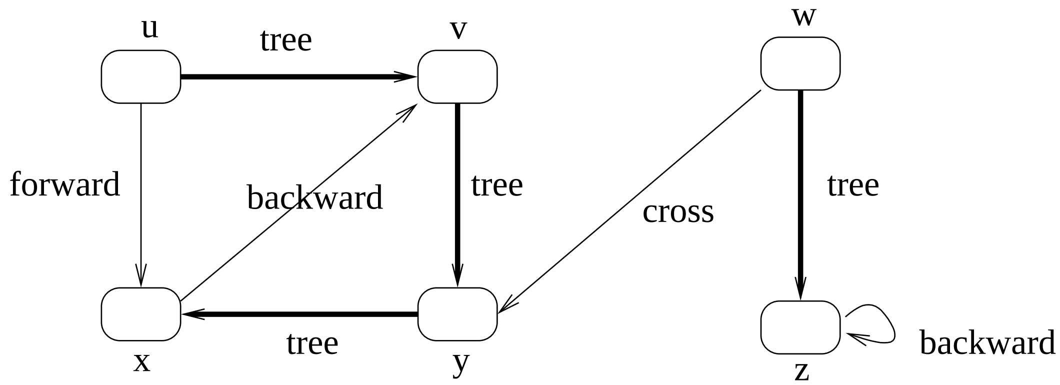

u v w
x y z
u v w
x y z
u v w
x y z
u v w
x y z
u v w
x y z
u v w
x y z

- Syvyyssuuntaisen läpikäynnin vaativuus
  - taulukon $color$ alustus vie aikaa $\mathcal{O}(|V|)$
  - operaatio **DFS-visit** kutsutaan (korkeintaan) kerran jokaiselle solmulle, sillä operaatiota kutsutaan ainoastaan solmuille v, joilla $color[v] =$ white, ja heti kutsun jälkeen asetetaan $color[v] =$ gray eikä väri enää muutu missään vaiheessa valkoiseksi
  - yhteensä **DFS-visit**-operaation kutsuja siis enintään $|V|$ kappaletta
  - **DFS-visit**:in for-lause käy jokaisen solmun vieruslistan läpi, eli for-osa toistetaan yhteensä $|E|$ kertaa
  - kokonaisuudessaan aikaa siis kuluu $\mathcal{O}(|V| + |V| + |E|)$ eli $\mathcal{O}(|V| + |E|)$
  - tilavaativuus algoritmilla on $\mathcal{O}(|V|)$ sillä pahimmassa tapauksessa aloitussolmusta pääsee yhtä polkua pitkin kaikkiin muihin solmuihin ja tällöin sisäkkäisiä rekursiivisia **DFS-visit**-kutsuja tehdään $|V|$ kappaletta; myös aputaulukko vie tilaa $\mathcal{O}(|V|)$
- Aivan kuten leveyssuuntaisen läpikäynnin tapauksessa myös syvyyssuuntaisen läpikäynnin algoritmi toimii sellaisenaan niin suunnatuilla kuin suuntaamattomillakin verkoilla

Kaarten luokittelu

- Verkon kaaret voidaan luokitella neljään eri luokkaan sen perusteella, miten kaaret käyttäytyvät syvyyssuuntaisen läpikäynnin suhteen
- Läpikäynti etenee puunkaaria (engl. tree arc) pitkin uusiin vielä löytymättömiin solmuihin, eli puunkaari kohdistuu valkoiseen solmuun
- Takautuva kaari (engl. backward arc) kohdistuu taaksepäin syvyyssuuntaispuussa, eli takautuva kaari ilmenee kun algoritmi yrittää edetä solmuun joka on jo löydetty, mutta jonka käsittely on kesken, eli takautuva kaari kohdistuu harmaaseen solmuun
- Etenevä kaari (engl. forward arc) kohdistuu eteenpäin syvyyssuuntaispuussa, eli etenevä kaari ilmenee kun algoritmi yrittää edetä solmuun, joka on nykyisen solmun jälkeläinen mutta löydetty ja käsitelty (eli musta) jo jotain nykyisen solmun muuta jälkeläistä tutkittaessa
- Poikittaiskaari (engl. cross arc) kulkee jo löydettyyn eli mustaan solmuun joko
  - kahden eri syvyyssuuntaispuun välillä, tai
  - syvyyssuuntaispuun sisällä sellaisten solmujen välillä joista kumpikaan ei ole toisensa jälkeläinen puussa

- Alla edellisen esimerkin verkon kaaret luokiteltuina



- Läpikäynti etenee aluksi valkoisia solmuja pitkin reittiä $u \to v \to y \to x$, kaikki nämä harmaasta valkoiseen solmuun kohdistuvat ovat puunkaaria

- Kun ollaan solmussa x, ainoa kaari kohdistuu harmaaseen solmuun v, joka siis on jo löydetty mutta jonka vierussolmuja ei ole käsitelty loppuun, $x \to v$ on siis takautuva kaari

- Kun läpikäynti on peruuttanut takaisin solmuun u, tutkitaan kaari $u \to x$ joka kohdistuu u:n seuraajaan syvyysuuntaispuussa, $u \to x$ siis on etenevä kaari

- Kaari $w \to y$ on kahden eri syvyyssuuntaispuussa sijaitsevan solmun välinen, eli kyseessä on poikittaiskaari

- Huom: Kaarten luokittelu on riippuvainen, mistä solmusta läpikäynti aloitetaan

- Suuntaamattomassa verkossa kaaret jaetaan puukaariin ja takautuviin kaariin: takautuvia ja eteneviä ei voi erottaa, ja poikittaisia ei voi esiintyä

Valkopolkulause

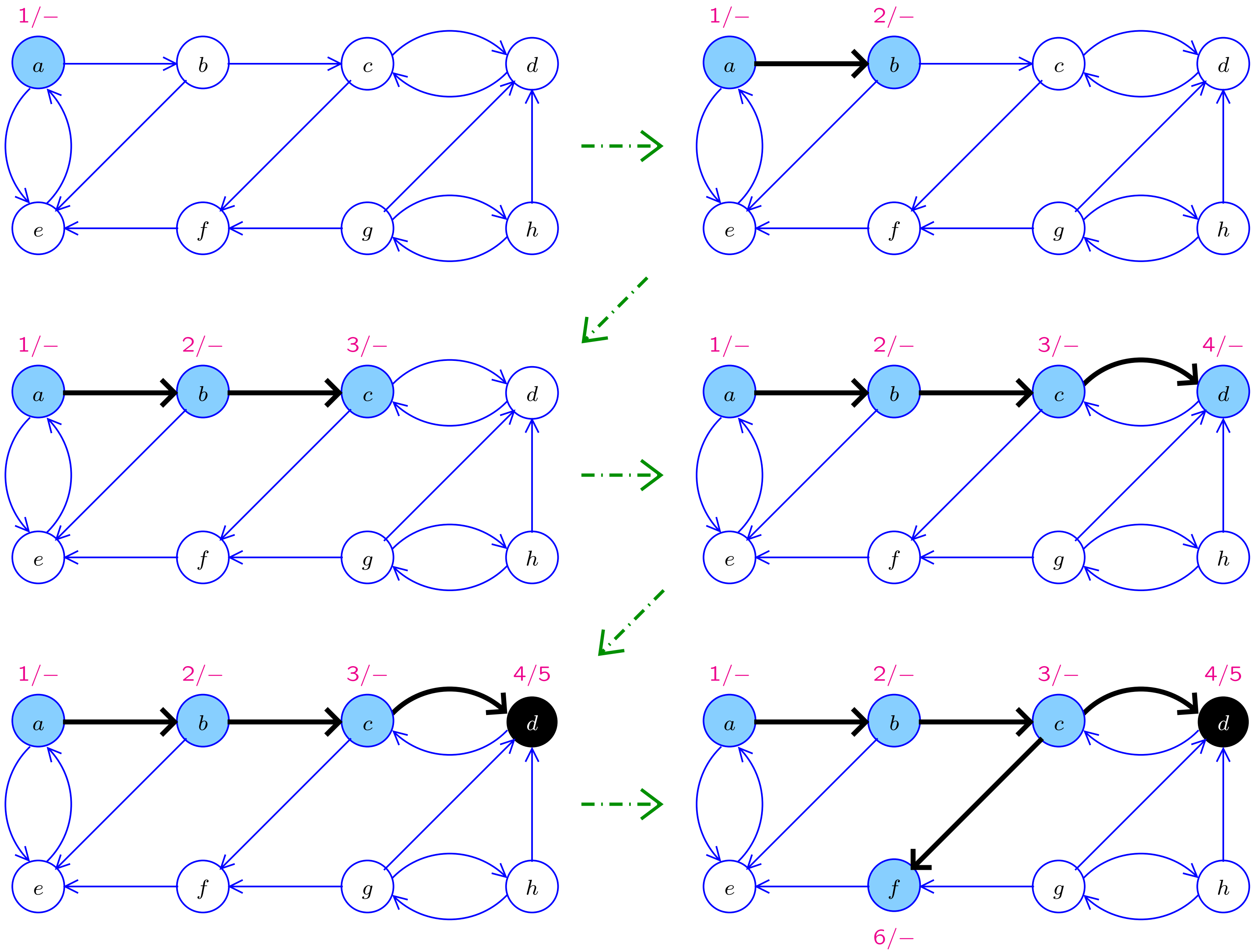
- Algoritmin toiminnan tarkempaa tarkastelua varten, käytämme siitä versiota, johon on lisätty kaksi apumuuttujaa
  - löytymishetki $d[u]$ (discovery): milloin solmu muuttui harmaaksi
  - päättymishetki $f[u]$ (finish): milloin solmu muuttui mustaksi

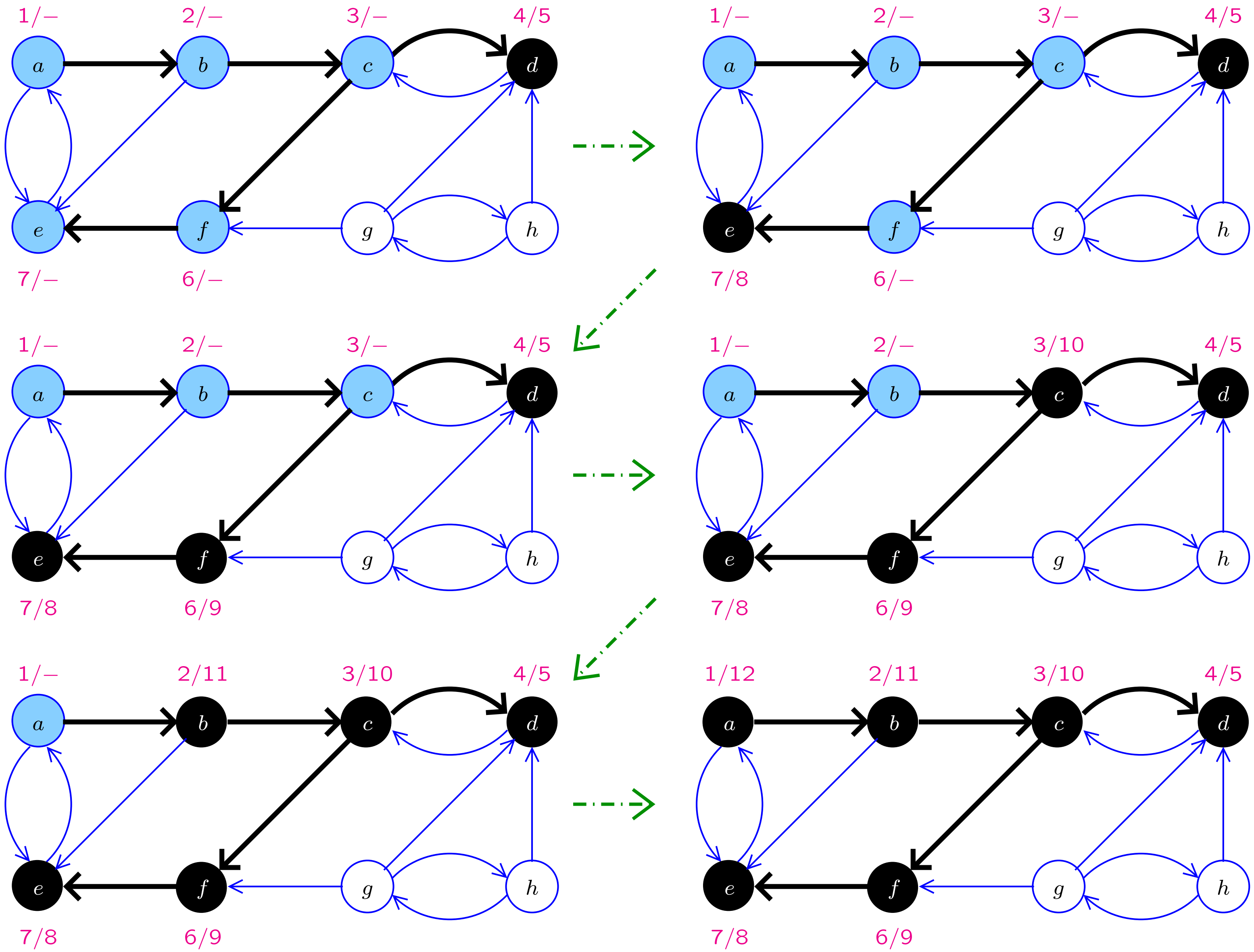
- Aikaa mitataan kaikkiaan tehtyjen värinmuutosten määrällä

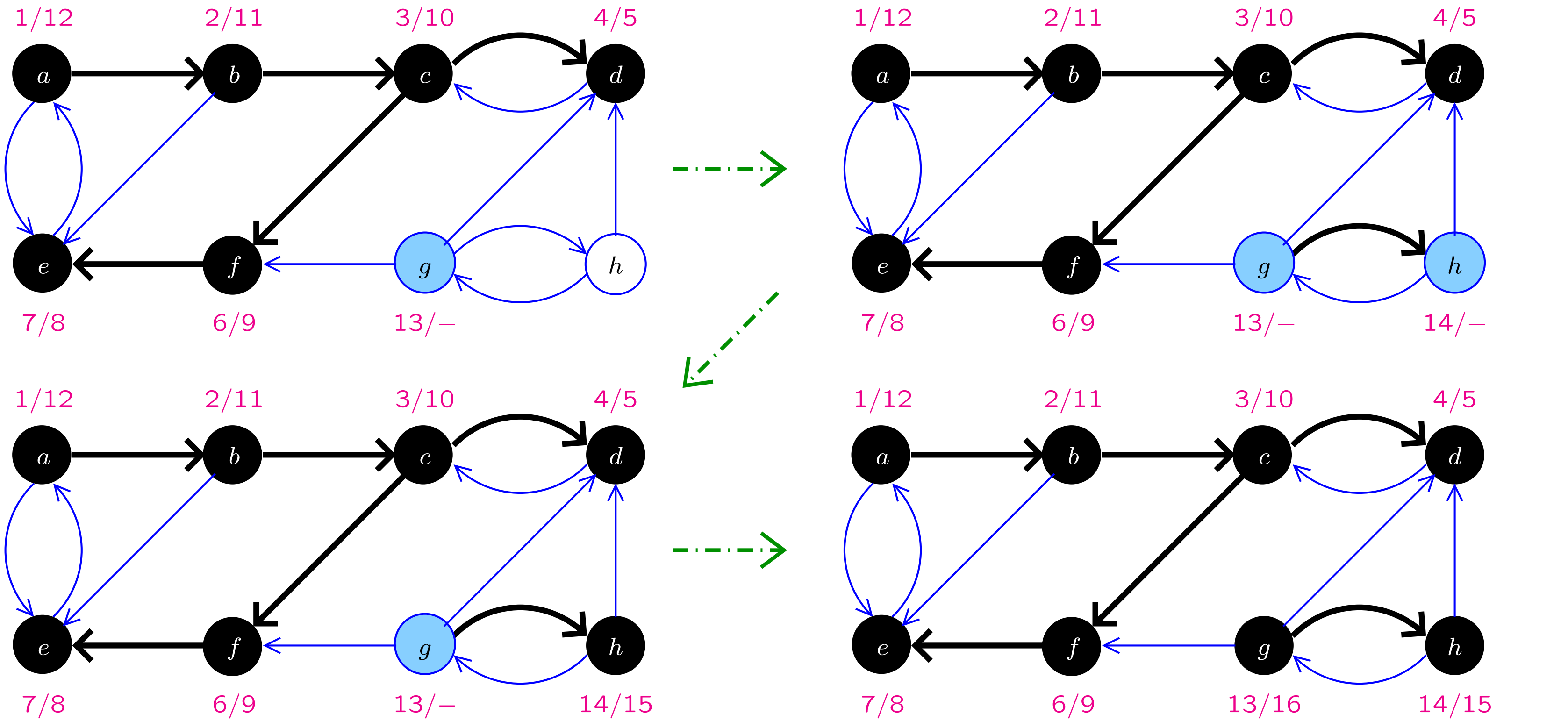
- Algoritmi on nyt seuraavanlainen:

```
DFS-all2(G)
for jokaiselle solmulle u ∈ V
    color[u] = white
time = 0
for jokaiselle solmulle u ∈ V
    if color[u]==white
        DFS-visit2(G,u)

DFS-visit2(G,u)
color[u] = gray
time = time + 1
d[u] = time
for jokaiselle solmulle v ∈ vierus[u]     // kaikille u:n vierussolmuille v
    if color[v]==white                    // solmua v ei vielä löydetty
            DFS-visit2(G,v)
color[u] = black
time = time +1
f[u] = time
```

- Esimerkki suunnatusta tapauksesta seuraavilla kolmella sivulla: solmujen d- ja f-arvot on merkitty muodossa $d[u]/f[u]$; edetyt kaaret (eli puunkaaret) on vahvennettu


1/–
a b c d e f g h
1/– 2/–
a b c d e f g h
1/– 2/– 3/–
a b c d e f g h
1/– 2/– 3/– 4/–
a b c d e f g h
1/– 2/– 3/– 4/5
a b c d e f g h
1/– 2/– 3/– 4/5 6/–
a b c d e f g h


1/– 2/– 3/– 4/5
a b c d
e f g h
7/– 6/–
1/– 2/– 3/– 4/5
a b c d
e f g h
7/8 6/–
1/– 2/– 3/– 4/5
a b c d
e f g h
7/8 6/9
1/– 2/– 3/10 4/5
a b c d
e f g h
7/8 6/9
1/– 2/11 3/10 4/5
a b c d
e f g h
7/8 6/9
1/12 2/11 3/10 4/5
a b c d
e f g h
7/8 6/9



1/12 2/11 3/10 4/5
a b c d
e f g h
7/8 6/9 13/–
1/12 2/11 3/10 4/5
a b c d
e f g h
7/8 6/9 13/– 14/–
1/12 2/11 3/10 4/5
a b c d
e f g h
7/8 6/9 13/– 14/15
1/12 2/11 3/10 4/5
a b c d
e f g h
7/8 6/9 13/16 14/15

- **DFS-visit2**:ta tarkastelemalla näemme suoraan seuraavan "sulkumerkkiteoreeman"

- **Lause 8.1**: Mille tahansa syvyysuuntaispuun solmuille u ja v pätee jokin seuraavista:

  1. $d[u] < d[v] < f[v] < f[u]$, ja solmu v on solmun u jälkeläinen

  2. $d[v] < d[u] < f[u] < f[v]$, ja solmu u on solmun v jälkeläinen

  3. $d[u] < f[u] < d[v] < f[v]$ tai $d[v] < f[v] < d[u] < f[u]$, ja solmuista kumpikaan ei ole toisen jälkeläinen.

$\square$

- Lause sanoo, että kutsujen alku- ja loppumisajat vastaavat alku- ja loppusulkuja hyvinmuodostetussa lausekkeessa

- Seuraava valkopolkulause (White-path Theorem) pätee suunnatuissa ja suuntaamattomissa verkoissa

- Se tulee käyttöön myöhemmin

- **Lause 8.2**: Solmu v tulee solmun u jälkeläiseksi syvyyssuuntaisessa metsässä, jos ja vain jos kutsun **DFS-visit**(G, u) alkaessa on olemassa pelkistä valkoisista solmuista koostuva polku $u \rightsquigarrow v$.

- **Todistus:**

  $\Rightarrow$: Kun v on u:n jälkeläinen, olkoon $(u, w_1, \ldots, w_k, v)$ puukaaria pitkin kulkeva polku $u \rightsquigarrow v$. Sulkumerkkiteoreeman mukaan $d[u] < d[w_1] < \ldots < d[w_k] < d[v]$, joten kutsun **DFS-visit**(G, u) alkaessa kaikki polun solmut ovat valkoisia

$\Leftarrow$: Olkoon $(u, w_1, \ldots, w_k, v)$ valkoisista solmuista koostuva polku, kun **DFS-visit**(G, u) alkaa. Tehdään vastaoletus, että ainakin yksi polun solmuista ei tule u:n jälkeläiseksi. Olkoon v näistä ensimmäinen. Olkoon lisäksi w solmua v edeltävä solmu polulla. Nyt siis w on joko u tai u:n aito jälkeläinen. Sulkumerkkiteoreeman mukaan, kun w on u:n jälkeläinen (ottamalla huomioon erikoistapaus $w = u$), pätee

$$d[u] \le d[w] < f[w] \le f[u].$$

Koska v:hen tullaan u:n jälkeen, mutta ennen kuin w on käsitelty loppuun, niin $d[u] < d[v] < f[w]$. Mutta tästä siis seuraa, että $d[u] < d[v] < f[u]$, eli sulkumerkkiteoreeman mukaan vain tapaus

$$d[u] < d[v] < f[v] < f[u]$$

on mahdollinen. Mutta tämä tarkoittaa, että v on sittenkin u:n jälkeläinen; ristiriita. $\square$

- Jos et ymmärtänyt tätä valkopolku-osuutta, niin suurta vahinkoa ei ole tapahtunut...

Verkon syklittömyyden tarkastus

- Joskus voi olla tarvetta testata onko annetussa suunnatussa verkossa sykliä, eli onko kyseessä syklitön suunnattu verkko

- Pienellä muutoksella voimme käyttää syvyyssuuntaisen läpikäynnin algoritmia syklittömyyden testaamiseen:
  - oletetaan että ollaan tutkimassa solmun u vieruslistaa $vierus[u]$
  - jos vieruslistalta löytyy solmu v jolle $color[v] =$ gray, tiedämme että v:n käsittely on kesken, ja
  - solmusta v johtaa polku solmuun u
  - nyt siis on olemassa polku $v \leadsto u \to v$, eli verkossa on sykli
  - syklin olemassaolo siis havaitaan jos syvyyssuuntaispuussa on takautuva kaari

- Tarvittava muutos on siis testi löytyykö tutkittavan solmun vieruslistalta solmu v, jolle $color[v] =$ gray

- Seuraavalla sivulla algoritmi palauttaa true jos verkossa on sykli ja muuten false

DFS-cycles(G)

```
for jokaiselle solmulle u ∈ V
    color[u] = white
for jokaiselle solmulle u ∈ V
    if color[u]==white
        if DFS-search-cycles(G,u) == true
            return true
return false
```

DFS-search-cycles(G,u)

```
color[u] = gray
for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
    if color[v] == gray                  // löytyi sykli, voidaan lopettaa
        return true
    if color[v]==white                   // solmua v ei vielä löydetty
        if DFS-search-cycles(G,v) == true
            return true
color[u] = black
return false
```

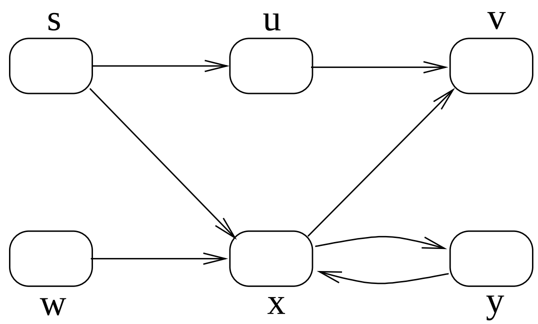
- Jos siis tutkittaessa solmun u vieruslistaa $vierus[u]$ löytyy solmu v jolle $color[v] =$ gray, tiedämme että v:n käsittely on kesken, ja solmusta v johtaa polku solmuun u eli on olemassa polku $v \leadsto u \to v$, eli verkossa on sykli

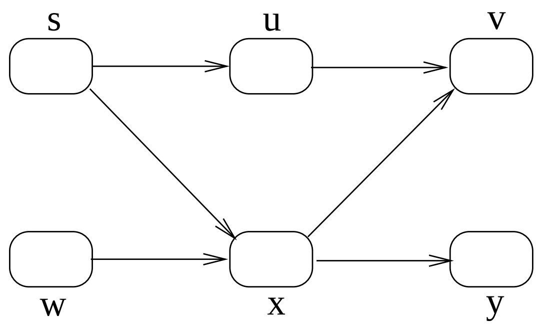
- Perustellaan seuraavassa vielä asia toisinpäin eli, jos verkossa on sykli, algoritmi huomaa syklin olemassaolon
  - Oletetaan, että verkossa on sykli ja olkoon v läpikäynnissä ensimmäisenä vastaantuleva syklin solmu
  - Koska v on osa sykliä, on olemassa solmu u jolle $v \leadsto u \to v$, eli u on syklissä oleva solmu josta on kaari v:hen
  - Kun läpikäynti tulee solmuun v, ei oletuksen mukaan u:ta eikä muita syklin solmuja ole vielä löydetty ja koska $v \leadsto u$, niin läpikäynti etenee solmuun u
  - Kun u:n vierussolmuja tutkitaan, havaitaan, että u:sta on kaari harmaaseen solmuun v, eli algoritmi löytää syklin

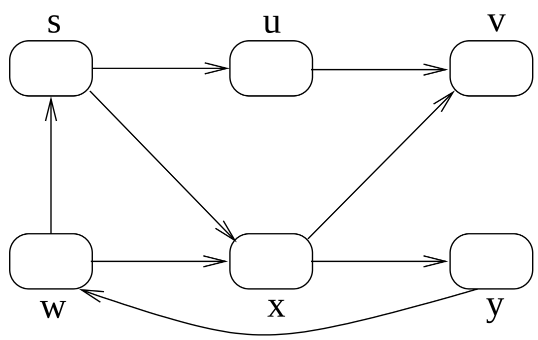
- Algoritmi siis löytää syklin jos ja vain jos verkossa on sykli, eli algoritmi toimii oikein

- Aika- ja tilavaativuus algoritmilla on tietenkin sama kuin modifioimattomalla syvyyssuuntaisella läpikäynnillä

- Esim: miten syklien etsintä toimisi seuraavissa verkoissa?



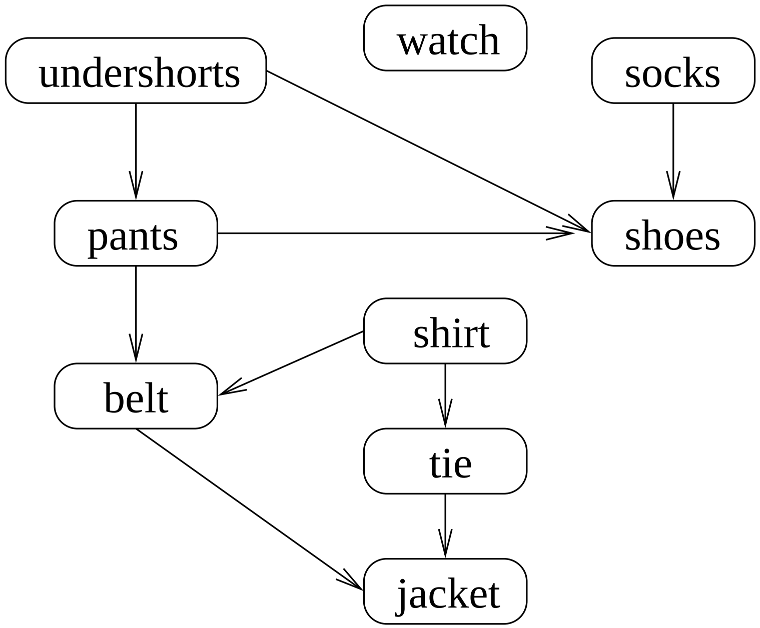




Topologinen järjestäminen

- Syklittömien suunnattujen verkkojen (engl. Directed Acyclic Graph, DAG) avulla voimme kuvata tapahtumien välisiä riippuvuuksia

- Alla oleva verkko sisältää pukeutumiseen kannalta oleelliset riippuvuudet:



- Eli esim. sukat on laitettava ennen kenkiä koska on kaari socks $\rightarrow$ shoes

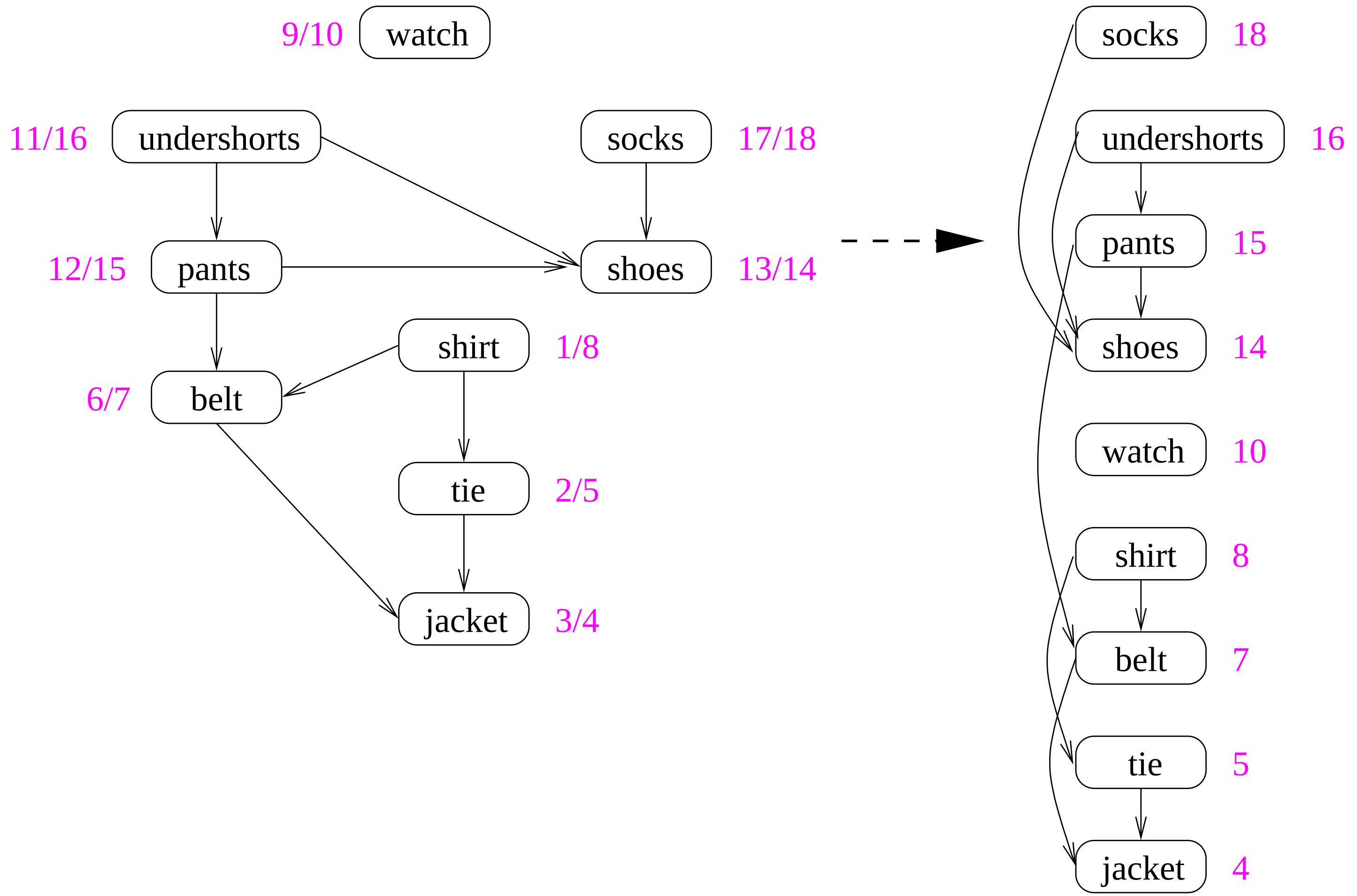
- Kello voidaan laittaa käteen missä vaiheessa tahansa koska sillä ei ole mitään riippuvuutta

- Herää kysymys voisimmeko järjestää asiat sellaiseen lineaariseen järjestykseen että voisimme pukea vaatekappaleen kerrallaan siten että mikään riippuvuuksista ei rikkoudu, eli
  - jos riippuvuusverkossa on kaari $u \to v$, tulee u järjestyksessä ennen v:tä
- Tällaista järjestystä kutsutaan topologiseksi järjestykseksi
- Asia hoituu helposti käyttäen apuna syvyyssuuntaista läpikäyntiä

Topological-Sort(G)
    kutsutaan DFS-all(G)
        kun solmun v käsittely on ohi eli solmu muuttuu mustaksi
        lisätään se pinoon P
    return P

- Operaation jälkeen solmut ovat pinossa P järjestettynä siten että pinon päällä on viimeiseksi käsitelty solmu, eli solmu v_n, johon ei varmuudella kohdistu yhtään kaarta eli jolla ei ole yhtään riippuvuutta
- Pinossa toisena on solmu v_{n-1} johon on olemassa kaari eli riippuvuus korkeintaan solmusta v_n, jne.
- Solmut ovat siis pinossa P topologisessa järjestyksessä

- Seuraavassa esimerkkimme topologisesti järjestettynä. Solmujen yhteyteen on merkitty solmujen d- ja f-arvot on merkitty muodossa $d[u]/f[u]$; solmujen järjestys pinossa siis on käänteinen f-numerojärjestys ja f-arvot on merkitty kuvaan



- Perustellaan algoritmin oikeellisuus vielä hieman tarkemmin

- Algoritmi toimii oikein jos kaikille solmuille v, w missä v topologisessa järjestyksessä solmun w edellä, ei ole olemassa kaarta $w \to v$

- Oletetaan, että kaari $w \to v$ on olemassa, ja näytetään, että tästä seuraa ristiriita

  koska v on solmua w edellä topologisessa järjestyksessä, on sen käsittely päättynyt myöhemmin kuin w:n käsittely

  - jos w:n käsittely olisi aloitettu v:n löytymisen jälkeen, olisi verkossa polku $v \rightsquigarrow w$ sekä kaari $w \to v$ eli verkossa olisi sykli. Tämä on ristiriita, sillä oletus on, että verkko on syklitön

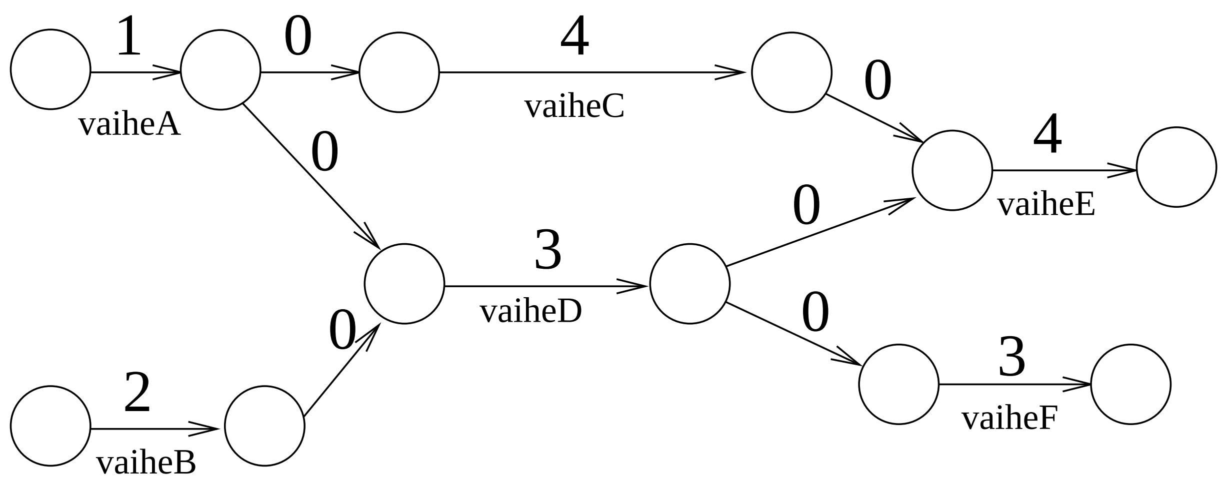
  - jos w:n käsittely olisi aloitettu ennen v:n löytymistä, olisi läpikäynti kaaren $w \to v$ seurauksena edennyt solmuun v ja v:n käsittelyn olisi täytynyt päättyä ennen w:n käsittelyn päättymistä. Tämä taas on ristiriita solmujen topologisen järjestyksen takia

- On siis osoitettu, että kaaren $w \to v$ olemassaolo johtaisi ristiriitaan, eli topologisen järjestyksen rikkovaa kaarta ei voi olla olemassa, eli topologinen järjestäminen toimii oikein

Topologisen järjestämisen sovellus: kriittiset työvaiheet

- Ohjelmistoprojektissa on tiettyjä määräaikoja eli deadlineja
  - dokumenttien ja komponenttien deadlinet, katselmuksia, asiakasdemoja
- Deadlineilla on osittainen järjestys:
  - käyttöliittymä on saatava valmiiksi ennen asiakasdemoa
  - käyttöliittymää ei voida aloittaa ennen kuin tietokantaliittymä on valmis
  - tietokantaliittymää voi testata käyttöliittymästä riippumatta
- Eri vaiheilla on kestoja
  - käyttöliittymän aloittamisesta sen valmistumiseen kuluu (arviolta) 4 viikkoa
- Ero topologiseen järjestämiseen:
  - lisätään eri vaiheille kestot ja sallitaan eri vaiheiden tekeminen rinnakkain
- Mallissa tehdään yksinkertaistuksia, eikä oteta huomioon esim. että:
  - joitakin vaiheita ei voida tehdä yhtä aikaa, koska käytettävissä on vain yksi henkilö, joka pystyy tekemään niitä
  - joitakin vaiheita ei voida tehdä yhtä aikaa, mutta niiden keskinäisellä järjestyksellä ei ole merkitystä

- Ongelma: halutaan selvittää, kauanko projektiin vähintään kuluu aikaa
- Ongelma mallinnetaan suunnattuna verkkona:
  - Jokaisesta vaiheesta tehdään kaari, jonka päätepisteinä ovat vaiheen alku ja loppu ja painona vaiheen kesto
  - Jos kahdella vaiheella määrätty järjestys, tehdään kaari, jonka lähtösolmuna on ensimmäisen vaiheen loppu ja maalisolmuna jälkimmäisen vaiheen alku
  - Lasketaan tämän verkon pisin polku (siis painavin polku)
- Esim.
  - vaiheA kestää kuukauden, vaiheB 2 kuukautta, vaiheC 4 kuukautta, . . .
  - vaiheA:lla ja vaiheB:llä ei riippuvuutta, vaiheA tehtävä ennen vaiheC:tä, . . .



- Talletetaan kustakin solmusta lähtevän painavimman polun paino taulukkoon $heaviest$

- Solmusta u alkavan polun suurin mahdollinen paino $heaviest[u]$ voidaan määritellä seuraavasti:
  - $heaviest[u] = 0$, jos solmusta ei ole kaarta eteenpäin
  - muuten paino on

$$\begin{aligned} heaviest[u] \quad = \quad & \text{max}\{w(u,v) + heaviest[v] \mid \text{solmusta } u \text{ on solmuun } v \text{ kaari,} \\ & \text{jonka paino on } w(u,v)\} \end{aligned}$$

- Laskusääntöä ei voi käyttää, jos verkossa on sykli, mutta tällöin projekti on myös mahdoton

- Solmun u luku $heaviest[u]$ voidaan laskea vasta sitten, kun on laskettu sen kaikkien seuraajasolmujen v luvut $heaviest[v]$

- Solmut on siis käsiteltävä käänteisessä topologisessa järjestyksessä

- Solmun u luku $heaviest[u]$ voidaan laskea sillä hetkellä, kun u viedään topologisen järjestyksen laskevassa algoritmissa pinoon P, eli kun solmu värjätään mustaksi

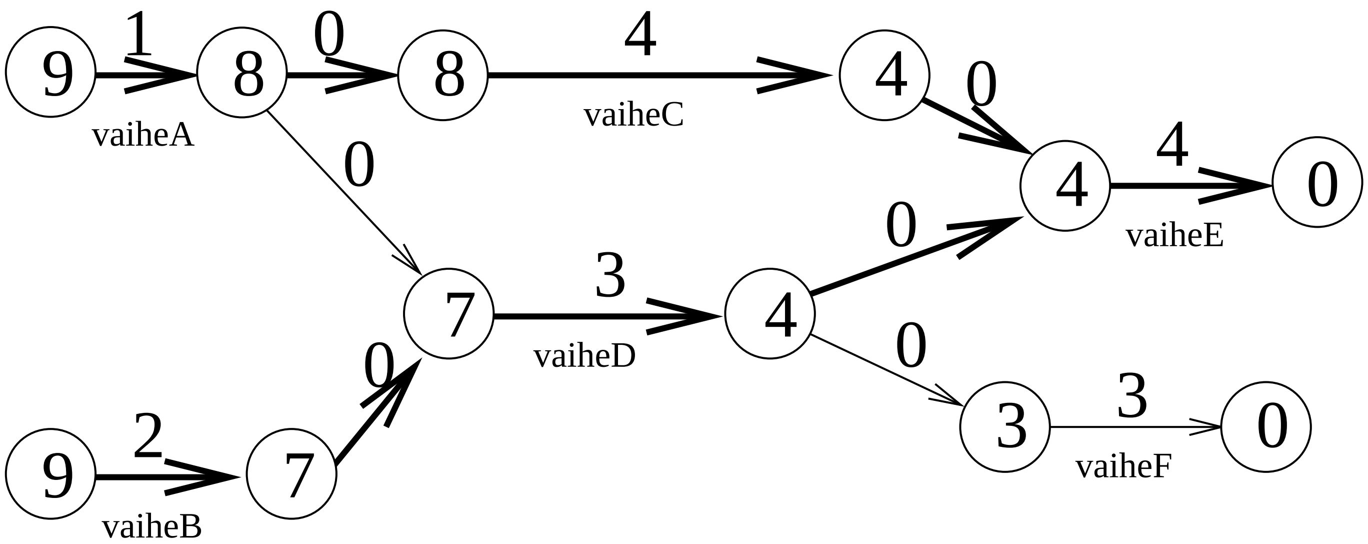
- Projektin kokonaiskesto on

$$t = \max_{u \in V} heaviest[u]$$

- Käytännössä halutaan tietää myös, mitkä projektin poluista ovat kriittisiä: sellaisia, joiden myöhästyminen venyttää koko projektin kestoa
  - Kriittisiä ovat ainakin ne vaiheiden alkusolmut u, joiden $heaviest[u] = t$
  - Jos solmu u on kriittinen, niin sen seuraajasolmuista v ovat kriittisiä ne, joille $heaviest[u] = w(u, v) +$ $heaviest[v]$

- Kriittiset polut voidaan tulostaa syvyyssuuntaisella läpikäynnillä sen jälkeen, kun kunkin solmun $heaviest$-arvo on ensin laskettu
  - lähtösolmuina ovat ne solmut u, joille ehto $heaviest[u] = t$ pätee
  - kaari (u, v) kuuluu kriittiselle polulle, jos $heaviest[u] = w(u, v) +$ $heaviest[v]$

- Esimerkki kriittisten polkujen löytämisestä
  - jokaiseen solmuun on merkitty sen *heaviest*-arvo
  - kriittiset polut on vahvennettu



Pisin yksinkertainen polku painotetussa verkossa

- Pisin tarkoittaa tässä siis painavin
- Ongelma ei ole mielekäs ilman polun yksinkertaisuusoletusta, jos verkossa on positiivinen sykli
- Yleisessä tapauksessa ongelma on NP-täydellinen, josta seuraa, että ongelmalle ei tunneta polynomista ratkaisualgoritmia
- Edellä kuitenkin näimme, että kun kyseessä on DAG, tämä vastaa kriittisten polkujen etsintää
- Kirjoitamme tässä vielä algoritmin auki DAGille

```
max-path(G)
    for jokaiselle solmulle u ∈ V
        color[u] = white
    for jokaiselle solmulle u ∈ V
        if color[u] == white
            DFS-visit3(u)
    return max{ h[u] | u ∈ V }
```

```
DFS-visit3(u)
    color[u] = gray
    h[u] = 0
    for jokaiselle solmulle v ∈ vierus[u]
        if color[v] == white
            DFS-visit3(v)
        if h[u] < w(u,v) + h[v]
            h[u] = w(u,v) + h[v]
```

Verkon vahvasti yhtenäiset komponentit

- Suunnattua verkkoa $G = (V, E)$ sanotaan vahvasti yhtenäiseksi (engl. strongly connected), jos kaikilla solmuilla $u, v \in V$ on olemassa polut $u \leadsto v$ ja $v \leadsto u$

- Eli vahvasti yhtenäisen verkon kaikki solmut ovat saavutettavia toisistaan

- Jos verkko ei ole vahvasti yhtenäinen, se voidaan jakaa erillisiin vahvasti yhtenäisiin komponentteihin (engl. strongly connected components) $V_1, V_n, \ldots, V_n$ seuraavasti:

  1. jokaisella solmulla u pätee $u \in V_i$ tasan yhdellä i

     eli kukin solmu kuuluu tasan yhteen vahvasti yhtenäiseen komponenttiin

  2. jos $u \in V_i$ ja $v \in V_i$, niin $u \leadsto v$ ja $v \leadsto u$

     eli vahvasti yhtenäisen komponentin kaikki solmut ovat toisistaan saavutettavissa

  3. jos $u \in V_i$ ja $v \in V_j$, missä $i \neq j$, niin ei ole olemassa molempia poluista $u \leadsto v$ ja $v \leadsto u$

     eli kahden eri vahvasti yhtenäisen komponentin solmut eivät ole molemmat toisistaan saavutettavia

- Matemaattisemmin ilmaistuna voidaan sama sanoa käyttäen käsitteitä ekvivalenssirelaatio ja ekvivalenssiluokka

- Voidaan merkitä $u \sim v$, jos verkossa G on sekä polku $u \rightsquigarrow v$ että polku $v \rightsquigarrow u$; $\sim$ on nyt ekvivalenssirelaatio eli toteuttaa ehdot

  1. $u \sim u$ kaikilla u (refleksiivisyys)

  2. jos $u \sim v$ niin $v \sim u$ (symmetrisyys)

  3. jos $u \sim v$ ja $v \sim w$ niin $u \sim w$ (transitiivisuus)

- Tästä seuraa, että solmut voidaan jakaa sen suhteen ekvivalenssiluokkiin V_i, jolloin

  – jokaisella u pätee $u \in V_i$ tasan yhdellä i

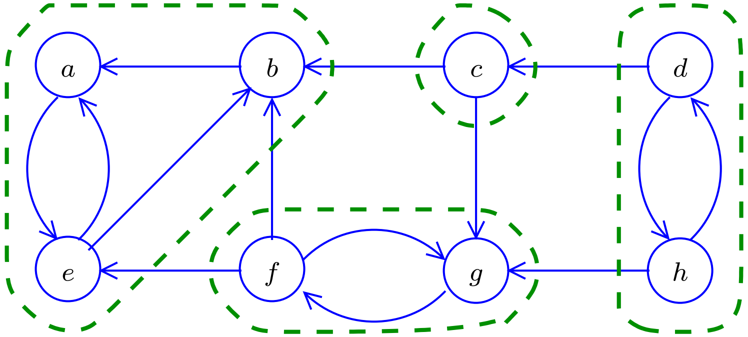
  – jos $u \in V_i$ ja $v \in V_i$, niin $u \sim v$

  – jos $u \in V_i$ ja $v \in V_j$, missä $i \neq j$, niin $u \not\sim v$

- Näitä ekvivalenssiluokkia V_i sanotaan siis verkon vahvasti yhtenäisiksi komponenteiksi

- Verkko on siis vahvasti yhtenäinen, jos sillä on vain yksi vahvasti yhtenäinen komponentti

- Tarkastellaan seuraavaa suunnattua verkkoa:



- Verkon vahvasti yhtenäiset komponentit ovat

$$
\begin{aligned}
V_1 &= \{a, b, e\} \\
V_2 &= \{c\} \\
V_3 &= \{d, h\} \\
V_4 &= \{f, g\}
\end{aligned}
$$

- Verkko, jonka solmut ovat vahvasti yhtenäiset komponentit ja kaaret yhteydet komponettien välillä kutsutaan komponenttiverkoksi

- Vaikka verkon vahvasti yhtenäisten komponenttien tuntemisen hyödyllisyys ei ole päällepäin kovin ilmeinen seikka, on vahvasti yhtenäisten komponenttien selvittämiselle paljon käyttöä erilaisissa sovellustilanteissa

- Vahvasti yhtenäiset komponentit voidaan muodostaa seuraavalla algoritmilla:

  Strongly-Connected-Components(G)

  1. Suorita verkon $G = (V, E)$ syvyyssuuntainen läpikäynti.

  Kun solmun v käsittely on ohi, eli asetetaan $color[v] =$ black, lisätään se pinoon P (eli pinon päällimmäisenä on solmu, jonka f-arvo on suurin)

  2. Muodosta verkon G transpoosi $G^\top = (V^\top, E^\top)$, missä $V^\top = V$ ja $E^\top = \{ (v, u) \mid (u, v) \in E \}$.

  Verkon transpoosi on siis muuten sama verkko, mutta kaaret on käännetty päinvastaiseen suuntaan

  3. Suorita verkon $G^\top$ syvyyssuuntainen läpikäynti alkaen pinon P huipulla olevasta alkiosta.

  Jokainen verkon $G^\top$ syvyyssuuntaisen läpikäynnin aikana muodostunut syvyyssuuntaispuu on verkon G vahvasti yhtenäinen komponentti.

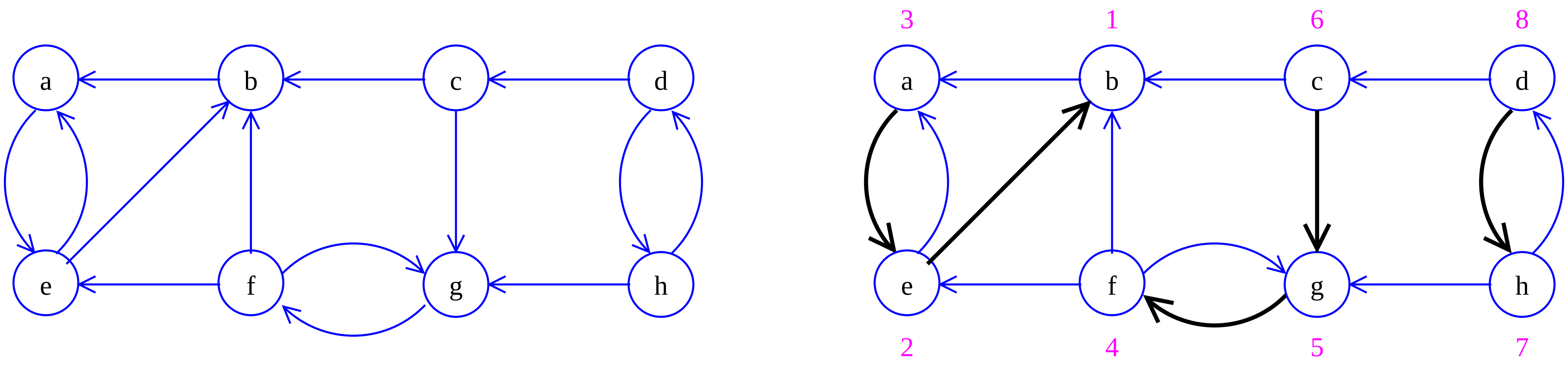
  Aina kun yksi puu on tullut läpikäydyksi, seuraava **DFS-visit**$(G^\top, u)$ alkaa pinosta P ensimmäisenä löytyvästä vielä läpikäymättömästä solmusta u (eli f-arvoltaan suurimmasta vielä läpikäymättömästä solmusta)

- Koska verkon transpoosi voidaan muodostaa ajassa $\mathcal{O}(|V| + |E|)$, vahvasti yhtenäiset komponentit selvittävän algoritmin pahimman tapauksen aikavaativuus on $\mathcal{O}(|V| + |E|)$ ja tilavaativuus on $\mathcal{O}(|V|)$ kuten syvyyssuuntaisella läpikäynnillä

- Tarkastellaan esimerkkiä:

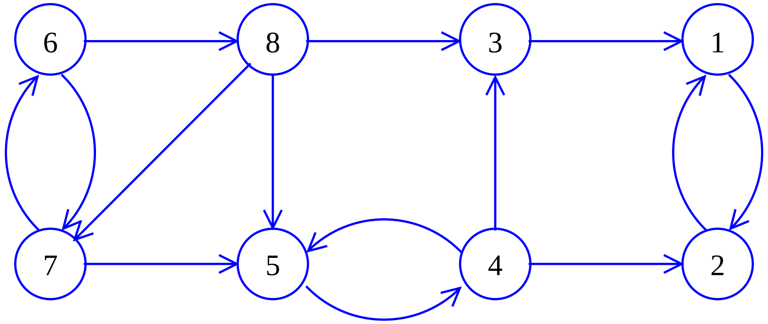
- Vasemmalla verkko G jonka vahvasti yhtenäiset komponentit halutaan selvittää

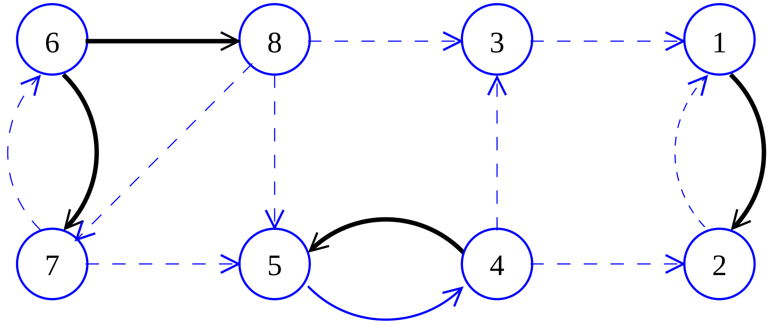
- Oikealla verkon G syvyyssuuntaisen läpikäynnin tulos, solmut on numeroitu siinä järjestyksessä missä niiden käsittely valmistui, eli missä järjestyksessä ne laitettiin pinoon P



- Vasemmalla transpoosiverkko G^{T}, solmut on numeroitu siinä järjestyksessä missä ne otetaan pinosta, joka siis on järjestys, missä transpoosin syvyyssuuntainen läpikäynti etenee

- Oikealla läpikäynnin tulos, verkon G vahvasti yhtenäiset komponentit muodostuvat läpikäynnin aikana muodostuneista syvyyssuuntaispuiden solmuista





- Algoritmi on suhteellisen yksinkertainen, mutta päältäpäin on vaikea nähdä, että algoritmi todella toimii

- Todistetaan seuraavaksi, että algoritmi toimii oikein

- Algoritmin toiminta perustuu seuraavaan havaintoon:

- **Lause 8.3**: Jos $A \neq B$ ovat kaksi vahvasti yhtenäistä komponenttia ja $(u, v) \in E$ joillain $u \in A$ ja $v \in B$, niin

$$\max_{x \in A} f[x] > \max_{y \in B} f[y].$$

  Siis jos komponentista A on kaari komponenttiin B, niin vaiheessa 1 komponentin B läpikäynti loppuu ennen kuin komponentin A läpikäynti.

- **Todistus**: Kaksi tapausta sen mukaan, kumman komponentin läpikäynti alkaa ensin

  1. Oletetaan ensin, että komponentin A läpikäynti alkaa solmusta $x \in A$, kun kaikki komponentin B solmut ovat vielä valkoisia. Koska $x \leadsto u \to v \leadsto y$ millä tahansa $y \in B$ ja kutsun **DFS-visit**(G, x) alkaessa kaikki nämä solmut ovat valkoisia, valkopolkulauseen mukaan jokainen $y \in B$ tulee solmun x jälkeläiseksi. Siis $f[y] < f[x]$ kaikilla $y \in B$

  2. Oletetaan nyt, että komponentin B läpikäynti alkaa solmusta $y \in B$, kun kaikki komponentin A solmut ovat vielä valkoisia. Koska komponenttiverkko on syklitön, solmusta y ei ole polkua mihinkään komponentin A solmuun. Siis kaikista komponentin B solmuista tulee mustia, ennen kuin mikään komponentin A solmu on edes harmaa. $\square$

- **Lause 8.4**: Algoritmi **Strongly-Connected-Components** tuottaa oikein verkon vahvasti yhtenäiset komponentit.

- **Todistus**: Todistamme induktiolla k:n suhteen, että k ensimmäistä vaiheessa 3 tuotettua puuta ovat verkon G vahvasti yhtenäisiä komponentteja.

  Tapaus $k = 0$ on triviaali.

  Oletetaan, että ensimmäiset $k - 1$ puuta ovat oikein ja tarkastellaan puuta numero k.

  Olkoon y puun numero k juuri, ja olkoon B se vahvasti yhtenäinen komponentti, johon y kuuluu.

  Induktio-oletuksen mukaan aiemmat puut eivät sisältäneet komponentin B solmuja, joten kaikki komponentin B solmut ovat kutsun **DFS-visit**(G^{T}, y) alkaessa valkoisia. Valkopolkulauseen mukaan ne tulevat solmun y jälkeläisiksi transpoosin syvyyssuuntaisessa puussa.

Pitää vielä osoittaa, että solmun y jälkeläisiksi ei tule yhtään solmua x, missä x kuuluu johonkin toiseen komponenttiin $A \neq B$.

Ei-valkoisista solmuista ei tietenkään enää tehdä kenenkään jälkeläisiä. Olkoon $A \neq B$ jokin vielä valkoisista solmuista koostuva komponentti.

Koska valitaan **DFS-visit** alkamaan f-arvoltaan suurimmasta vielä läpikäymättömästä solmusta, pätee, että $f[y] > f[x]$ kaikilla $x \in A$. Edellisen lauseen mukaan verkossa G ei voi olla kaarta komponentista A komponenttiin B. Siis transpoosissa G^{T} ei ole kaaria komponentista B komponenttiin A.

Täten solmun y jälkeläisiksi tulee tasan kaikki komponentin B solmut. $\square$

- Todistus siis osoittaa, että algoritmi toimii oikein vaikka algoritmin toiminnan perusteita onkin hiukan vaikea nähdä päältäpäin

- Vahvasti yhtenäiset komponentit etsivä algoritmi onkin hyvä esimerkki tilanteessa, jossa matematiikkaa tarvitaan pelkän intuition lisäksi vakuuttamaan algoritmin oikeellisuudesta

Lyhimmät polut painotetussa verkossa

- Tarkastellaan painotettuja verkkoja ja tulkitaan kaaren paino sen yhdistämien solmujen etäisyydeksi
- Kaaripainon voi tulkita myös esim. ajaksi mikä kuluu matkustettaessa kaaren yhdistävien solmujen välinen matka
- Tehtävänä on löytää annetusta solmusta s lyhin etäisyys, eli vähiten painava polku kaikkiin muihin solmuihin
- Ongelma on keskeinen esim. tietoliikenneverkkojen reitityksessä

- Tarkastelemme kahta eri versiota ongelmasta löytää lyhimmät polut annetusta lähtösolmusta kaikkiin muihin verkon solmuihin:
  - jos kaarten painot voivat olla mielivaltaisia, ongelma ratkeaa ajassa $\mathcal{O}(|V||E|)$ soveltamalla **Bellmanin-Fordin algoritmia**
  - jos kaarten painot oletetaan ei-negatiivisiksi, ongelma ratkeaa ajassa $\mathcal{O}((|V|+|E|)\log|V|)$ soveltamalla **Dijkstran algoritmia**
- Tapaukseen, jossa halutaan lyhin polku lähtösolmusta u vain yhteen annettuun kohdesolmuun v, ei tunneta yleisessä tapauksessa tehokkaampaa ratkaisua kuin laskea samalla kaikki lyhimmät polut lähtösolmusta
- Lisäksi
  - lyhimmät polut kaikkien $|V|^2$ solmuparin välille voidaan kerralla laskea ajassa $\mathcal{O}(|V|^3)$ soveltamalla **Floyd-Warshallin algoritmia**, joka sallii myös negatiivisia kaaripainoja

- Olkoon $G = (V, E)$ suunnattu verkko ja s eräs verkon solmu
- Oletetaan että jokaiseen kaareen $(u, v) \in E$ on liitetty pituus $w(u, v)$
- Oletetaan lisäksi että jos $(u, v) \notin E$ niin $w(u, v) = \infty$ ja $w(v, v) = 0$, eli jos solmuja ei yhdistä kaari, niiden välisen reitin pituus on ääretön ja jokaisesta solmusta matka itseensä on nolla
- Huomautus: Jos negatiiviset painot ovat sallittuja, voi esiintyä negatiivisen painoinen sykli: kiertämällä sykliä mielivaltaisen monta kertaa, saataisiin mielivaltaisen pieni ("$-\infty$") polun pituus
- **Bellman-Fordin** ja **Dijkstran** algoritmit käyttävät aputaulukkoja $distance$ ja $path$, joihin talletetaan tietoa verkon solmuihin liittyen
  - $distance[v]$ kertoo mikä on solmun v etäisyysarvio solmusta s
  - $path[v]$ on solmu, joka edeltää solmua v toistaiseksi tunnetulla lyhimmällä polulla

- Tarvitsemme molemmissa algoritmeissa samat aliohjelmat: alustus ja päivitysoperaatio, jota kutsutaan kaaren löysäämiseksi (relaxation)

Initialise-Single-Source(G,s)
```
for kaikille solmuille v ∈ V
    distance[v] = ∞
    path[v] = NIL
distance[s] = 0
```

Relax(u,v,w)
```
if distance[v] > distance[u] + w(u,v)
    distance[v] = distance[u]+w(u,v)
    path[v] = u
```

- Löysääminen voi tuntua omituiselta termiltä, mutta se viittaa siihen, että operaation lopputuloksena oleva ehto $distance[v] \leq distance[u] + w(u, v)$ on "relaksoitunut"

- Ongelmana on, että kun relaksaatio-operaatiossa muutetaan $distance[v]$, niin solmusta v alkaviin kaariin (v, r) voi syntyä jännitteitä, eli tilanteita

  $distance[r] > distance[v] + w(v, r)$, joita pitää taas löysätä

- Algoritmit eroavat siinä, missä järjestyksessä suoritetaan löysäämisoperaatiota

- **Dijkstran algoritmi** järjestää operaatiot niin tehokkaasti, että kutakin kaarta tarvitsee löysätä vain kerran. Tämän onnistuminen voidaan kuitenkin taata vain, kun verkossa ei ole negatiivisia painoja

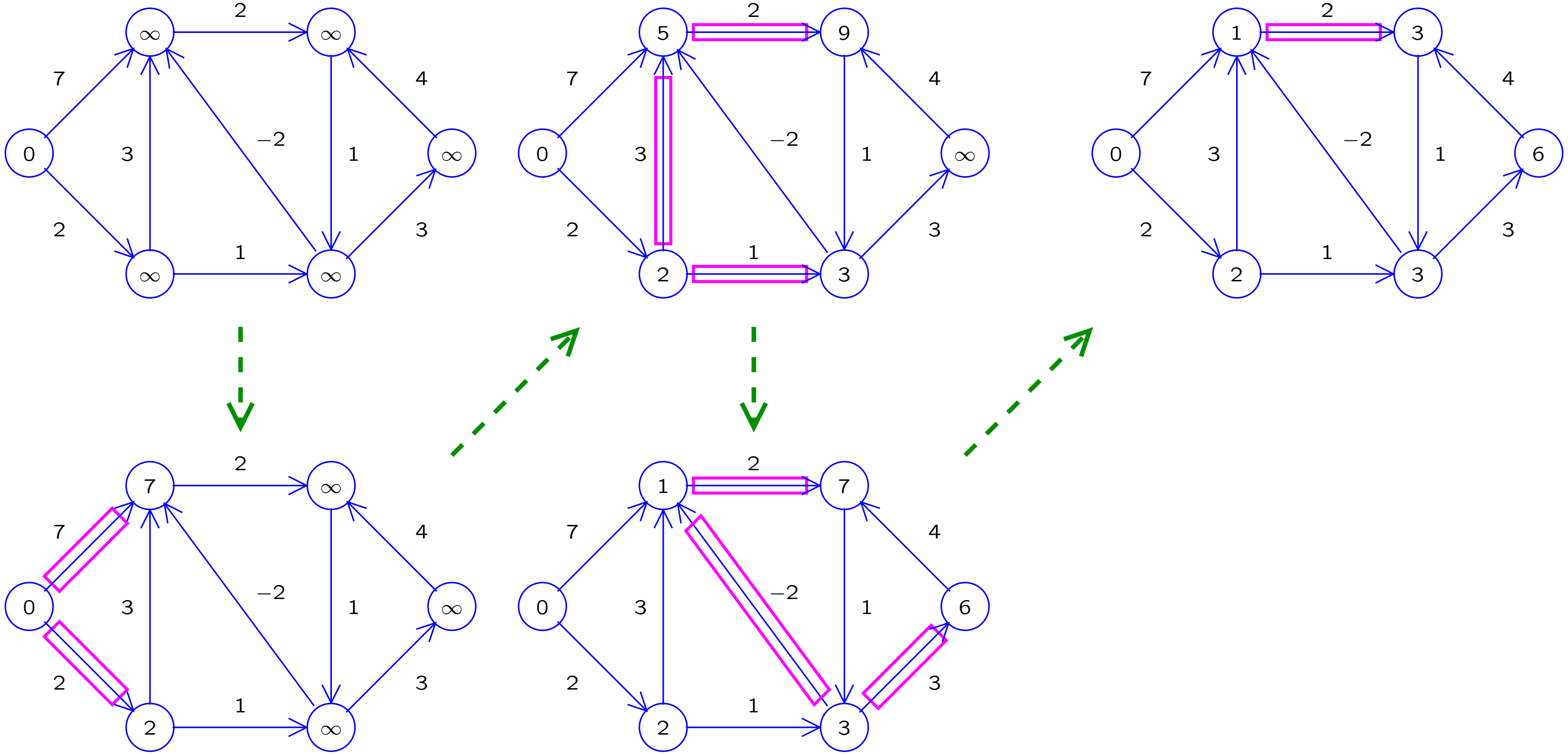
Bellman-Fordin algoritmi [Richard Bellman ja Lester Ford Jr., 1958]

- Tässä algoritmissa löysäämisjärjestystä ei yritetäkään optimoida: Kaikki kaaret käydään järjestyksessä läpi useita kertoja, kunnes tulos on valmis

- Kuten pian näemme, $|V| - 1$ läpikäyntiä riittää, ellei verkossa ole negatiivisen painoisia syklejä

- Jos tämän ajan jälkeen on jännitteitä jäljellä, algoritmi palauttaa false merkiksi negatiivisen painoisesta syklistä

```
Bellman-Ford(G,w,s)
Initialise-Single-Source(G,s)
for i = 1 to |V| − 1
    for jokaiselle kaarelle (u,v) ∈ E
        Relax(u,v,w)
for jokaiselle kaarelle (u,v) ∈ E
    if distance[v] > distance[u] + w(u,v)
        return false
return true
```


∞ 2 ∞
7 4
0 3 −2 1 ∞
2 3
∞ 1 ∞
5 2 9
7 4
0 3 −2 1 ∞
2 3
2 1 3
1 2 3
7 4
0 3 −2 1 6
2 3
2 1 3
7 2 ∞
7 4
0 3 −2 1 ∞
2 3
2 1 ∞
1 2 7
7 4
0 3 −2 1 6
2 3
2 1 3

- Seuraava lemma sanoo oleellisesti, että $|V|-1$ iteraatiota riittää kaikkien jännitteiden löysäämiseen, ellei verkossa ole negatiivisen painoisia kehiä:

- **Lemma 8.5**: Jos

  – solmusta s on polku solmuun v ja

  – solmusta s ei voi saavuttaa mitään negatiivisen painoista sykliä,

  niin algoritmin **Bellman-Ford** suorituksen päättyessä $distance[v] = \delta(s, v)$, missä $\delta(s, v)$ on lyhimmän polun $s \rightsquigarrow v$ paino.

- **Todistus:** Oletusten mukaan jokin yksinkertainen polku $\pi = (s, v_1, \ldots, v_{k-1}, v)$ on lyhin polku $s \rightsquigarrow v$. Merkitään $s = v_0$ ja $v = v_k$. Nyt $\pi_i = (v_0, \ldots, v_i)$ on lyhin polku $s \rightsquigarrow v_i$ kaikilla $1 \leq i \leq k$.

  Selvästi aina pätee, että $distance[u] \geq \delta(s, u)$. Huomaa, että $distance[u]$ ei koskaan kasva millään u, eli jos ehto $distance[u] = \delta(s, u)$ kerran tulee voimaan, se pysyy voimassa.

Osoitetaan induktiolla indeksin i suhteen, että kun for-silmukkaa rivillä 2 on iteroitu i kertaa, pätee $distance[v_i] = \delta(s, v_i)$. Koska $k \leq |V| - 1$, tästä seuraa väite.

Tapaus $i = 0$ pätee selvästi.

Oletetaan, että $distance[v_i] = \delta(s, v_i)$ pätee iteraation i jälkeen. Kun on seuraavan kerran suoritettu **Relax**(v_i, v_{i+1}, w), pätee

$$\begin{aligned} \delta(s, v_{i+1}) &\leq distance[v_{i+1}] \\ &\leq distance[v_i] + w(v_i, v_{i+1}) \\ &= \delta(s, v_i) + w(v_i, v_{i+1}) \\ &= \delta(s, v_{i+1}), \end{aligned}$$

missä viimeinen askel perustuu oletukseen, että $s \rightsquigarrow v_i \rightarrow v_{i+1}$ on lyhin polku $s \rightsquigarrow v_{i+1}$. $\square$

- Seuraavat kaksi lemmaa perustelevat, että algoritmin lopussa tehtävä testi menee oikein

- **Lemma 8.6**: Jos solmusta s ei voi saavuttaa negatiivisen painoisia kehiä, **Bellman-Ford** palauttaa true.

- **Todistus:** Edellisen lemman mukaan lopuksi pätee $distance[v] = \delta(s,v)$ kaikilla $v \in V$. Siis kaikilla $u \in V$ saadaan

$$\begin{aligned} distance[v] &= \delta(s,v) \\ &\leq \delta(s,u) + w(u,v) \\ &= distance[u] + w(u,v). \end{aligned}$$

$\square$

- **Lemma 8.7**: Jos verkossa on sykli $c = (v_0, v_1, \ldots, v_k)$, missä $v_k = v_0$ ja
  - v_0 on saavutettavissa solmusta s ja
  - syklin paino $w(c) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$ on negatiivinen,

  niin **Bellman-Ford** palauttaa false.

- **Todistus:** Tehdään vastaoletus, että **Bellman-Ford** palauttaa true. Tällöin erityisesti $\mathit{distance}[v_{i+1}] \leq \mathit{distance}[v_i] + w(v_i, v_{i+1})$ kaikilla i. Saadaan

$$
\begin{array}{rcl}
\mathit{distance}[v_k] & \leq & \mathit{distance}[v_{k-1}] + w(v_{k-1}, v_k) \\
& \leq & \mathit{distance}[v_{k-2}] + w(v_{k-2}, v_{k-1}) + w(v_{k-1}, v_k) \\
& \cdots & \\
& \leq & \mathit{distance}[v_0] + \displaystyle\sum_{i=1}^{k} w(v_{i-1}, v_i).
\end{array}
$$

  Koska $v_0 = v_k$ ja $\mathit{distance}[v_0] < \infty$, pätee $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$, mikä on ristiriita oletuksen $w(c) < 0$ kanssa. $\square$

- Edellisten kolmen lemman perusteella
  - **Bellman-Ford** palauttaa true, jos ja vain jos solmusta s ei voi saavuttaa negatiivisen painoista sykliä ja
  - tällöin lopuksi pätee $distance[v] = \delta(s, v)$ kaikilla $v \in V$
- Algoritmin aikavaativuus on selvästi $\mathcal{O}(|V|\,|E|)$

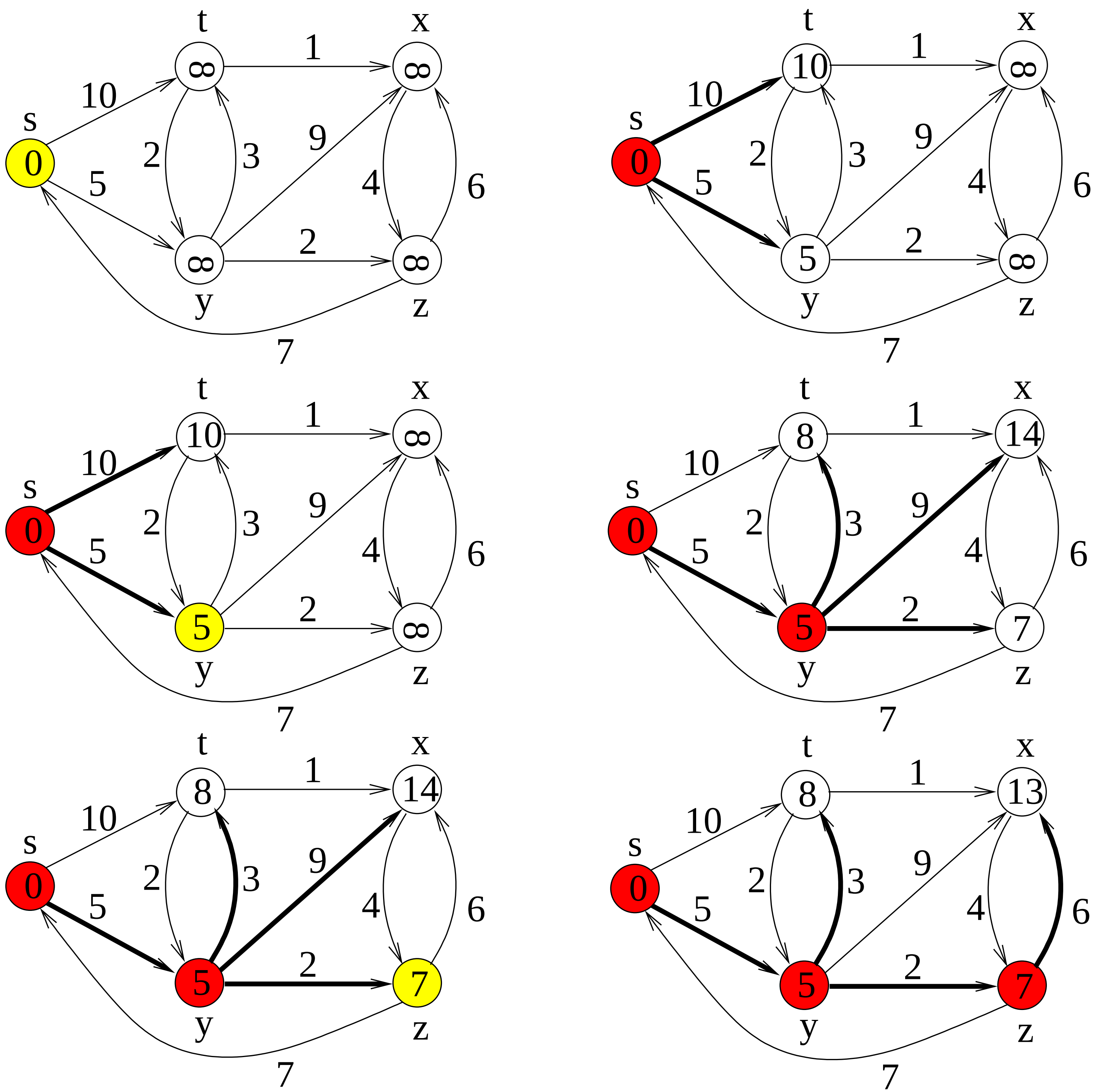
Dijkstran algoritmi [Edsger Dijkstra, 1956]

- **Dijkstran algoritmi** pitää yllä joukkoa S, joka muodostuu solmuista joiden lyhin etäisyys solmuun s on jo selvitetty
- Algoritmi valitsee toistuvasti solmun $u \in V \setminus S$, jonka etäisyysarvio solmuun s on pienin
  - valittu solmu u lisätään joukkoon S, ja
  - kaikkien solmun u vierussolmujen v etäisyysarvio solmuun s sekä polkutieto $path[v]$ päivitetään

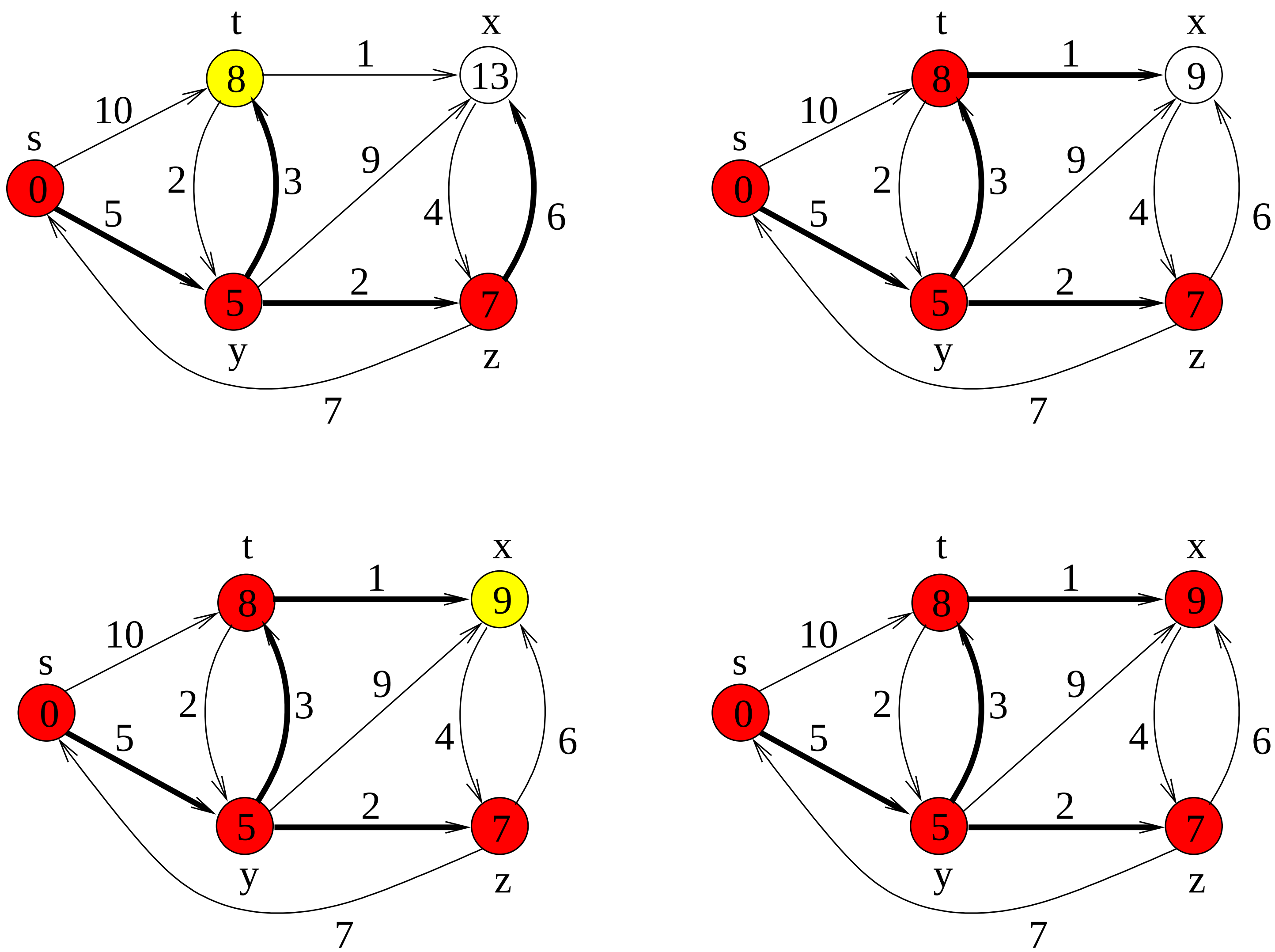
- Algoritmi osittain abstraktissa muodossa:

  Dijkstra(G,w,s)
  ```
  Initialise-Single-Source(G,s)
  S = ∅
  while ( kaikki solmut eivät vielä ole joukossa S )
      valitse solmu u ∈ V \ S, jonka etäisyysarvio lähtösolmuun s on lyhin
      S = S ∪ {u}
      for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
          Relax(u,v,w)
  ```

- Seuraavilla sivuilla on esimerkki algoritmin toiminnasta
  - joukkoon S lisätyt alkiot tummanharmaita (värikuvassa punaisia)
  - käsittelyvuorossa oleva alkio vaaleanharmaa (värikuvassa keltainen)



t x s y z
0 ∞ ∞ ∞ ∞
10 1 2 3 9 5 4 6 2 7
t x s y z
0 10 ∞ 5 ∞
10 1 2 3 9 5 4 6 2 7
t x s y z
0 10 ∞ 5 ∞
10 1 2 3 9 5 4 6 2 7
t x s y z
0 8 14 5 7
10 1 2 3 9 5 4 6 2 7
t x s y z
0 8 14 5 7
10 1 2 3 9 5 4 6 2 7
t x s y z
0 8 13 5 7
10 1 2 3 9 5 4 6 2 7



- Tieto lyhimmistä poluista on kuvattu paksunnettuina kaarina

- Miten algoritmin rivi 4, jossa on valittava solmu $u \in V \setminus S$, jonka etäisyys lähtösolmuun s on lyhin, voidaan toteuttaa tehokkaasti?
  – yksi mahdollisuus olisi tietysti käydä läpi kaikki solmut joukosta $V \setminus S$
- Tehokkaampaan ratkaisuun päästään käyttämällä aputietorakenteena minimikekoa H
  – solmut $v \in V \setminus S$ pidetään keossa
  – solmun v avaimena sen etäisyysarvion $distance[v]$ arvo
  – näin seuraavaksi käsiteltävä solmu saadaan nopeasti operaatiolla **heap-delete-min**
  – jos valitun solmun u jonkin vierussolmun v etäisyysarviota pienennetään, kutsutaan sille **heap-decrease-key**-operaatiota, joka asettaa solmun v taas oikealle paikalle keossa

- Algoritmi joka hyödyntää minimikekoa

```
Dijkstra-with-heap(G,w,s)
Intialise-Single-Source(G,s)
S = ∅
for kaikille solmuille v ∈ V
    heap-insert(H,v,distance[v])
while not empty(H)
    u = heap-del-min(H)
    S = S ∪ {u}
    for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
        Relax(u,v,w)
        heap-decrease-key(H,v,distance[v])
                    // ei tee mitään, jos distance[v] ei ole muuttunut
```

- Huom: Joukkoa S ei oikeastaan enää tarvita, sillä joukossa S ovat täsmälleen ne alkiot jotka eivät ole keossa H

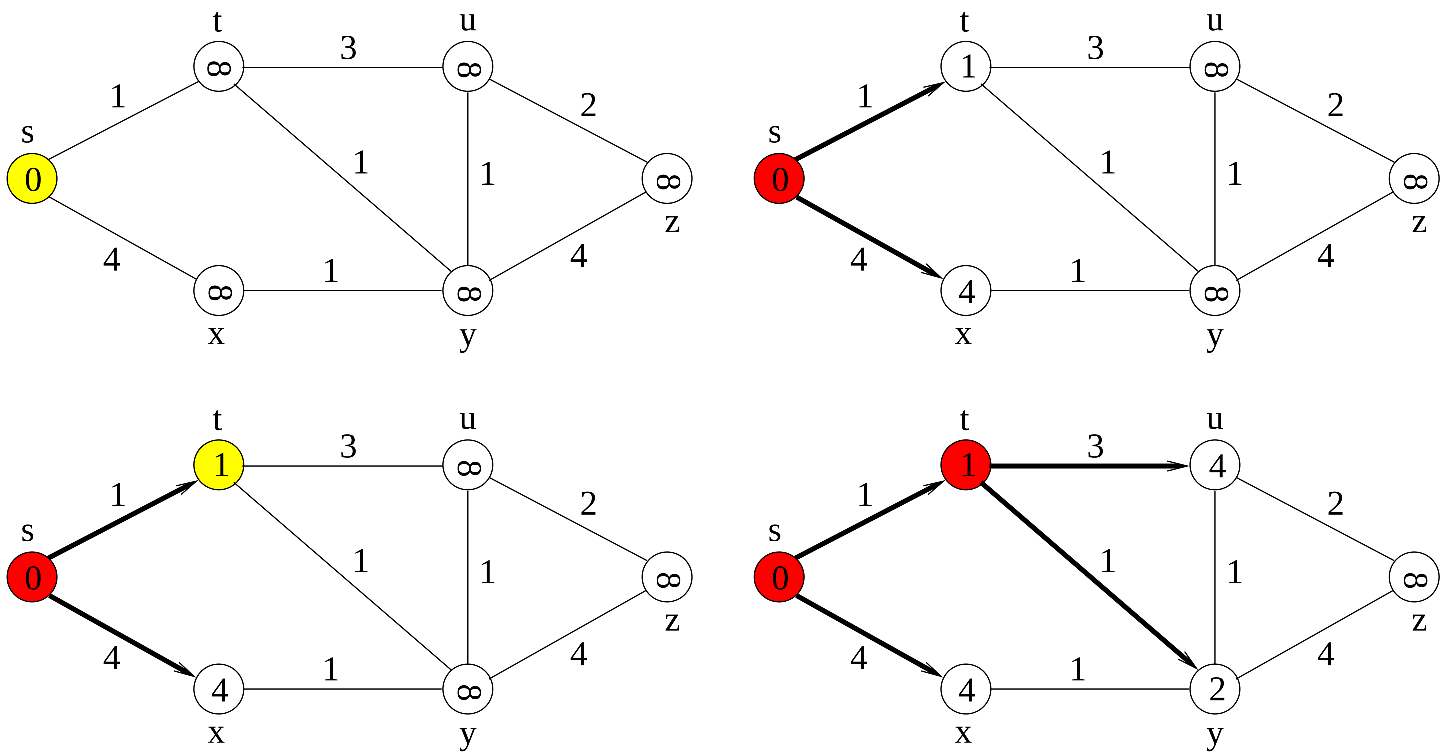
- Pidetään kuitenkin S mukana sillä se selkeyttää pian esitettävää oikeellisuustodistusta

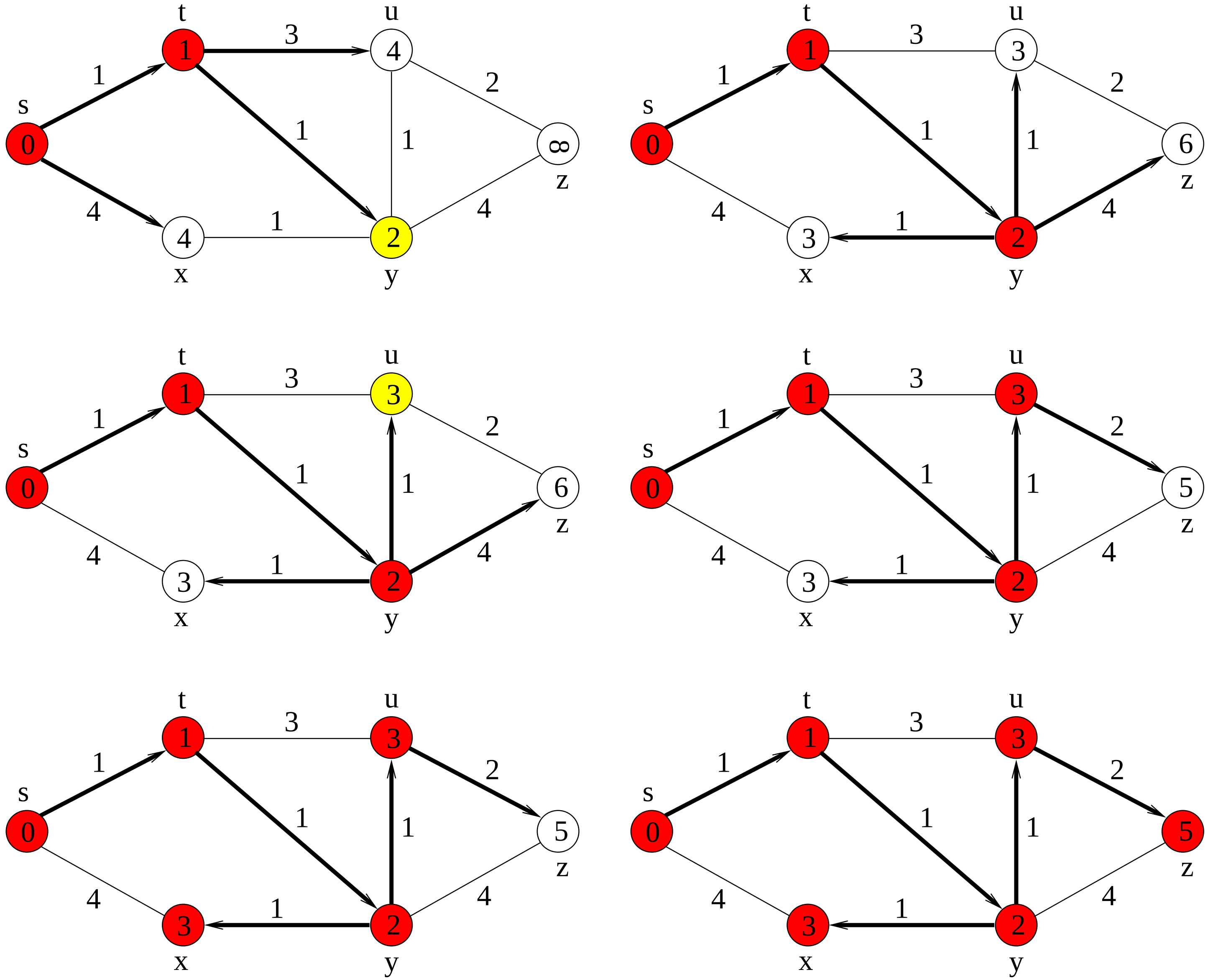
- Algoritmin suorituksen jälkeen lyhin polku $s \leadsto v$ saadaan selville seuraavasti:
  - $path[v]$ kertoo minkä solmun kautta lyhin polku $s \leadsto v$ saapuu solmuun v
  - solmuun $path[v]$ lyhin polku saapuu solmun $path[path[v]]$ kautta, jne
  - laitetaan pinoon $path[v]$, $path[path[v]]$, $path[path[path[v]]]$ ja tulostetaan pinon sisältö
  - näin saadaan tulostettua polulla koko polku $s \leadsto v$ alusta loppuun

- Algoritmina:

```
shortest-path(G,v)
u = path[v]
while u ≠ s
    push(pino,u)
    u = path[u]
print("lyhin polku solmusta s solmuun v kulkee seuraavien solmujen kautta:")
while not empty(pino)
    u = pop(P)
    print(u)
```

- Toinen esimerkki **Dijkstran algoritmin** toiminnasta, tällä kertaa kyseessä suuntaamaton verkko





t
u
s
z
x
y
1 3 4 0 ∞ 4 2
1 3 1 1 2 4 4 1
t
u
s
z
x
y
1 3 0 6 3 2
1 3 1 1 2 4 4 1
t
u
s
z
x
y
1 3 0 6 3 2
1 3 1 1 2 4 4 1
t
u
s
z
x
y
1 3 0 5 3 2
1 3 1 1 2 4 4 1
t
u
s
z
x
y
1 3 0 5 3 2
1 3 1 1 2 4 4 1
t
u
s
z
x
y
1 3 0 5 3 2
1 3 1 1 2 4 4 1

- **Dijkstran algoritmin** vaativuus
  - rivien 1-2 alustustoimet vievät aikaa selvästi $\mathcal{O}(|V|)$
  - käytössä siis minimikeko, ja kutsutaan keko-operaatioita **heap-insert**, **heap-del-min** ja **heap-decrease-key**
  - keko-operaatioiden vaativuus on $\mathcal{O}(\log n)$ jos keossa n alkiota
  - riveillä 3-4 tehdään $|V|$ kappaletta **heap-insert**-operaatioita, aikaa siis kuluu $\mathcal{O}(|V| \log |V|)$; arvot ovat 0 ja ∞, joten keon voi myös luoda ajassa $\mathcal{O}(|V|)$
  - rivien 5-10 toistolauseessa $\mathcal{O}(\log |V|)$ aikaa vievää operaatiota **heap-del-min** kutsutaan kullekin solmulle kerran, eli yhteensä $|V|$ kertaa
  - koska jokainen solmu v lisätään joukkoon S vain kertaalleen, käydään kukin vieruslista läpi täsmälleen kerran
  - jokaista kaarta siis tutkitaan **Relax**-aliohjelmassa kerran, eli $\mathcal{O}(\log |V|)$ aikaa vievää **heap-decrease-key**-operaatiota kutsutaan maksimissaan $|E|$ kertaa
  - yhteensä toistolauseessa kuluu aikaa $\mathcal{O}((|E| + |V|) \log |V|)$, joka on samalla koko algoritmin aikavaativuus

  - algoritmin alussa kaikki solmut ovat keossa ja tämän jälkeen keko alkaa pienentyä solmujen siirtyessä samalla joukkoon S
  - tilavaativuus on siis selvästi $\mathcal{O}(|V|)$

- Joskus riittää tietää pelkästään kahden solmuparin s ja v välinen lyhin polku
- Otetaan lähtösolmuksi s ja suoritetaan algoritmia kunnes solmu v lisätään joukkoon S
- Tämän jälkeen voidaan lopettaa sillä lyhin polku $s \rightsquigarrow v$ on jo selvinnyt
  - itseasiassa ei ole $\mathcal{O}$-analyysin mielessä helpompaa etsiä lyhintä polkua solmusta s yhteen solmuun v kuin solmusta s kaikkiin solmuihin
- **Dijkstran algoritmi** noudattaa ns. ahnetta (engl. greedy) strategiaa:
  - rivillä 6 valitaan aina käsiteltäväksi se solmu mikä on lähimpänä aloitussolmua s
  - ahneudella tarkoitetaan tässä sitä että algoritmi pyrkii joka hetkellä juuri silloin "parhaalta" vaikuttavaan ratkaisuun
  - ei ole itsestäänselvää että ahne strategia tuottaa nimenomaan lyhimmät polut, mutta **Dijkstran algoritmin** tapauksessa näin on
  - todistus perustuu seuraavaan havaintoon: kaikille solmuille $v \in S$ pätee: $distance[v]$ on sama kuin lyhimmän polun $s \rightsquigarrow v$ pituus
  - eli kun solmu on lisätty joukkoon S, sen lyhin etäisyys aloitussolmuun on selvinnyt
- Perustellaan seuraavassa tarkemmin miksi **Dijkstran algoritmi** toimii oikein

- Näytetään, että

  (1) solmuille $v \in S$ pätee: $\mathit{distance}[v]$ on sama kuin lyhimmän polun $s \rightsquigarrow v$ pituus

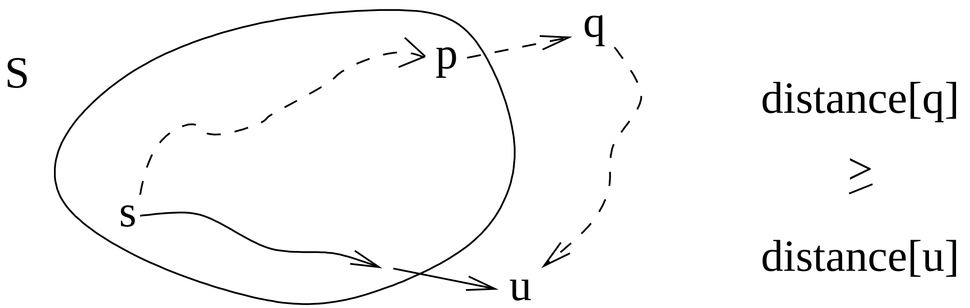
  (2) solmuille $v \in V \setminus S$ pätee: $\mathit{distance}[v]$ on lyhimmän tunnetun, eli käytännössä joukon S solmujen kautta kulkeva polun $s \rightsquigarrow v$ pituus

- Väittämät ovat voimassa alustuksen jälkeen:
  - alussa $S = \{s\}$ ja $\mathit{distance}[s] = 0$, ja koska kaikki kaaripainot positiivisia, ei lyhempää polkua solmuun s ole olemassa
  - algoritmi päivittää s:n vierussolmujen v etäisyysarvioiksi $w(s, v)$, eli selvästi kaikkien solmujen $v \in V \setminus S$ etäisyysarvio on alussa lyhin tunnettu etäisyys

- Oletetaan nyt, että väittämät ovat voimassa tietyllä suoritushetkellä, ja näytetään että ne säilyvät voimassa kun uusia solmuja lisätään joukkoon S

- Olkoon u seuraavaksi käsittelyvuorossa oleva eli joukkoon S lisättävä solmu
  - Käsittelyvuorossa oleva solmu on siis joukon S ulkopuolisista solmuista se, jonka etäisyysarvio on pienin
  - Kun solmu u lisätään joukkoon S ei sen etäisyysarviota enää muuteta
  - eli jotta väittämä (1) pysyisi voimassa, on lisättävän solmun etäisyysarvion jo oltava sama kuin lyhimmän polun $s \rightsquigarrow u$ pituus

- Voiko verkossa olla vielä lyhyempää polkua s:stä u:hun kuin joukon S kautta kulkeva $distance[u]$:n pituinen polku?
  - oletetaan, että verkossa olisi vielä lyhempi polku solmuun u
  - tämän polun täytyisi käydä jossain vaiheessa joukon S ulkopuolella
  - jaetaan tämä polku osiin: $s \leadsto p \rightarrow q \leadsto u$ missä q on polun ensimmäinen solmu, joka on joukon S ulkopuolella



  - nyt $distance[q] = distance[p] + w(p, q) \geq distance[u]$ sillä algoritmi valitsi solmun u käsittelyyn ennen q:ta
  - polun loppuosan $q \leadsto u$ pituuden pitäisi olla negatiivinen, jotta polku olisi todella lyhyempi kuin $distance[u]$. **Dijkstran algoritmi** kuitenkin käsittelee ainoastaan tilanteita, joissa kaarten paino on ei-negatiivinen
  - eli kun solmu u lisätään joukkoon ei koko verkossa voi olla polkua $s \leadsto u$ jonka pituus olisi pienempi kuin $distance[u]$
  - väittämä (1) siis säilyy voimassa uusia solmuja lisättäessä

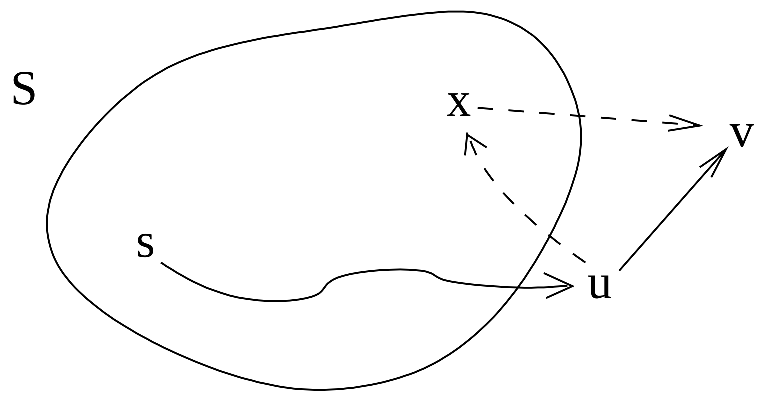
- On vielä osoitettava, että algoritmi päivittää oikein etäisyysarviot taulukkoon $distance$, eli että väittämä (2) säilyy voimassa uusia solmuja joukkoon S lisättäessä

- Olkoon edelleen u seuraavaksi joukkoon S lisättävä solmu, ja tarkastellaan u:n mielivaltaista vierussolmua v joka ei vielä ole joukossa S

- Jos polku $s \rightsquigarrow u \rightarrow v$ on lyhempi kuin aiemmin tunnettu lyhin polku $s \rightsquigarrow v$, asettaa algoritmi **Relax**-aliohjelmassa v:lle uuden etäisyysarvion

- Onko näin asetettu uusi etäisyysarvio pienin etäisyys polulle $s \rightsquigarrow v$, joka voidaan tässä suoritusvaiheessa tietää, eli jossa kaikki solmut polun varrella kuuluvat joukkoon S?

- Ainoa mahdollisuus, että näin ei olisi, on alla kuvattu tilanne, jossa v:hen kulkisi vielä lyhempi polku $s \rightsquigarrow u \rightarrow x \rightarrow v$, eli missä solmusta u palattaisiin vielä johonkin joukon S solmuun x



- Kuvatun kaltaista polkua ei kuitenkaan voi olla olemassa, sillä koska x laitettiin joukkoon S ennen solmua u, ei lyhin polku solmuun x voi kulkea u:n kautta

- Algoritmi siis päivittää oikein joukon S ulkopuolisten solmujen etäisyysarviot eli väittämä (2) säilyy voimassa uusia solmuja lisättäessä

- On siis osoitettu, että

  (1) solmuille $v \in S$ pätee: $distance[v]$ on sama kuin lyhimmän polun $s \leadsto v$ pituus

  (2) solmuille $v \in V \setminus S$ pätee: $distance[v]$ on lyhimmän tunnetun, eli käytännössä joukon S solmujen kautta kulkeva polun $s \leadsto v$ pituus

  ovat voimassa algoritmin suorituksen alussa ja pysyvät voimassa kun solmuja lisätään joukkoon S

- Lopussa kaikki saavutettavissa olevat solmut ovat joukossa S, eli väittämästä (1) seuraa, että algoritmi löytää lyhimmän polun kaikkiin saavutettavissa oleviin solmuihin

Esimerkki ongelmanratkaisusta Dijkstralla: vankilapako

- Syötteenä saadaan vankilan pohjapiirros matriisina $B[0 \ldots 2 \cdot m][0 \ldots 2 \cdot m]$
- Jokainen paikka $B[i][j]$ on joko käytävää tai muuria

●			●	●
●		●		●
●	●	X	●	
●			●	●
	●	●		●

- Vanki on aluksi vankilan keskipaikassa $B[m][m]$, joka on käytävää
- Tavoitteena on päästä vankilasta sen jonkin reunan yli vapauteen
- Vankilassa saa liikkua seuraavasti:
  - Nykyisestä paikasta voi siirtyä mihin tahansa naapuripaikkaan, mutta ei kulmittain.
  - Käytävillä voi hiiviskellä, mutta niillä ei kannata maleksia turhaan
  - Muuriin täytyy ensin kaivautua ennen kuin siihen voi kulkea

- Pakosuunnitelmassa on tärkeintä, että ei kaiveta yhtään enempää kuin aivan välttämätöntä.

- Myös hiiviskelyä pitäisi välttää, jos se on mahdollista

- Mallinnetaan pakoreitti lyhyimpänä polkuna verkossa G
  - $V =$ vankilan paikat sekä lisäpaikka t vapaudessa
  - Kaari $(p, q) \in E$ jos ja vain jos paikka q on paikan p naapuripaikka
  - Jokaisen reunapaikan naapurina on myös lisäpaikka t

- Kaarilla on painot niiden maalisolmun (kohdepaikan) q mukaan:
  - Jos q on käytävllä, $w(p, q) = 1$
  - Jos q on muurilla $w(p, q) = (2 \cdot m + 1)^2$ eli suurempi kuin koko vankilan pinta-ala (ilman aloituspaikkaa)
  - Jos q on vapaudessa eli $q = t$, $w(p, q) = 0$

- Näillä kaaripainoilla yksikin kaivautuminen johtaa suurempaan etäisyyteen kuin pisinkään hiiviskely

- Eräs mahdollisimman hyvä pakosuunnitelma saadaan siis seuraavasti:
  - Suoritetaan **Dijkstran algoritmia** verkossa G alkusolmusta $B[m][m]$ lähtien, kunnes solmu t liitetään joukkoon S
  - Tällöin pakosuunnitelma saadaan seuraamalla taulukon *path* polkutietoja solmusta t taaksepäin ja kääntämällä saadun polun järjestys:

$$s \to \ldots \to path[path[t]] \to path[t] \to t$$

- Verkkoa G ei tarvitse tallentaa erikseen, vaan vieruslistat lasketaan algoritmin suorituksen aikana nykyisen solmun indekseistä i, j:
  - Solmun $B[i][j]$ mahdolliset vierussolmut ovat $B[i+1][j]$, $B[i-1][j]$, $B[i][j+1]$ ja $B[i][j-1]$
  - Indeksoinnin ylitys tarkoittaa lisäsolmua t
  - Kaaripaino katsotaan kohdesolmusta
- Vankilapako on (yli)yksinkertaistettu version VLSI-piirisuunnittelusta, jossa komponentti sijoitetaan piisirulle valitun ruudukon pisteisiin ja minimoitavia suureita ovat mm. yhteyksien pituus ja kytkentöjen risteyskohtien lukumäärä
  - Käytännössä VLSI-suunnittelun kriteerit ovat monimutkaisempia ja algoritmisesti hankalia

Kaikki lyhimmät polut

- Merkintöjen yksinkertaistamiseksi oletetaan, että solmujoukko on $V = \{1, \ldots, n\}$, missä n on solmujen lukumäärä

- Halutaan muodostaa $n \times n$-matriisi D, missä $D[i, j]$ on lyhimmän polun paino $i \rightsquigarrow j$, eli halutaan selvittää jokaisen solmun välisen lyhimmän polun paino

- Jos kaarten painot ovat ei-negatiivisia, ongelma voidaan ratkaista suorittamalla **Dijkstran algoritmi** n kertaa

- Olettamalla prioriteettijono toteutetuksi kekona saadaan aikavaativuudeksi $\mathcal{O}(|V| \cdot (|V| + |E|) \log |V|) = \mathcal{O}((|V|^2 + |V||E|) \log |V|)$. Jos verkko on tiheä (kaaria on melkein jokaisen solmun välillä) eli $|E| = \mathcal{O}(|V|^2)$, tämä on $\mathcal{O}(|V|^3 \log |V|)$

- **Floydin-Warshallin** algoritmi tai **Floydin** algoritmi (Robert Floyd, 1962, aikaisemmin Bernard Roy, 1959, ja myös Stephen Warshall, 1962) ratkaisee ongelman ajassa $\mathcal{O}(|V|^3)$, mikä siis tiheillä verkoilla on asymptoottisesti parempi kuin **Dijkstran** toistaminen

- Lisäksi **Floyd-Warshall** sallii negatiiviset painot (mutta ei negatiivisia syklejä), on helppo toteuttaa ja vakiokertoimet ovat pieniä

- Verkko oletetaan annetuksi vierusmatriisina $A[1\,..\,n, 1\,..\,n]$. Oletetaan $A[i,j] = \infty$, jos $(i,j) \notin E$. Jos verkko on vieruslistamuodossa, on tieto kaarista konvertoitavissa matriisimuotoon ajassa $\mathcal{O}(|V|^2)$

- Algoritmi laskee välituloksenaan etäisyysmatriiseja $D^0, D^1, D^2, \ldots D^n$

- Matriisin D^k alkio $D^k[i,j]$ pitää yllä tietoa siitä mikä on solmujen i, j lyhin etäisyys, jos niitä yhdistävä polku käyttää ainoastaan solmuja $1, 2, \ldots, k$, eli
  - $D^0[i,j]$ kertoo mikä on solmujen i ja j etäisyys jos niiden välisellä polulla ei ole mitään solmua, toisin sanoen, mikä on i:n ja j:n välisen mahdollisen suoran kaaren pituus eli $D^0[i,j] = A[i,j]$
  - $D^1[i,j]$ kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välisellä polulla käydään korkeintaan solmussa 1
  - $D^2[i,j]$ kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välisellä polulla käydään korkeintaan solmuissa 1 ja 2 (molemmissa tai vain toisessa tai ei kummassakaan)
  - ...
  - $D^n[i,j]$ kertoo mikä on solmujen i ja j pienin etäisyys siten, että niiden välisellä polulla voidaan käydä missä tahansa verkon solmuista

- Eli matriisi D^n kertoo halutun lopputuloksen kaikille solmuille $i, j \in V$
- Miten matriisit $D^0, D^1, D^2, \ldots D^n$ saadaan laskettua?
  - D^0 siis saadaan suoraan siirtymämatriisista $D^0[i,j] = A[i,j]$
- Entä D^1 joka kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välisellä polulla käydään korkeintaan solmussa 1?
  - solmujen i ja j pienin etäisyys silloin kun niiden välinen polku käy korkeintaan solmussa 1 on selvästi joko kaaren $i \rightarrow j$ paino tai polun $i \rightarrow 1 \rightarrow j$ pituus
  - jälkimmäisessä vaihtoehdossa polku $i \rightsquigarrow j$ siis kulkee solmun 1 kautta
  - eli $D^1[i,j] = min\{D^0[i,j], D^0[i,1] + D^0[1,j]\}$
- Entä D^2 joka kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välinen polku saa käydä solmuissa 1 ja 2?
  - $D^1[i,j]$ kertoo pienimmän etäisyyden jos välissä käydään korkeintaan solmussa 1
  - vielä lyhempi polku voi löytyä, jos kuljetaan solmun 2 kautta $i \rightsquigarrow 2 \rightsquigarrow j$, tämä ei tosin ole välttämättä lyhempi kuin jo tunnettu solmua 2 hyödyntämätön polku $i \rightsquigarrow j$
  - eli $D^2[i,j] = min\{D^1[i,j], D^1[i,2] + D^1[2,j]\}$

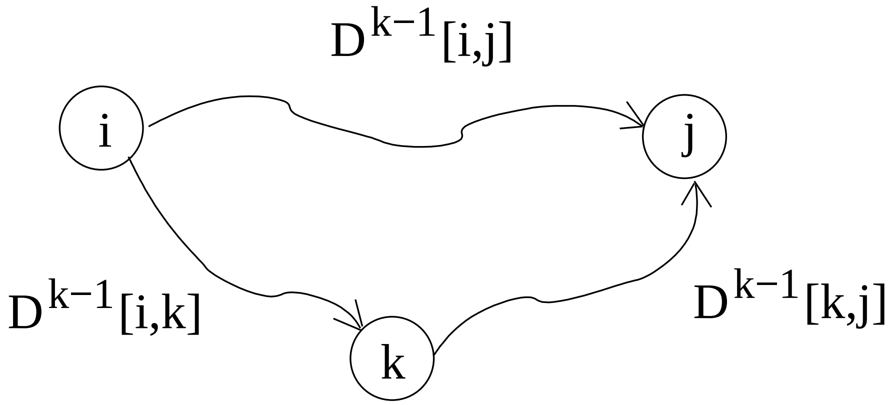
- $D^2[i,j]$ siis pitää sisällään lyhimmän polun i ja j välillä, joka voi olla joko
  - $i \to j$
  - $i \to 1 \to j$
  - $i \to 2 \to j$
  - $i \to 1 \to 2 \to j$
  - $i \to 2 \to 1 \to j$
- Huomattavaa on, että polku voi kulkea solmun 2 kautta kolmella eri tavalla
  - joko käymättä solmussa 1 tai kulkemalla ensin tai lopuksi solmun 1 kautta
  - tieto lyhimmästä korkeintaan solmun 1 kautta kulkevasta polun $i \rightsquigarrow 2$ pituudesta on huomioitu laskettaessa $D^1[i,2]$, polku on joko $i \to 2$ tai $i \to 1 \to 2$
  - vastaavasti lyhimmän korkeintaan solmun 1 kautta kulkevan polun $2 \rightsquigarrow j$ pituus on laskettu $D^1[2,j]$ siten että on huomioitu mahdollinen solmun 1 kautta kulkeminen
  - huom: polku $i \to 1 \to 2 \to 1 \to j$ ei voi tulla kyseeseen, sillä silloin $1 \to 2 \to 1$ olisi negatiivinen sykli, ja algoritmi ei toimi näissä tapauksissa
- Algoritmi siis näyttää käyvän läpi kaikki vaihtoehtoiset polut laskiessaan alkioiden $D^2[i,j]$ arvot

- Edellisestä yleistämällä saamme laskusäännön matriisin D^k alkioiden $D^k[i,j]$ arvoille:

  $D^k[i,j] = min\{\ D^{k-1}[i,j],\ D^{k-1}[i,k] + D^{k-1}[k,j]\ \}$

- Eli lyhin polku $i \rightsquigarrow j$ joka käyttää solmuja $1, 2, \ldots, k$ on sama joka ei käytä solmua k ollenkaan tai $i \rightsquigarrow k \rightsquigarrow j$ missä k:ta pienempiä solmuja ei ole käytetty

- Kuvana:



- D^0 saadaan suoraan siirtymämatriisista ja matriisin D^k kaikki arvot voidaan edellisen matriisin D^{k-1} avulla

- Näin on helppo muodostaa algoritmi joka selvittää matriisit numerojärjestyksessä päätyen matriisiin D^n joka on haluttu lopputulos

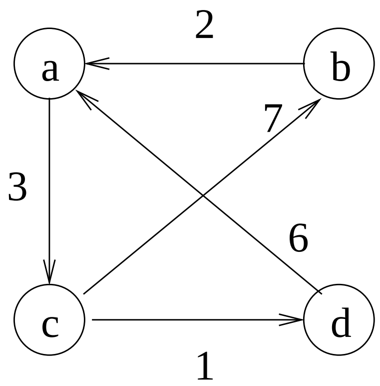
- Tästä saamme algoritmin ensimmäisen version

Floyd-Warshall-v1(A)

```
for i = 1 to n
    for j = 1 to n
        if i == j
            D^0[i, j] = 0
        else D^0[i, j] = A[i, j]

for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            D^k[i, j] = min { D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j] }
```

- Kolmen sisäkkäisen for-lauseen takia aikavaativuus on selvästi $\mathcal{O}(n^3)$
- Tilavaativuus on tässä versiossa $\mathcal{O}(n^3)$, sillä $\mathcal{O}(n^2)$:n kokoisia matriiseja on käytössä n kappaletta. Kuten pian havaitaan, apumatriisien D^k käyttöä voidaan tehostaa ja tilavaativuus saadaan putoamaan neliöiseksi
- Tarkastellaan seuraavilla sivuilla esimerkkiä algoritmin toiminnasta, selvyyden vuoksi solmut on nimetty aakkosin a, b, c ja d, matriisi D^1 vastaa lyhimpiä korkeintaan a:n kautta kulkevia polkuja, jne

- Verkko ja sitä vastaava siirtymämatriisi



$$D^0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

- Matriisia D^1 laskettaessa, selvitetään lyhentääkö a:n kautta kulkeminen joidenkin solmujen välistä polkua

$$D^0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad D^1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

- Huomataan, että $b \rightsquigarrow a \rightsquigarrow c$ ja $d \rightsquigarrow a \rightsquigarrow c$ ovat lyhempiä kuin aiemmin tunnetut a:ta käyttämättömät polut

- Matriisia D^2 laskettaessa, huomataan, että b:n kautta kulkeminen lyhentää ainoastaan polkua $c \rightsquigarrow a$

$$D^1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array} \qquad D^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

- Matriisia D^3 laskettaessa selvitetään lyhentääkö c:n kautta kulkeminen joidenkin solmujen välistä polkua

$$D^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array} \qquad D^3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

- Huomaamme, että esim. $a \rightsquigarrow c \rightsquigarrow b$ ja $a \rightsquigarrow c \rightsquigarrow d$ ovat lyhempiä kuin aiemmin tunnetut c:tä käyttämättömät polut

- Matriisia D^4 laskettaessa siis selvitetään lyhentääkö d:n kautta kulkeminen joidenkin solmujen välistä polkua, eli tässä vaiheessa on kaikki vaihtoehdot otettu huomioon ja algoritmi on selvittänyt halutun lopputuloksen

$$D^3 = \begin{array}{cc} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array} \qquad D^4 = \begin{array}{cc} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

- Tilantarve saadaan helposti pudotetuksi luokkaan $\mathcal{O}(n^2)$ toteamalla, että matriisin D^k laskemiseen ei enää tarvita matriiseja $D^0, D^1, \ldots, D^{k-2}$

- Riittää siis pitää muistissa kaksi viimeisintä matriisia $D = D^k$ ja $D' = D^{k-1}$

- Tästäkin voidaan säästää puolet: tarvitaan vain yksi matriisi D, jonka päivitys on $D[i,j] = \min\{\, D[i,j], D[i,k] + D[k,j]\,\}$

- Tämä seuraa siitä, että solmuun k rajautuvissa poluissa ei ole väliä, sallitaanko k välisolmuna, eli kaikilla i, j ja k pätee

$$D^k[i,k] = D^{k-1}[i,k] \qquad \text{ja} \qquad D^k[k,j] = D^{k-1}[k,j]$$

- Saadaan ajassa $\mathcal{O}(n^3)$ ja työtilassa $\mathcal{O}(n^2)$ toimiva algoritmi:

Floyd-Warshall-v2(A)

```
for i = 1 to n
    for j = 1 to n
        if i == j
            D[i,j] = 0
        else D[i,j] = A[i,j]

for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            if D[i,k] + D[k,j] < D[i,j]
                D[i,j] = D[i,k] + D[k,j]
```

- Algoritmin yhteydessä on mahdollista selvittää lyhimpien polkujen painojen lisäksi lyhimmät polut

- Jätämme tämän harjoitustehtäväksi

Transitiivinen sulkeuma

- Suunnatun verkon $G = (V, E)$ transitiivinen sulkeuma (engl. transitive closure) on verkko $G^* = (V, E^*)$, missä

  $(u, v) \in E^*$ jos ja vain jos verkossa G on polku $u \leadsto v$

- Esimerkki: Jos G on vahvasti yhtenäinen, niin $E^* = V \times V$

- Yleisemmin jos u ja v kuuluvat samaan vahvasti yhtenäiseen komponenttiin, niin kaikilla $w \in V$ pätee

  $(u, w) \in E^*$ jos ja vain jos $(v, w) \in E^*$
  $(w, u) \in E^*$ jos ja vain jos $(w, v) \in E^*$

- Tämän perusteella verkon transitiivinen sulkeuma voidaan laskea
  - muodostamalla ensin komponenttiverkko G^{SCC} ja
  - laskemalla sitten komponenttiverkon transitiivinen sulkeuma

- Tämä säästää laskentaa, jos G^{SCC} sisältää paljon vähemmän solmuja kuin G

- Transitiivinen sulkeuma voidaan laskea ajassa $\mathcal{O}(|V|^3)$ soveltamalla **Floydin-Warshallin algoritmia**:

  1. Asetetaan kaikille kaarille $(u, v) \in E$ jokin äärellinen paino, esim. $w(u, v) = 1$, samoin kaikille solmuille $w(u, u) = 1$

  2. Suoritetaan Floydin-Warshallin algoritmi

  3. Nyt $(u, v) \in E^*$, jos ja vain jos $D[u, v] < \infty$

- Algoritmia voidaan hieman virtaviivaistaa: Pidetään kirjaa ainoastaan siitä, onko $D[i, j]$ äärellinen vai ääretön

- Saadaan seuraava algoritmi:

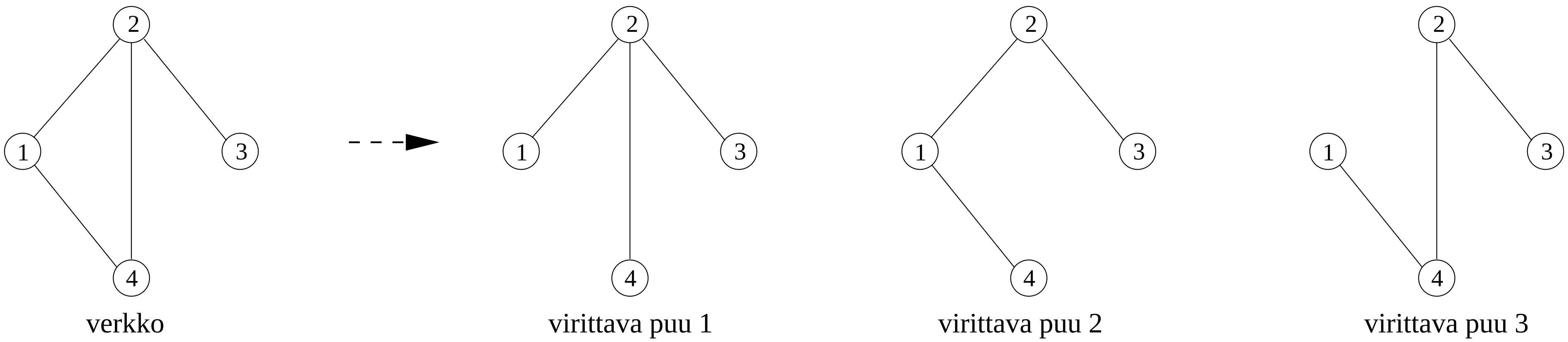
  Transitive-Closure(A)

```
for i = 1 to n
    for j = 1 to n
        if i == j or A[i, j] < ∞
            T[i, j] = 1
        else T[i, j] = 0
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            if T[i, k] = 1 and T[k, j] = 1
                    T[i, j] = 1
```

- Ajassa $\mathcal{O}(n^3)$ saadaan taulukko T, missä $T[i, j] = 1$, jos $(i, j) \in E^*$, ja $T[i, j] = 0$, muuten

Verkon virittävät puut

- Olkoon $G = (V, E)$ suuntaamaton yhtenäinen verkko
  - verkon yhtenäisyydellä tarkoitamme että kaikki verkon solmut ovat saavutettavissa toisistaan, eli
  - verkossa ei ole erillisiä osia
- Verkon G virittävä puu (engl. spanning tree) on G:n yhtenäinen syklitön aliverkko joka sisältää kaikki G:n solmut
- Huomio: virittävässä puussa on $|V| - 1$ kaarta
- Verkko ja sen virittäviä puita

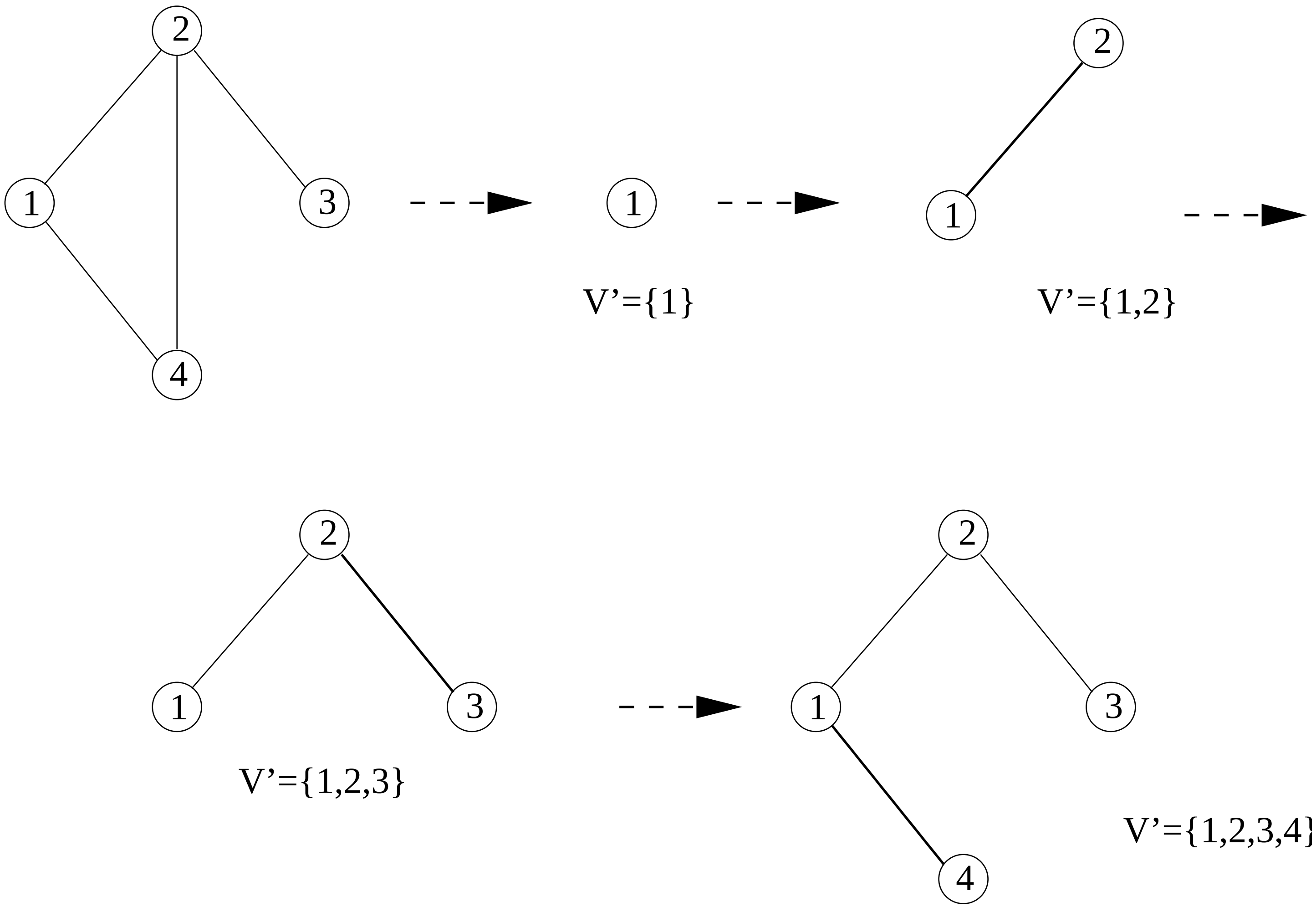


- Seuraavassa on yksinkertainen algoritmi, joka muodostaa verkolle $G = (V, E)$ jonkin virittävän puun

- Algoritmin lopussa joukossa T olevat kaaret muodostavat virittävän puun

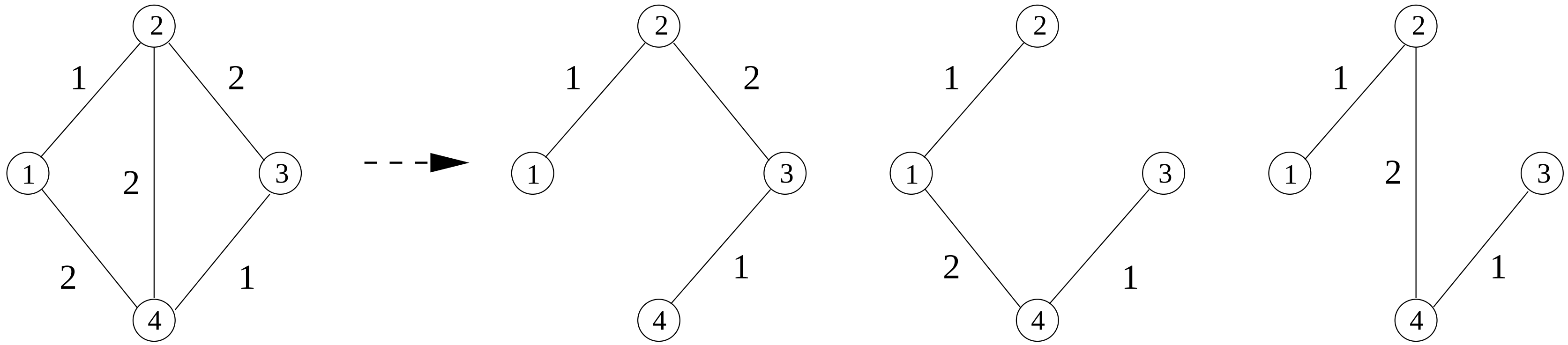
spanning-tree(G)
valitaan mielivaltainen solmu $v \in$ V
V'$= \{v\}$
T $= \emptyset$
while V' $\neq$ V
    valitse kaari $(u, v) \in E$ siten että $u \in V'$ ja $v \in V \setminus V'$
    V'$=$ V' $\cup \{v\}$
    T $=$ T $\cup \{(u, v)\}$

- Esimerkki algoritmin toiminnasta:



- Virittäviä puita hyödynnetään useissa sovelluksissa
- Esim. tietokoneverkkoprotokollissa ja hajautetuissa algoritmeissa koneet joutuvat usein jakamaan tietoa keskenään, tämä hoidetaan muodostamalla virittävä puu ja käyttämällä sitä sanomien välittämiseen
- Äsken esitetty algoritmi muodostaa verkosta jonkin virittävän puun. Jos kyseessä on painotettu verkko, olemme yleensä kiinnostuneita pienimmän virittävän puun muodostamisesta
- Olkoon $G = (V, E)$ suuntaamaton yhtenäinen painotettu verkko, jonka kaaripainot määrää funktio w
- Verkon G pienin (tai minimaalinen) virittävä puu (engl. Minimum Spanning Tree, MST) on G:n virittävistä puista se jonka kaaripainojen summa on pienin

- Yhdellä verkolla voi olla useita pienimpiä virittäviä puita:



- Yleensä riittää että pystytään muodostamaan jokin minimaalisista virittävistä puista

- Pienimmän virittävän puun muodostamiseen esitetään kurssilla kaksi algoritmia:
  - **Primin algoritmi**
  - **Kruskalin algoritmi**

- Yleisperiaate on sama kuin aikaisemmin esitetyssä algoritmissa jonkun virittävän puun laskemiseen: puuhun lisätään kaaria yksi kerrallaan niin, että ei synny sykliä

- Nyt kuitenkin kaaria lisätään puuhun määrätyssä järjestyksessä jotta virittävästä puusta saadaan pienin

Primin algoritmi [Vojtech Jarník, 1930, Robert C. Prim keksi uudestaan 1957 ja Edsger Dijkstra keksi uudestaan 1959]

- Algoritmin toimintaperiaate on sivun 553 algoritmin mukainen
  - rakenteilla on virittävä puu, jota kasvatetaan solmu kerrallaan
  - kun kaikki solmut on lisätty, on virittävä puu valmis
- Puuhun lisättäviä solmuja ei valita mielivaltaisesti
- Lisättäväksi solmuksi valitaan se, joka on lähimpänä jotakin jo puussa olevaa solmua
- Tälläisen solmun ja sen puuhun yhdistävän kaaren valinta näyttää aina suoritushetkellä parhaalta valinnalta
- Aina tietyllä hetkellä parhaalta näyttävän valinnan tekemistä sanotaan ahneeksi toimintastrategiaksi (engl. greedy), ja ahneita valintoja tekevää algoritmia ahneeksi algoritmiksi
- Ei ole selvää, että ahne valinta tuottaa aina optimaalisen lopputuloksen. Kuten pian näemme **Primin algoritmin** kohdalla tilanne on kuitenkin näin
- Seuraavalla sivulla algoritmin hahmotelma

- Puun muodostaminen aloitetaan jostain verkon solmusta r. Tämä aloitussolmu voidaan valita vapaasti

- Algoritmi kerää pienimmän virittävän puun muodostavat kaaret joukkoon T

- Virittävän puun ulkopuolella olevista solmuista pidetään kirjaa joukossa H

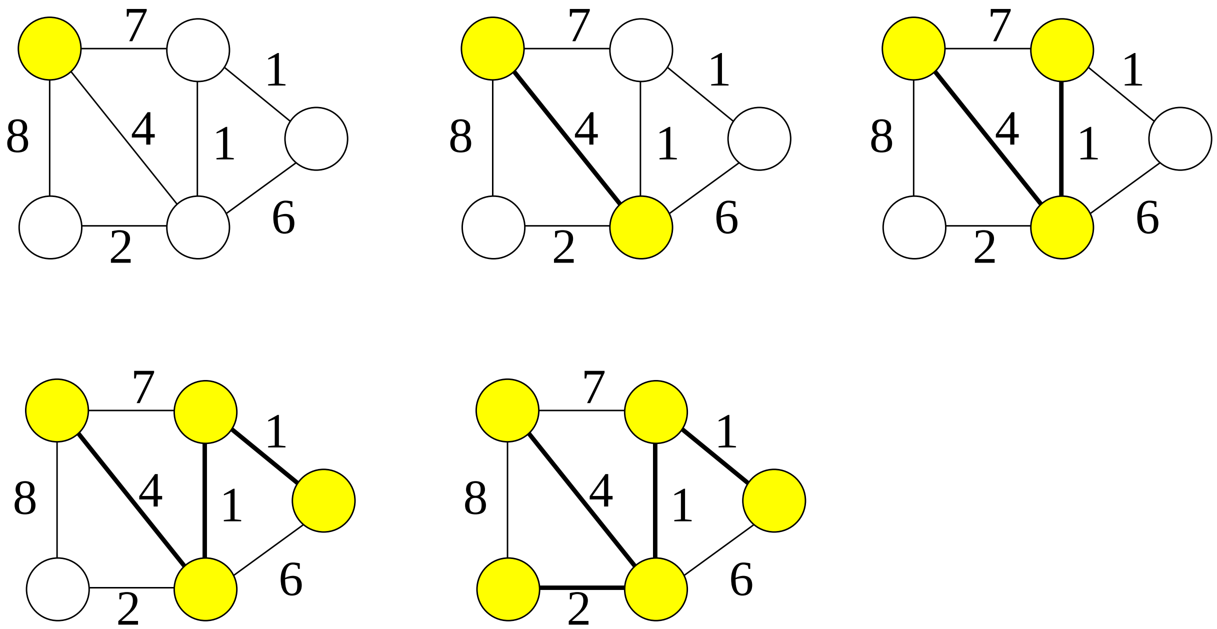
- Aluksi T on tyhjä ja H sisältää kaikki solmut

- Ensimmäisellä kierroksella r lisätään virittävään puuhun, eli poistetaan joukosta H

- Tämän jälkeen jokaisella kierroksella algoritmi lisää virittävään puuhun solmun, joka on lähimpänä jotain virittävässä puussa jo olevaa solmua
  - Lisättävä solmu on siis se, johon kohdistuu painoltaan pienin niistä kaarista, jotka yhdistävät jotain virittävään puuhun jo kuuluvaa ja jotain siihen kuulumatonta eli joukossa H olevaa solmua
  - uuden solmun virittävään puuhun yhdistävä kaari lisätään joukkoon T
  - lisätty solmu poistetaan joukosta H

- Kun joukko H on tyhjä, pienin virittävä puu on valmis ja se koostuu joukon T kaarista

- Ennen yksityiskohtaisempaa algoritmiesitystä tarkastellaan esimerkkiä
- Algoritmi aloittaa vasemman yläreunan solmusta, tummennetut kaaret muodostavat pienimmän virittävän puun



- Solmut siis liittyvät virittävään puuhun yksitellen, siten että jotain virittävässä puussa jo olevaa solmua lähimpänä oleva solmu liitetään puuhun kullakin kierroksella
- Algoritmin tehokkaan toteutuksen kannalta onkin oleellista että lähimpänä puun valmista osaa oleva solmu löytyy nopeasti

- Algoritmi käyttää aputaulukkoa $distance$, johon on talletettuna jokaiselle solmulle sen lyhin etäisyys johonkin jo virittävävässä puussa olevaan, eli joukon H ulkopuolella olevaan solmuun (eli ei etäisyyttä r:stä niin kuin Dijkstran algoritmissa)

- Aluksi asetetaan $distance[r] = 0$ ja kaikille muille solmuille $distance[v] = \infty$

- Jokaisella kierroksella lisätään puuhun se solmu u, jolla $distance[u]$ on pienin niistä solmuista, jotka kuuluvat joukkoon H

- Joukko H toteuteutetaan minimikekona siten, että avainkenttänä toimii $distance[v]$ eli solmun etäisyys johonkin virittävässä puussa jo olevaan solmuun
  - operaation **heap-del-min**(H) avulla saadaan siis selville nopeasti solmu, josta on lyhin kaari jo virittävässä puussa olevaan solmuun

- Kun virittävään puuhun lisätään uusi solmu u, käydään u:n vieruslista läpi
  - Jos vieruslistan solmu v on vielä keossa H ja $w(u, v) < distance[v]$, niin $distance[v]$-arvoa pienennetään $w(u, v)$:ksi
  - Näin siis solmun v pienin etäisyys jo virittävässä puussa olevaan solmuun saadaan pidettyä ajan tasalla
  - Jotta vierussolmu, jonka $distance$-arvo muuttuu, pysyisi keossa H oikealla paikalla, kutsutaan sille **heap-decrease-key**-operaatiota

- Algoritmilla on käytössä myös aputaulukko $parent$
- Jos solmu u ei vielä ole virittävässä puussa, kertoo $parent[u]$ sen virittävässä puussa olevan solmun, josta on lyhin kaari solmuun u
- Alussa asetetaan kaikille solmuille $parent[u] = NIL$, sillä virittävässä puussa ei ole vielä yhtään solmua
- Kun virittävään puuhun lisätään uusi solmu u
  - virittävään puuhun tulee kaari $(parent[u], u)$, eli se lisätään joukkoon T
  - käytäessä u:n vieruslistaa läpi

    jos vieruslistan solmu v on vielä keossa H ja $w(u, v) < distance[v]$, arvon $distance[v]$ pienennyksen lisäksi asetetaan $parent[v] = u$, sillä u on virittävän puun solmu, josta on lyhin kaari solmuun v

- **Primin algoritmi** seuraavalla sivulla
  - Algoritmin syötteenä on verkko G ja sen kaaripainot määrittelevä funktio w sekä solmu r josta virittävän puun muodostaminen aloitetaan
  - Algoritmi palauttaa joukon T joka sisältää virittävän puun kaaret

Prim(G,w,r)

```
T = ∅
for kaikille solmuille v ∈ V
    distance[v] = ∞
    parent[v] = NIL
distance[r] = 0
for kaikille solmuille v ∈ V
    heap-insert(H,v,distance[v])
while not empty(H)
    u = heap-del-min(H)
    if parent[u] ≠ NIL                        // ensimmäisen solmun lisäys ei tuo
        T = T ∪ { (parent[u],u ) }            // virittävään puuhun kaarta
    for jokaiselle solmulle v ∈ vierus[u]
        if solmu v on vielä keossa H ja w(u,v) < distance[v]
            parent[v] = u
            distance[v] = w(u,v)
            heap-decrease-key(H,v,distance[v])
return T
```

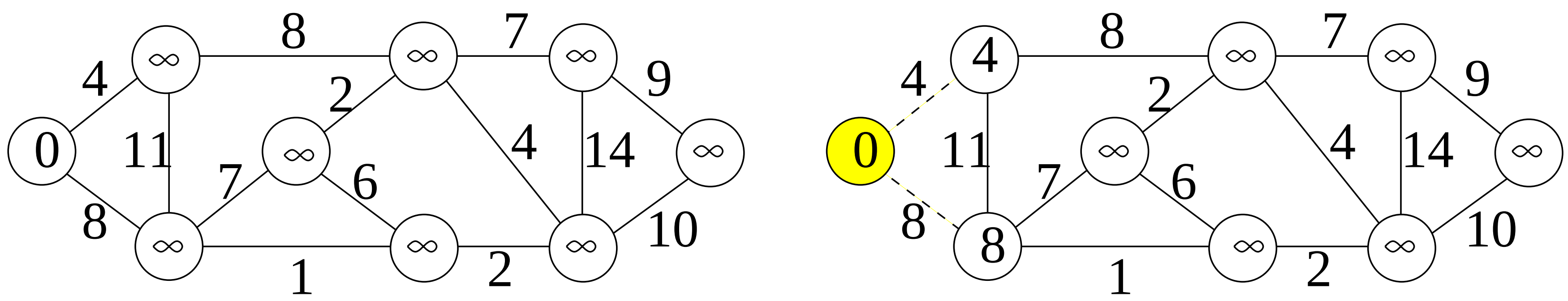
- Vaikka algoritmin toimintaperiaate onkin hyvin selkeä, mutkistavat toteutusyksityiskohdat algoritmia jossain määrin

- Algoritmin toimintaidea vielä kerran:
  - aluksi virittävä puu on tyhjä ja asetetaan kaikille solmuille paitsi r:lle etäisyyden $distance$ arvoksi ääretön (rivi 3)
  - myös lyhin kaari, joka yhdistää solmun virittävään puuhun on tässä vaiheessa tuntematon (rivi 4) kaikille solmuille
  - riveillä 6-7 solmut laitetaan minimikekoon H, avaimena siis solmun pienin etäisyys jostain virittävän puun solmusta joka on aluksi kaikille paitsi lähtösolmulle ääretön
  - ensimmäisessä toistossa tulee valituksi solmu r, joka siis poistuessaan keosta liittyy virittävään puuhun
  - kun virittävään puuhun liitetään solmu, pidetään voimassa ominaisuutta

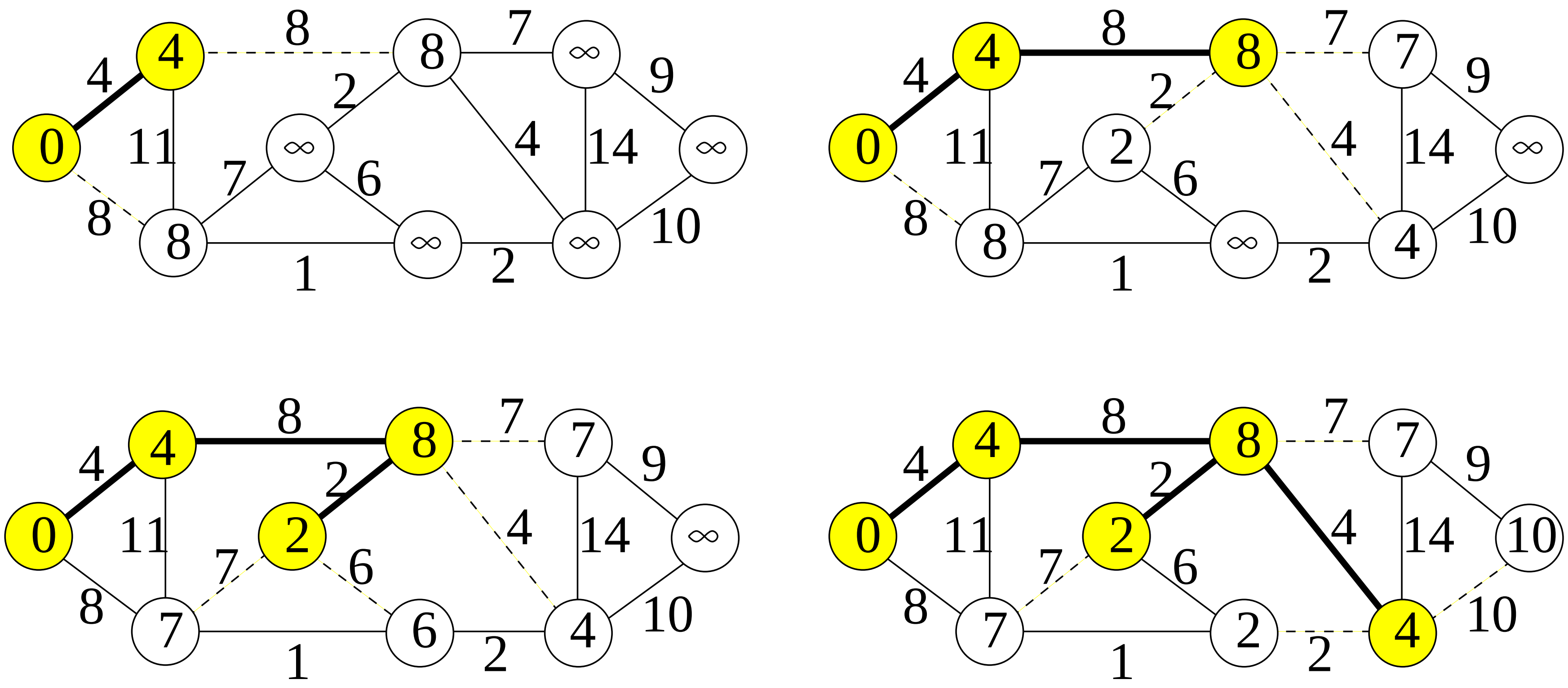
    jokaisen keossa olevan solmun v arvona $distance[v]$ on sen kaaren paino, minkä on kevyin niiden kaarten joukossa jotka yhdistävät v:n konstruktion alla olevaan virittävään puuhun;

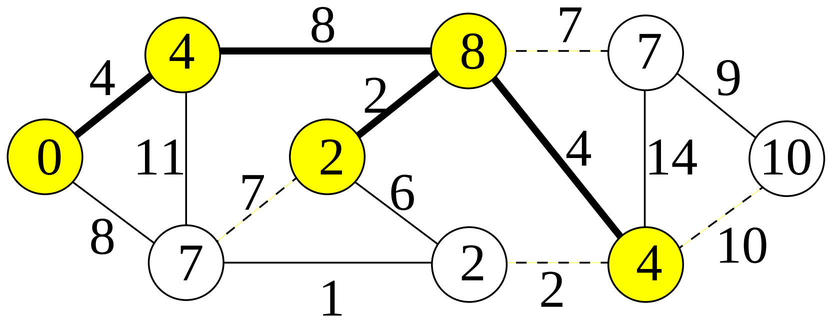
    kyseessä oleva kaari on $(parent[v], v)$
  - tämän jälkeen otetaan keosta käsittelyyn solmu jonka etäisyys virittävään puuhun on pienin ja liitetään solmu virittävään puuhun
  - riveillä 12-16 huolehditaan edelleen, että yllä oleva ominaisuus virittävän puun ulkopuolisten solmujen $distance$- ja $parent$-arvoille pysyy voimassa
  - jatketaan niin kauan kun solmuja on keossa jäljellä

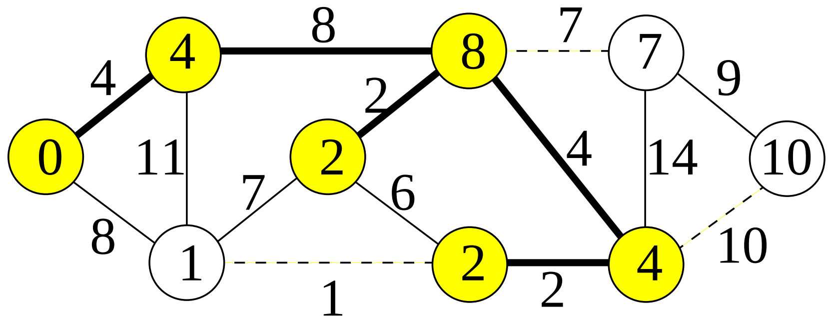
- Seuraavassa esimerkki algoritmin toiminnasta
- Taulukoiden *distance* ja *parent* informaatio on merkitty suoraan solmujen yhteyteen, myöskään keon H sisältöä ei eksplisiittisesti näytetä, kekoon kuuluvat kaikki värjäämättömät solmut
- Rivien 1-7 alustusvaiheen jälkeinen tilanne vasemmalla, eli kaikille paitsi aloitussolmulle on merkitty arvoksi $distance = \infty$
- Ensimmäisenä keosta poistetaan aloitussolmu, joka on värjätty oikeanpuoleisessa kuvassa, jossa on rivien 12-16 aikana suoritettujen keosta poistetun solmun vierussolmujen *distance*- ja *parent*- arvojen päivityksen jälkeinen tilanne
- *parent*-arvo, eli lyhin kaari virittävässä puussa olevaan solmuun on ilmaistu katkoviivallisena kaarena

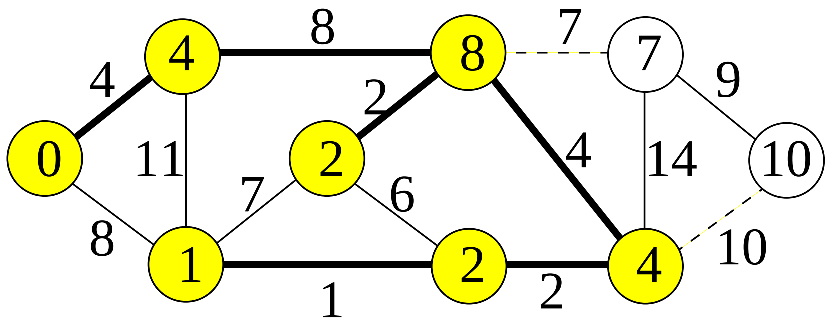


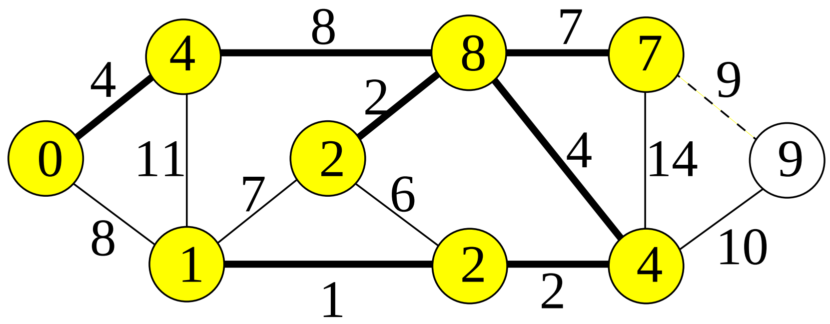
- Algoritmi jatkaa valitsemalla solmun, jonka $distance$-arvo on pienin
- Joukkoon T lisätään valitun solmun u rakenteilla olevaan puuhun yhdistävä kaari $(parent[u], u)$, joka on kuvassa merkitty tummennetulla
- Lisäyksen jälkeen päivitetään lisätyn solmun vierussolmujen $distance$- ja $parent$-arvot

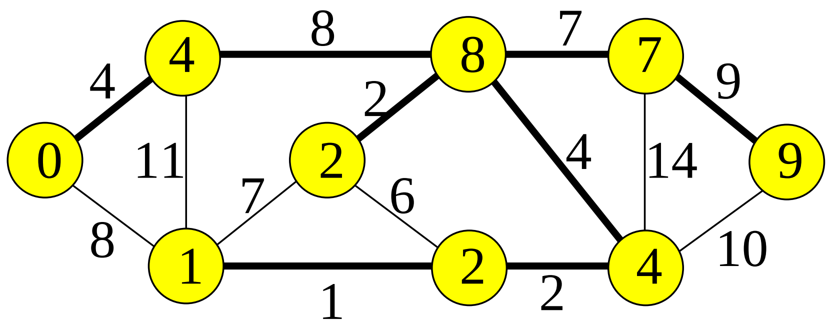



8 7
4 4 8 7 9
2
0 11 2 4 14 10
7 6
8 7 2 4 10
1 2


8 7
4 4 8 7 9
2
0 11 2 4 14 10
7 6
8 1 2 4 10
1 2


8 7
4 4 8 7 9
2
0 11 2 4 14 10
7 6
8 1 2 4 10
1 2


8 7
4 4 8 7 9
2
0 11 2 4 14 9
7 6
8 1 2 4 10
1 2


8 7
4 4 8 7 9
2
0 11 2 4 14 9
7 6
8 1 2 4 10
1 2

- **Primin algoritmin** aikavaativuus:
  - rivien 1-5 alkutoimet vievät aikaa $\mathcal{O}(|V|)$
  - algoritmi käyttää kekoa jossa pahimmillaan $|V|$ alkiota, kuten toivottavasti muistamme, kaikkien keko-operaatioiden aikavaatimus on $\mathcal{O}(\log|V|)$
  - riveillä 6-7 kutsutaan $|V|$ kertaa operaatiota **heap-insert**, eli aikaa kuluu $\mathcal{O}(|V|\log|V|)$; kuten Dijkstran algoritmissa, arvot ovat yksi nolla ja muut ääretön, joten tämä voitaisiin tehdä lineaarisessa ajassa
  - rivien 8-16 toistolauseessa operaatio **heap-del-min** suoritetaan kertaalleen jokaiselle solmulle, ja tähän kuluu aikaa $\mathcal{O}(|V|\log|V|)$
  - sisempi toistolause riveillä 12-16 suoritetaan $\mathcal{O}(|E|)$ kertaa sillä suuntaamattomassa verkossa kaikkien vieruslistojen yhteispituus on $2 \times |E|$
  - sisemmässä toistolauseessa suoritetaan **heap-decrease-key**-operaatio korkeintaan kertaalleen jokaisen kaaren yhteydessä, eli tästä koituva vaiva on $\mathcal{O}(|E|\log|V|)$
  - näin saamme **Primin algoritmin** aikavaativuudeksi $\mathcal{O}(|V|\log|V| + |E|\log|V|) = \mathcal{O}((|E| + |V|)\log|V|) = \mathcal{O}(|E|\log|V|)$ koska yhtenäisessä verkossa solmuja ei voi olla kuin korkeintaan yksi enemmän kuin kaaria
- Aputietorakenteena keko, ja alussa kaikki $|V|$ solmua ovat keossa eli tilavaativuus $\mathcal{O}(|V|)$

- Jos verkko on tiheä, eli $|E| = \Theta(|V|^2)$, voimme saada aikaan ajassa $\mathcal{O}(|V|^2)$ toimivan algoritmin

- Voimme etsiä solmun, jolla on pienin *distance*-arvo, lineaarisessa ajassa

```
Prim-Dense(G,w,r)
S = {r}
distance[r] = 0
for kaikille solmuille v ∈ V \ {r}
    distance[v] = w(r,v) // ääretön, jos kaari puuttuu
    parent[v] = r
while S ≠ V
    valitse u ∈ V \ S, jolla distance[u] on pienin
        T = T ∪ {(parent[u],u)}
    for jokaiselle solmulle v ∈ vierus[u]
        if v ∉ S and w(u,v) < distance[v]
            parent[v] = u
            distance[v] = w(u,v)
return T
```

Primin algoritmin oikeellisuus

- **Primin algoritmi** noudattaa ahnetta strategiaa: lisätään aina virittävään puuhun lähimpänä oleva puun ulkopuolinen solmu
- Ei ole aivan itsestään selvää että algoritmi tuottaa nimenomaan pienimmän virittävän puun
- Todistetaan, että **Primin algoritmi** todella tuottaa pienimmän virittävän puun
- Todistuksen idea:
  - Vaihdetaan joku algoritmin tuottaman virittävän puun kaari toiseksi
  - Osoitetaan, että näin saatavan virittävän puun paino ei voi olla pienempi kuin alkuperäisen virittävän puun paino

- Osoitetaan ensin seuraava aputulos, jonka avulla voidaan todistaa, että **Primin algoritmi** (ja seuraavaksi esitettävä **Kruskalin algoritmi**) tuottaa pienimmän virittävän puun

- **Lemma 8.8.:** Oletetaan suuntaamattomasta yhtenäisestä verkosta $G = (V, E)$, että

  1. sen solmut on jaettu kahteen epätyhjään osajoukkoon S ja $V \setminus S$

  2. kaari (u, v) kulkee osajoukosta toiseen eli $u \in S$ ja $v \in V \setminus S$

  3. kaari (u, v) on painoltaan pienin kaikista tällaisista osajoukkojen S ja $V \setminus S$ välisistä kaarista

  4. kaarijoukko $F \subseteq T$, jollekin pienimmälle virittävälle puulle (V, T)

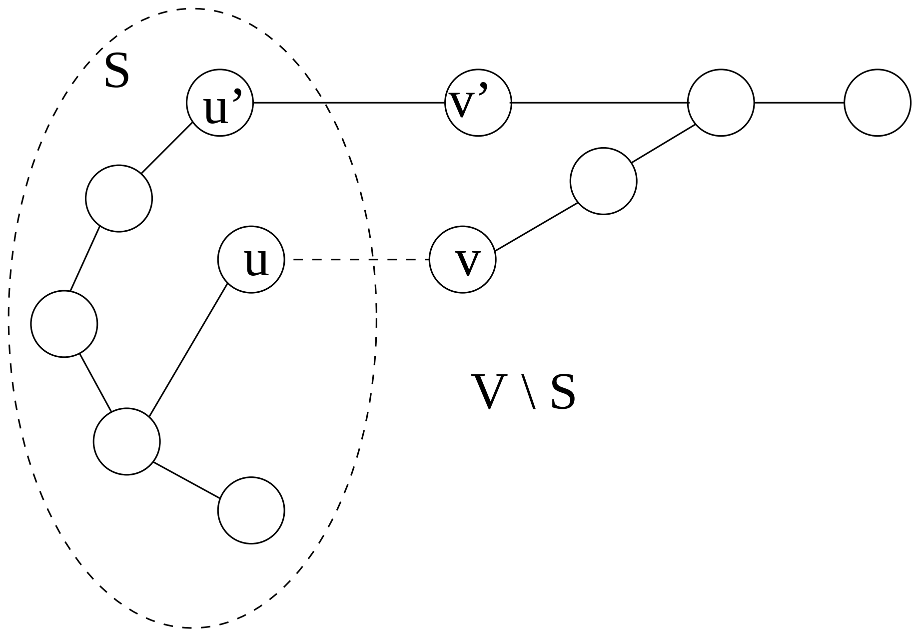
  5. ei ole kaaria F:ssä, jotka kulkevat osajoukosta toiseen, eli S:n ja $V \setminus S$:n välillä

  Nyt $F \cup \{(u, v)\} \subseteq T'$ jollekin pienimmälle virittävälle puulle (V, T').

- **Todistus:** Jos $(u, v) \in T$, niin väite on selvä. Oletamme nyt, että (u, v) ei kuulu väitteessä mainittuun pienimpään virittävään puuhun T.

  Nyt verkossa $(V, T \cup \{(u, v)\})$ on sykli, joka sisältää kaaren (u, v). Koska tämä kaari kulkee joukkojen S ja $V \setminus S$ välillä, syklillä on oltava toinen kaari (u', v'), joka myös kulkee joukkojen S ja $V \setminus S$ välillä. Koska F:ssä ei ole kaaria osajoukkojen välillä, tiedämme että $(u', v') \notin F$.

Poistetaan T:stä (u', v') ja lisätään (u, v). Lopputulos on virittävä puu (V, T'), missä $T' = T \cup \{(u, v)\} \setminus \{(u', v')\}$. Koska (u, v) oli pienin kaari joukkojen S ja $V \setminus S$ välillä, tiedämme, että $w(u, v) \leq w(u', v')$.



Uuden puun painoksi saamme siis

$$\sum_{e \in T'} w(e) = \sum_{e \in T} w(e) + w(u, v) - w(u', v') \leq \sum_{e \in T} w(e).$$

Mutta T oli pienin virittävä puu, joten myös T':n on oltava, eli (u, v) kuuluu pienimpään virittävään puuhun T'. $\square$

- Huomautus: Siis $w(u, v) = w(u', v')$

- **Primin algoritmi** on helppo osoittaa oikeaksi Lemmaan 8.8 vedoten
- Tarkastellaan sitä hetkeä, kun algoritmi päättää lisätä kaaren $e = (u, v)$ joukkoon T
- Valitaan nyt joukoksi S ne solmut, jotka on jo poistettu keosta H
- Kaari e on nyt kevyin kaari joukkojen S ja $V \setminus S$ välillä, eli sen täytyy lemman perusteella kuulua johonkin minimaaliseen virittävään puuhun
- Jokaisen algoritmin valitseman kaaren siis täytyy kuulua verkon minimaaliseen virittävään puuhun
- Algoritmi pitää huolen siitä, että T pysyy koko ajan puuna ja lopuksi se kattaa kaikki verkon G solmut
- Huomautus: Jos on samanpainoisia kaaria, pienin virittävä puu ei ole välttämättä yksikäsitteinen

Kruskalin algoritmi [Joseph Kruskal, 1956]

- Algoritmin suorituksen aikana pidetään yllä ns. virittävää metsä (erillisiä puita), joita vähitellen yhdistellään
- Jotta emme menisi sekaisin liiallisista puista, puhumme tässä paloista
- Aluksi verkon jokainen solmu on oma palansa eikä paloihin kuulu lainkaan kaaria
- Jokaisessa vaiheessa valitaan painoltaan pienin kaari, joka yhdistää kaksi tähän asti erillistä palaa. Näin palojen määrä pienenee yhdellä ja valittu kaari kuuluu pienimpään virittävään puuhun
- Kun jäljellä on vain yksi pala, on se verkon pienin virittävä puu
- Kaaret järjestetään ensin painonsa mukaiseen kasvavaan järjestykseen
- Jokaisella kierroksella tutkitaan järjestyksessä seuraavaa kaarta (u, v):

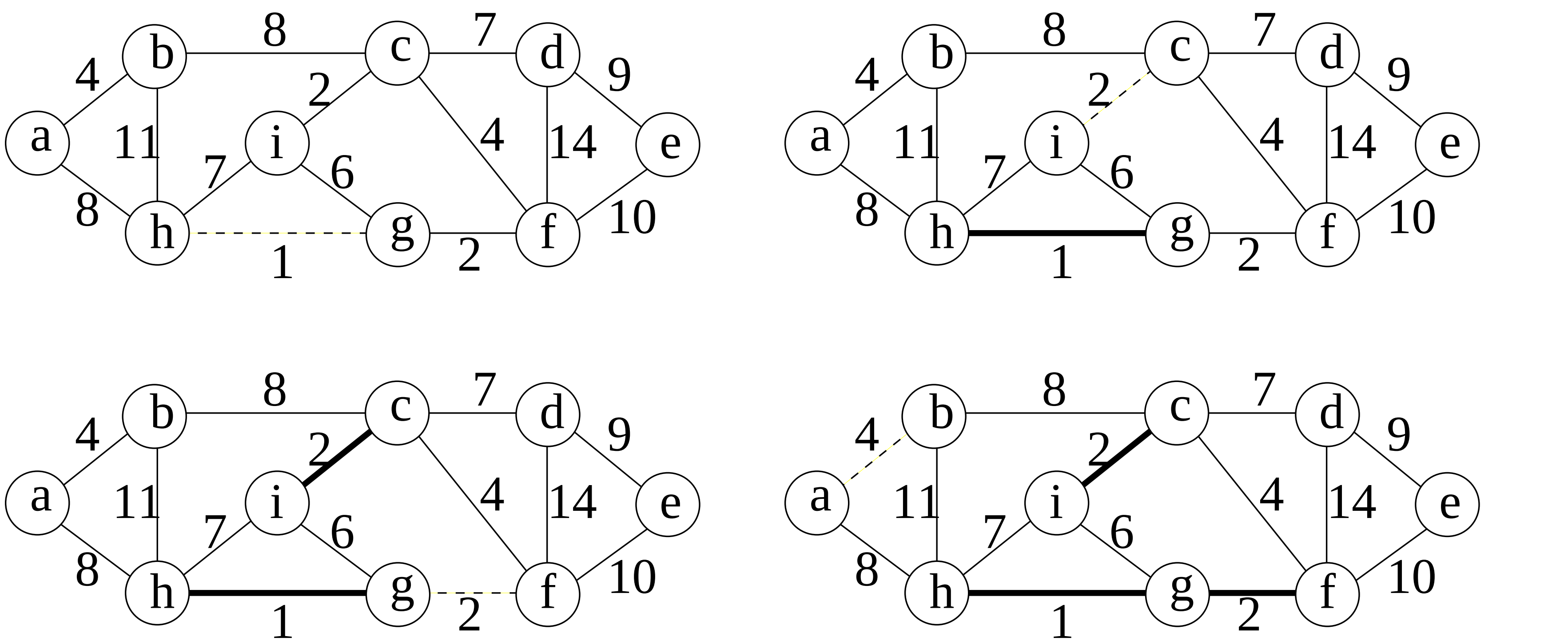
  jos solmut u ja v kuuluvat tällä hetkellä eri paloihin, kaari (u, v) otetaan mukaan minimaaliseen virittävään puuhun ja u:n ja v:n sisältävät palat yhdistetään

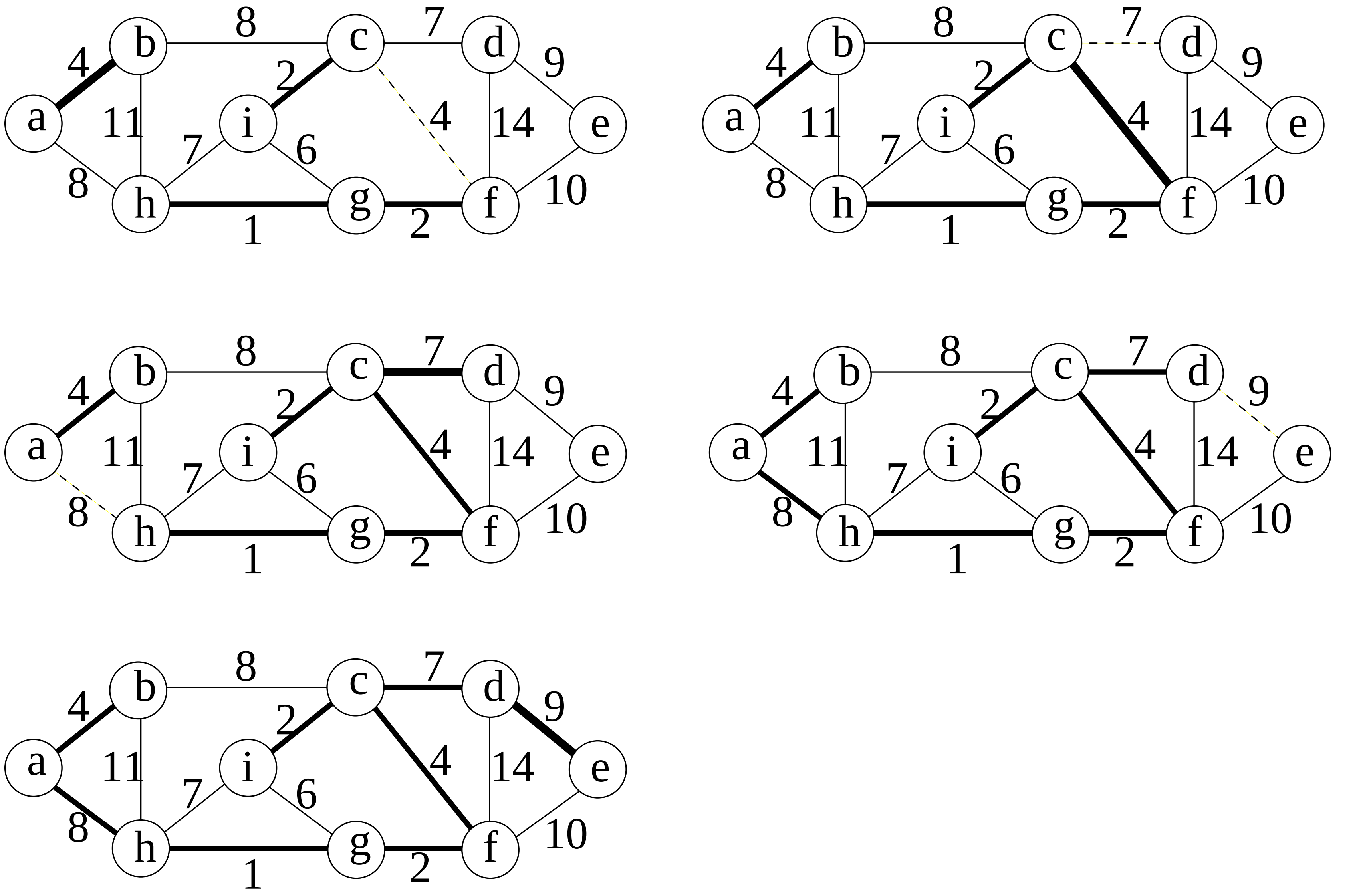
- Algoritmin ensimmäisessä versiossa tieto siitä mihin palaan solmut kuuluvat on talletettu taulukkoon $pala$
- Algoritmi kerää virittävän puun kaaria joukkoon T ja palauttaa lopuksi joukon
- Algoritmin lopussa verkon pienin virittävä puu siis koostuu joukon T kaarista

```
Kruskal(G,w)
T = ∅
for kaikille solmuille v ∈ V
    solmu v muodostaa oman palan pala[v]
järjestetään kaaret painon w mukaan kasvavaan järjestykseen
for jokaiselle kaarelle (u,v) ∈ E kasvavassa järjestyksessä
    if pala[u] ≠ pala[v]
        T = T ∪ {(u,v)}
        yhdistä pala[u] ja pala[v]
return T
```

- Esimerkki algoritmin toiminnasta seuraavalla sivulla:

- Aluksi jokainen solmu muodostaa oman palansa
- Paloja yhdistellään siten, että aina pyritään yhdistämään lähimpänä toisiaan olevat palat
  - algoritmi on järjestänyt kaaret kasvavaan pituusjärjestykseen joten kaarien järjestys määrää palojen yhdistelyjärjestyksen
- Paloja yhdistävä kaari (kuvassa paksunnettu) tulee osaksi minimaalista virittävää puuta




a
b
c
d
e
f
g
h
i
4
8
11
7
2
6
1
2
4
7
14
9
10

- Algoritmi itsessään on hyvin yksinkertainen mutta sen toteuttaminen tehokkaasti ei välttämättä ole aivan yksinkertaista
- Jotta toteutuksesta tulee tehokas, on erityisesti kiinnitettävä huomiota siihen miten solmuihin liittyvä palainformaatio toteutetaan
- Suoraviivainen tapa palainformaation tallettamiseen siis on käyttää $|V|$-paikkaista taulukkoa $pala$
  - oletetaan että palat on numeroitu, ja asetetaan alussa kunkin solmun palaksi oma luku
  - rivin 6 vertailu on nyt helppo ja hoituu järkevästi toteutettuna vakioajassa
  - rivillä 8 on yhdistettävä kaksi palaa yhdeksi
  - riittää kun asetetaan kaikille solmuille v minkä palanumerona on $pala[v]$ uudeksi palanumeroksi $pala[u]$
  - palat tallettava taulukko on siis käytävä läpi ja näin rivin 8 aikavaativuus on luokkaa $\mathcal{O}(|V|)$

- Koko algoritmin aikavaativuus:
  - oletetaan edellä kuvailtu suoraviivainen tapa toteuttaa palat
  - alkutoimet riveillä 1-3 vievät aikaa $\mathcal{O}(|V|)$
  - kaarten järjestäminen suuruusjärjestykseen, eli rivin 4 suorittaminen onnistuu ajassa $\mathcal{O}(|E| \log |E|)$
  - for-lause käy läpi kaikki $|E|$ kaarta
  - rivin 6 ehto toteutuu $|V| - 1$ kertaa, ja jokaisella näistä kerroista täytyy siis suorittaa $\mathcal{O}(|V|)$ vievä palat yhdistävä operaatio
  - for-silmukan kokonaissuoritusaika on siis $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$
  - saamme koko algoritmin aikavaativuudeksi $\mathcal{O}(|E| \log |E| + |V|^2)$
- Käyttämällä kohta esiteltävää union-find-tietorakennetta palojen toteuttamiseen, päästään algoritmissa aikavaativuuteen $\mathcal{O}(|E| \log |E|)$

Kruskalin algoritmin oikeaksi todistus

- Myös **Kruskalin algoritmi** noudattaa ahnetta strategiaa:
  - jokaisessa vaiheessa yhdistetään kaksi toisiaan lähimpänä olevaa palaa
  - eikä ole aivan itsestään selvää että algoritmi tuottaa nimenomaan pienimmän virittävän puun
- **Primin algoritmin** tapaan **Kruskalin algoritmin** oikeellisuus perustuu suoraan Lemmaan 8.8
- Tarkastellaan sitä hetkeä, kun algoritmi päättää lisätä kaaren $e = (u, v)$ tulosjoukkoonsa T
- Valitaan osajoukoksi S ne solmut jotka kuuluvat samaan palaan kuin solmu u, ja F on ne kaaret, jotka ovat jo T:ssä. Koska $pala[u] \neq pala[v]$, niin $v \notin S$
- Lemman perusteella $F \cup e$ on nyt osaa jotakin verkon G minimaalista virittävää puuta
- Algoritmin päättyessä T on puu ja siihen on lisätty vain sellaisia kaaria, jotka lemman perusteella kuuluvat minimaaliseen virittävään puuhun
- Kuten edellä **Primin algortimin** tapauksessa, jos on samanpainoisia kaaria, pienin virittävä puu ei ole välttämättä yksikäsitteinen

Union-find-tietorakenne

- Union-find-rakenne on tarkoitettu sovelluksiin, joissa seuraavat operaatiot on pystyttävä toteuttamaan tehokkaasti:
  - **make-set**(x): luo uusi yksialkioinen joukko
  - **find**(x): selvitä, mihin joukkoon parametrina annettu alkio x kuuluu
  - **union**(x, y): liitä yhteen kaksi joukkoa
- Joukot ovat erillisiä eli jokainen alkio kuuluu koko ajan tasan yhteen joukkoon
- Mikään joukko ei voi sisältää kahta tai useampaa samaa alkiota
- Joukon nimenä käytetään joukon edustajaa eli yhtä joukon alkiota. Joukon edustaja pysyy samana niin kauan kuin joukkoa ei muuteta

- Operaatioiden tarkempi määrittely:
  - **make-set**(x): luo uusi yksialkioinen joukko. Joukko sisältää alkion x ja sen nimeksi tulee x
  - **find**(x): selvitä, mihin joukkoon parametrina annettu alkio x kuuluu. Palauta tämän joukon nimi (ts. joukon edustaja)
  - **union**(x, y): liitä yhteen kaksi joukkoa, joiden nimet ovat x ja y. Operaatiota kutsutaan vain, jos $x \neq y$. Yhdistetyn joukon nimeksi tulee joko x tai y

  Kruskalin algoritmissa palat voidaan toteuttaa tällaisina joukkoina:
  - Aluksi luodaan oma joukko jokaista solmua varten
  - Kun halutaan selvittää, kuuluvatko solmut u ja v eri paloihin, tutkitaan, onko **find**(u) $\neq$ **find**(v)
  - Palat $pala[u]$ ja $pala[v]$ yhdistetään operaatiolla **union**(**find**(u), **find**(v))

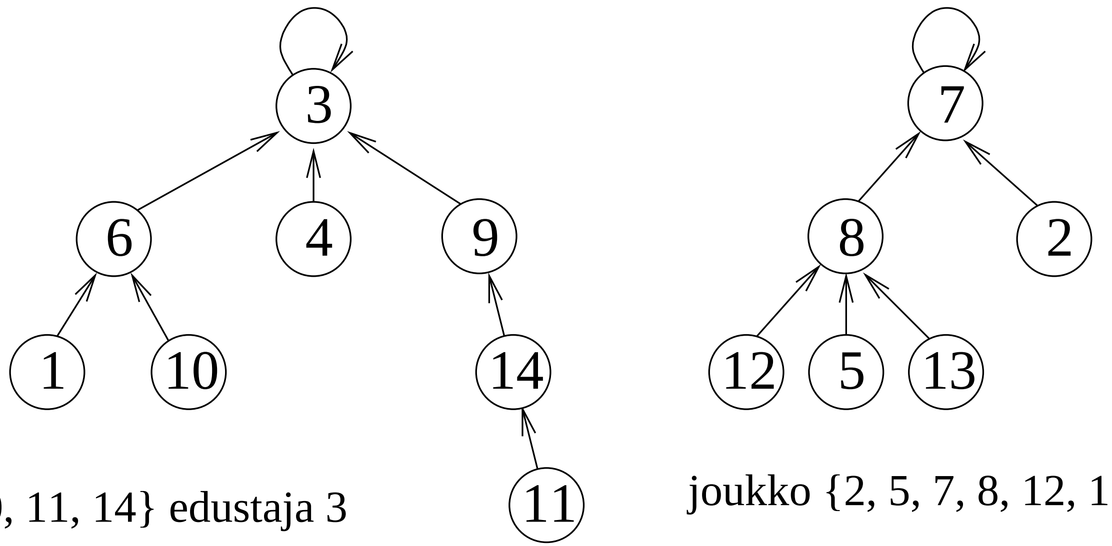
- **Kruskalin algoritmi** union-find-rakenteen avulla

```
Kruskal2(G,w)
T = ∅
for kaikille solmuille v ∈ V
    make-set(v)
järjestetään kaaret (u,v) ∈ E painon mukaan kasvavaan järjestykseen
while joukkojen lukumäärä > 1
    ota järjestyksessä seuraava kaari (u,v) ∈ E
    if find(u) ≠ find(v)
        T = T ∪ {(u,v)}
        union( find(u), find(v) )
return T
```

- Algoritmin tehokkuus perustuu siihen, että union-find-rakenne voidaan toteuttaa niin, että jono peräkkäisiä **union**- ja **find**- operaatiota voidaan suorittaa selvästi nopeammin kuin vastaavat operaatiot, jos palat toteutettaisiin taulukon avulla

Union-find-rakenteen toteutus

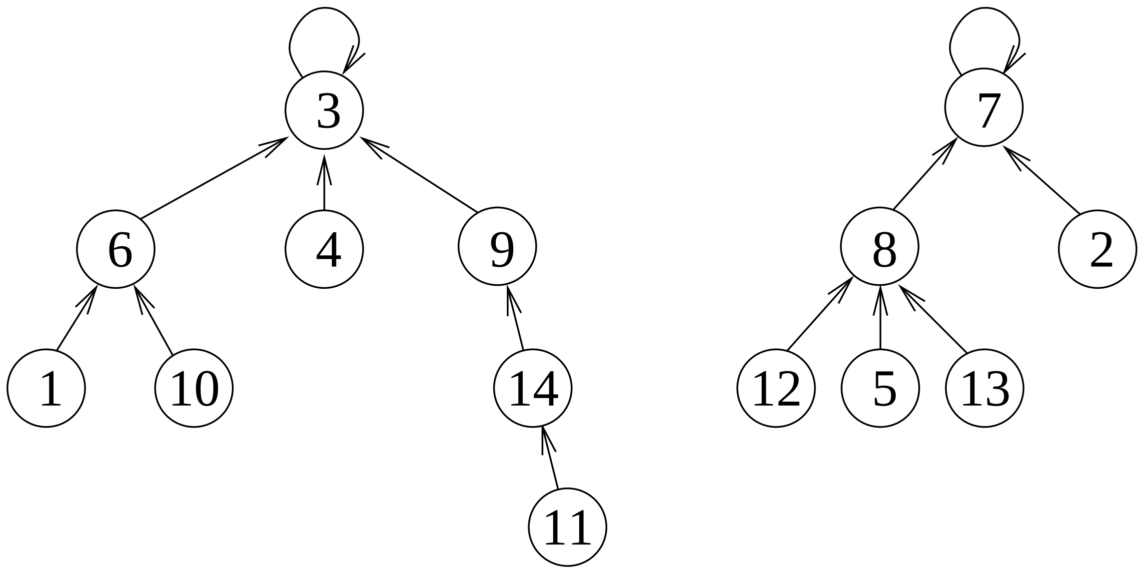
- Union-find-rakenne toteutetaan puiden avulla
- Jokaista joukkoa kuvaa yksi puu
- Puun juuressa on sen edustaja eli se alkio, joka antaa joukolle nimen
- Puussa jokaisesta solmusta on linkki sen vanhempaan, mutta ei lapsiin. Juuren vanhempi on juuri itse



joukko {1, 3, 4, 6, 9, 10, 11, 14} edustaja 3

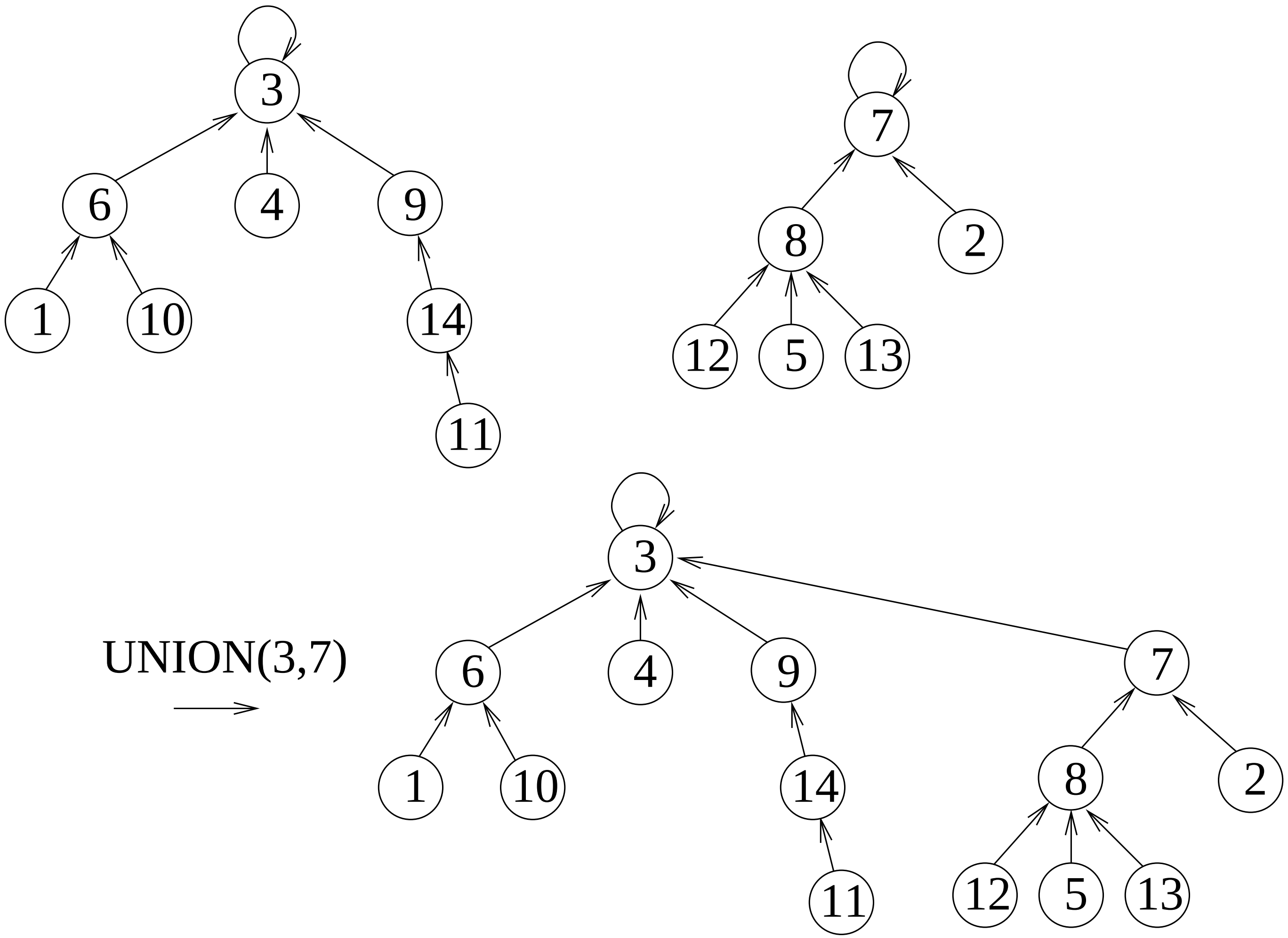
joukko {2, 5, 7, 8, 12, 13} edustaja 7

- Yksinkertaiset operaatiot: **find**
  - Operaatio **find**(x) lähtee solmusta x ja kulkee puussa ylöspäin nykyisen solmun vanhempaan niin kauan, kunnes tullaan juurisolmuun
  - Solmu x löydetään suoraan, sillä joukkoja kuvaavat puut on yleensä toteutettu taulukon avulla, missä kunkin indeksin arvona on tätä indeksiä vastaavan solmun vanhempi

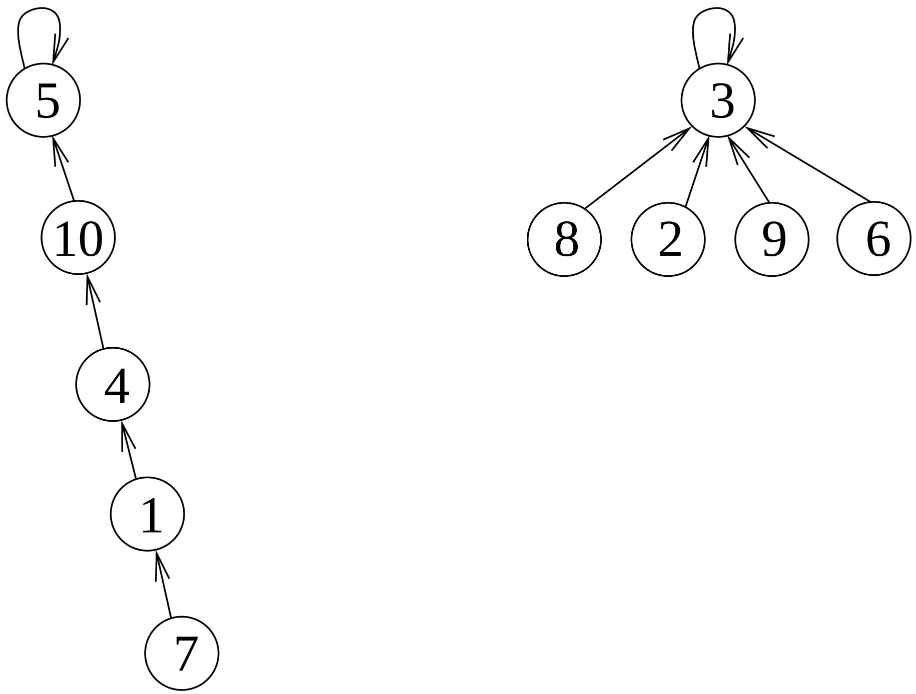


1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	7	3	3	8	3	7	7	3	6	14	8	8	9

- Yksinkertaiset operaatiot: **union**
  - Yhdistettävistä puista toisen juuri tulee toisen juuren lapseksi



- Puun pitäminen matalana
  - Operaatio **find**(x) kulkee koko matkan solmusta x joukkoa kuvaavan puun juureen saakka
  - **find**-operaation aikavaativuus on $\mathcal{O}(h)$, missä h on joukkoa kuvaavan puun korkeus
  - Pahimmassa tapauksessa puun jokaisella alkiolla on vain yksi lapsi, parhaassa tapauksessa taas kaikki puun muut solmut ovat juuren lapsia



- Jotta **find**-operaatiot pysyisivät tehokkaana, pyritään siihen, että joukkoja kuvaavat puut pysyisivät mahdollisimman matalina

- Tämä onnistuu muuttamalla **union**- ja **find**- operaatioita sopivasti

- Tapoja on useita, seuraavassa esitetään tapa, jolla pieni muutos **union**-operaatioon takaa joukkoa esittävien puiden korkeuden logaritmisuuden

- **Union**-operaatio suoritetaan aina siten, että matalamman puun juuri asetetaan korkeamman puun juuren lapseksi

- Näin yhdistetyn puun korkeus on sama kuin korkeamman puun ennen yhdistämistä, tai yksi korkeampi, jos alkuperäiset puut ovat yhtä korkeita

- Jotta **union**-operaatiota tehdessä voitaisiin päätellä, kumpi puu on korkeampi, on jokaisen puun juurisolmulla r attribuutti $r.korkeus$ joka on sen puun korkeus, jonka juurisolmu on r

- Induktiolla voidaan todistaa, että jos joukkoa kuvaavassa puussa on n solmua ja **union**-operaatiot on toteutettu edellisillä sivuilla kuvatulla tavalla, niin puun korkeus on $\mathcal{O}(\log n)$

- Näin m union-find -operaatiota voidaan suorittaa ajassa $\mathcal{O}(m \log n)$

- Tietorakennetta voidaan vielä tehostaa suorittamalla **find**-operaation yhteydessä poluntiivistys: Kun on löydetty polku puun juureen, niin oikaistaan kaikki osoittimet osoittamaan suoraan juureen

- Tällöin seuraavat **find**-operaatiot nopeutuvat

- Huomaa, että korkeus-arvoja ei päivitetä, joten ne menettävät tarkan vastaavuutensa polunpituuksiin

- Operaatioden koodi:

make-set(G,x)

```
x.p = x
x.korkeus = 0
```

union(x,y)

```
if x.korkeus < y.korkeus
    x.p = y
elsif x.korkeus > y.korkeus
    y.p = x
else
    x.p = y
    y.korkeus = x.korkeus + 1
```

find(x)

```
z = x
while z.p ≠ z
    z = z.p
y = z
z = x
while z.p ≠ z
    z = z.p
    z.p = y
return y
```

- Tasapainotusta ja poluntiivistystä käytettäessä voidaan osoittaa, että m union-find -operaatiota n alkion perusjoukossa vie vain ajan $\mathcal{O}(m\alpha(n))$, missä α on erittäin hitaasti kasvava funktion: karkeasti arvioiden $\alpha(n) \leq 4$, kun $n \leq 10^{80}$ (ks. Cormen luku 21.4.)

- Voidaan myös todistaa hiukan löysempi yläraja $\mathcal{O}(m \log_2^* n)$, missä $\log^* n$ määritellään seuraavasti:

$$\begin{aligned} \log^{(0)} n &= n \\ \log^{(i)} n &= \log(\log^{(i-1)} n), \text{ kun } \log^{(i-1)} n > 0 \\ \log^* n &= \min\{ i \geq 0 \mid log^{(i)} n \leq 1\}. \end{aligned}$$

- Esimerkiksi

$$\log_2^* 2^{65536} = \log_2^* 2^{2^{2^{2}}} = 5,$$

joten $\log_2^* n \leq 5$, kun $n \leq 2 \cdot 10^{19728}$

Union-find-rakenteen avulla toteutetun Kruskalin algoritmin aikavaativuus

- **Kruskalin algoritmissa** tarvittavat $\mathcal{O}(|E|)$ **find**-operaatiota voidaan suorittaa ajassa $\mathcal{O}(|E|\alpha(|V|))$

- **Union**- ja **make-set**- operaatiot voidaan suorittaa vakioajassa. **Kruskalin algoritmissa** molempia tarvitaan $\mathcal{O}(|V|)$-kappaletta

- Edellisillä sivuilla kuvatulla tavalla toteutettua union-find -rakennetta käyttävän **Kruskalin algoritmin** aikavaativuus on kaarten järjestämiseen käytettävä $\mathcal{O}(|E| \log |E|)$ + rivin 5-9 while-silmukkaan kuluva $\mathcal{O}(|E|\alpha(|V|))$ eli yhteensä $\mathcal{O}(|E|(\log |E| + \alpha(|V|)))$

- Koska yhtenäisessä verkossa solmuja ei voi olla kuin korkeintaan yksi enemmän kuin kaaria, sievenee aikavaativuus muotoon $\mathcal{O}(|E| \log |E|)$

- Kuitenkin $|E| \leq |V|^2$, joten

$$|E| \log |E| \leq |E| \log(|V|^2) = 2 \cdot |E| \log |V| = \mathcal{O}(|E| \log |V|)$$

- Aikavaativuus voidaan siis kirjoittaa muotoon $\mathcal{O}(|E| \log |V|)$

- Usein virittävä puu tulee valmiiksi, ennen kuin kaikki kaaret on käsitelty
- Tällaisissa tapauksissa algoritmia voidaan jonkin verran tehostaa käyttämällä kekoa, sen sijaan että heti järjestetään kaaret
- Pahimman tapauksen aikavaativuus on kuitenkin sama kuin ennen

Esimerkki virittävien puiden sovelluksesta

- Muita sovelluksia löytyy harjoitustehtävistä
- Halutaan osittaa verkon $G = (V, E)$ solmut kahteen luokkaan V_1 ja V_2 siten, että luokkien välinen pienin etäisyys

$$d(V_1, V_2) = \min \{ w(u, v) \mid u \in V_1, v \in V_2 \}$$

  on mahdollisimman suuri
- (Muistetaan, että osituksessa $V_1 \cap V_2 = \emptyset$ ja $V_1 \cup V_2 = V$)
- Intuitiivinen tulkinta on, että
  - $w(u, v)$ on jokin mitta alkioiden u ja v erilaisuudelle
  - alkiot halutaan jakaa kahteen mahdollisimman selvästi toisistaan erottuvaan luokkaan

- Annettu ongelma on erikoistapaus ryvästämisestä (engl. clustering), joka on tärkeä menetelmä data-analyysissä
- Halutaan osittaa perusjoukko X rypäisiin (engl. clusters) $X_1, \ldots, X_k$ siten, että
  - eri rypääseen kuuluvilla u ja v erilaisuus $d(u, v)$ on mahdollisimman suuri ja
  - samaan rypääseen kuuluvilla u ja v erilaisuus $d(u, v)$ on mahdollisimman pieni
- Tavoite voidaan täsmentää eri tavoin, mutta yleensä ongelma on laskennallisesti hankala
- Tässä tarkasteltu erikoistapaus ratkeaa kuitenkin helposti muodostamalla pienin virittävä puu
- Väitämme, että ongelma ratkeaa seuraavalla algoritmilla:
  1. Muodosta verkolle pienin virittävä puu (V, T)
  2. Olkoon e joukon T painoltaan suurin kaari
  3. Valitse luokiksi V_1 ja V_2 metsän $(V, T - \{ e \})$ yhtenäiset komponentit
- Tulos on sama, kuin jos ajettaisiin **Kruskalin algoritmia**, mutta lopetettaisiin juuri ennen viimeisen kaaren lisäämistä virittävään puuhun

- Todetaan ensin, että muodostettujen luokkien V_1 ja V_2 pienin etäisyys on

$$\min\{\, w(u,v) \mid u \in V_1, v \in V_2 \,\} = w(e)$$

- Valitaan mitkä tahansa $u \in V_1$ ja $v \in V_2$
- Nyt $T' = (T - \{\, e \,\}) \cup \{\, (u,v) \,\}$ on virittävä puu, jonka paino on

$$w(T') = w(T) - w(e) + w(u,v)$$

- Koska T on pienin virittävä puu, on oltava $w(u,v) \geq w(e)$
- Siis e on luokkia V_1 ja V_2 yhdistävistä kaarista painoltaan pienin
- Todetaan sitten, että millä tahansa solmujen osituksella (U_1, U_2) pätee

$$\min\{\, w(u,v) \mid u \in U_1, v \in U_2 \,\} \leq w(e)$$

- Tämä seuraa suoraan siitä, että ainakin yksi puun T kaari yhdistää joukkoja U_1 ja U_2, ja e on puun T kaarista painoltaan suurin
- Edellisistä kahdesta toteamuksesta seuraa, että algoritmin tuottama ositus (V_1, V_2) maksimoi lyhimmän etäisyyden

$$\min\{\, w(u,v) \mid u \in V_1, v \in V_2 \,\}$$

$\square$

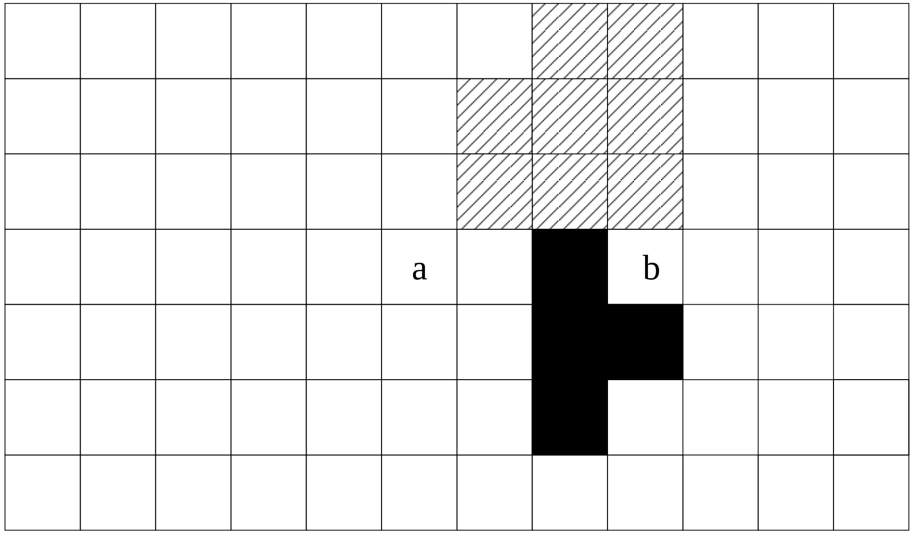
Kruskal vai Prim

- Voidaan osoittaa, että molemmat algoritmit toimivat negatiivisilla kaarenpainoilla
- Molempien algoritmien aikavaativuudet ovat $\mathcal{O}(|E| \log |V|)$
- Käytännössä **Prim** on yleensä parempi:
  - Avaimen arvon pienennystä tarvitaan yleensä vain pienelle osalle kaarista
  - **Kruskalin algoritmissa** järjestämisen keskimääräinen vaatimus ei ole pahinta tapausta parempi
  - Tiheissä verkoissa **Primin** aikavaativuus on $\mathcal{O}(|V|^2)$
- **Kruskal** tulee kuitenkin **Primiä** nopeammaksi, jos kaaret saadaankin valmiiksi järjestyksessä tai kaaret voidaan järjestää yleistä alarajaa nopeammin

Tähän loppuu tenttiin tulevien uusien asioiden esittäminen. Loppuosa on ylikurssia ja yhteenvetoa

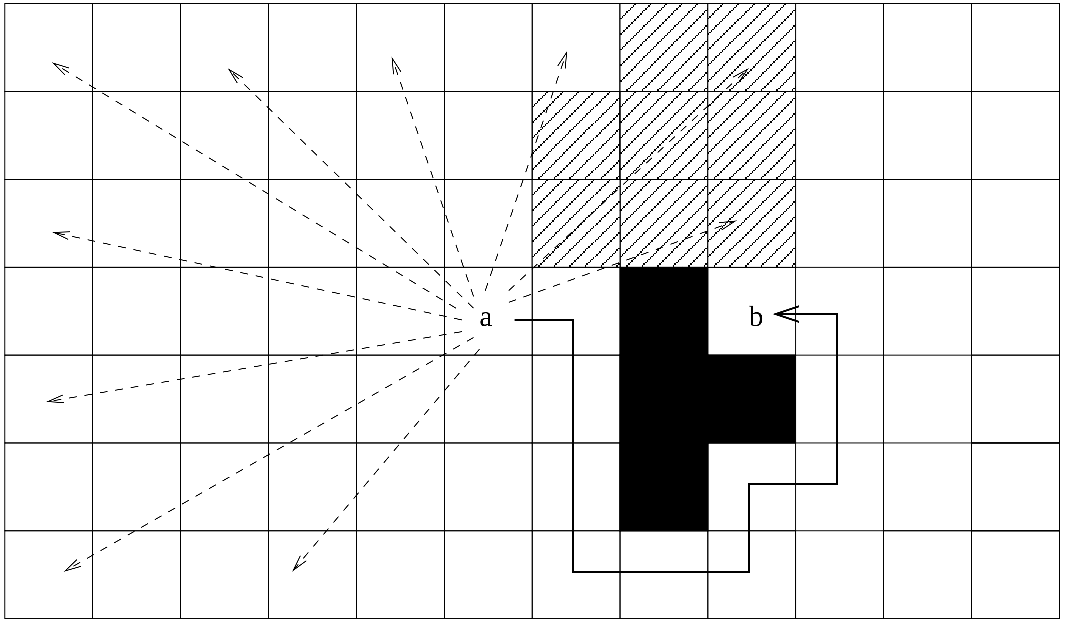
Lyhin kahden solmun välinen polku

- Haluamme etsiä lyhimmän polun alla olevan ruudukon kohdasta a kohtaan b
  - vierekkäisten (toistensa sivuilla, ylä- ja alapuolella olevien) valkoisten ruutujen välinen etäisyys on 1
  - mustien ruutujen läpi ei voi kulkea
  - raidalliset ruudut ovat vaikeakulkuista maastoa, niiden läpi voi kulkea, mutta "etäisyys" raidalliseen ruutuun on 5



- Tulkitaan ruudukko verkkona siten, että ruudut ovat solmuja ja vierekkäisiä ruutuja vastaavien solmujen välillä on kaari

- Lyhin polku voidaan selvittää **Dijkstran algoritmilla** (ks. sivut 520–534)
- Lyhin polku, jonka pituus on 11 on merkitty kuvaan



- **Dijkstran algoritmin** huono puoli on, että se ei huomioi millään tavalla, että kohdesolmu b on lähtöpaikan oikealla puolella
- **Dijkstra** etenee tasaisesti kaikkiin suuntiin ja tutkii samalla myös lyhimmät polut muihinkin solmuihin, myös täysin väärässä suunnassa oleviin
- Lyhimmän polun $a \rightsquigarrow b$ löytymisen kannalta **Dijkstra** siis tekee yleensä turhaa työtä

- **Dijkstran algoritmi** siis toimii seuraavasti:
  - ylläpidetään joukkoa S joka kohdistuu niistä solmuista joiden etäisyys lähtösolmuun s on selvitetty
  - jokaiselle solmulle v pidetään yllä etäisyysarviota $distance[v]$, joka kertoo lyhimmän tunnetun polun $s \leadsto v$ pituuden
  - alussa $distance[s] = 0$ ja muille solmuille $distance[v] = \infty$
  - aluksi S on tyhjä
  - algoritmi käsittelee solmuja yksi kerrallaan lisäten joukkoon S aina solmun jonka etäisyysarvio on pienin
  - kun solmu v tulee käsitellyksi tarkastetaan sen vierussolmujen etäisyysarviot

- Algoritmi lopettaa kun kaikki verkon solmut on lisätty joukkoon S eli tunnetaan pienin polku lähtösolmusta s kaikkiin muihin solmuihin

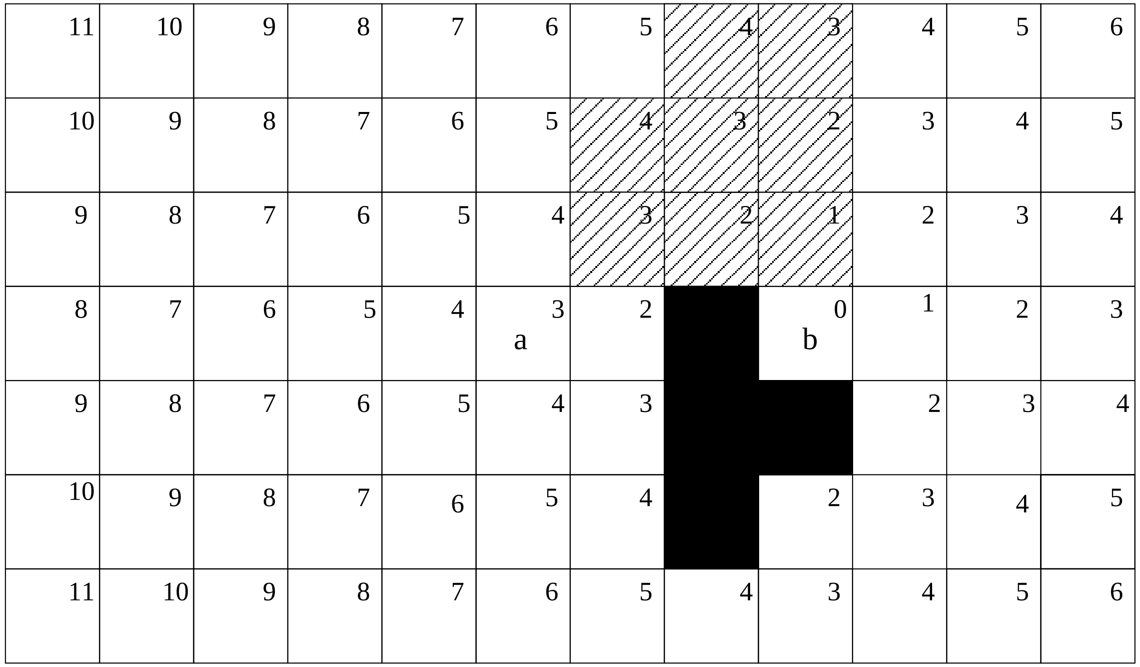
- Jos ollaan kiinnostuneita ainoastaan kahden solmun a ja b välisestä lyhimmästä polusta, tutkii **Dijkstra** siis samalla myös b:stä poispäin johtavia polkuja

- "Ohjaamalla" algoritmin toimintaa siten, että se suosii kohdesolmuun b päin johtavia polkuja, on yleensä mahdollista löytää lyhin polku nopeammin kuin **Dijkstran algoritmilla**

A*-algoritmi

- **A*-algoritmin** voi ajatella **Dijkstran algoritmin** laajennuksena, joka koko ajan arvioi mikä tutkimattomista solmuista näyttää olevan osa lyhintä polkua solmujen a ja b välillä
- Tätä varten algoritmi arvioi etäisyyden jokaisesta solmusta maalisolmuun b
- Arvio voidaan tehdä monella tavalla
- Seuraavalla sivulla olevaan kuvaan on merkitty etäisyysarviot, joiden arvo on lyhin mahdollinen etäisyys solmujen välillä huomioimatta millään tavalla esteitä tai vaikeakulkuista reittiä
- Arvion tulee olla helposti laskettavissa
  - esimerkkitapauksessamme kohdesolmu b on ruudussa $(4, 9)$ neljä alas, 9 oikealle
  - etäisyysarvio ruudukon kohdassa (i, j) olevasta solmusta kohdesolmuun b on $|(i - 4) + (j - 9)|$
  - esim. lähtösolmu on paikassa $(4, 6)$, joten sen etäisyysarvio on $|(4 - 4) + (6 - 9)| = | - 3| = 3$

- Etäisyysarviot jokaisesta solmusta kohdesolmuun b on merkitty ruudun oikeaan yläkulmaan



- Todellisuudessa arvioita ei tarvitse laskea kaikille solmuille etukäteen vaan riittää, että ne selvitetään tarpeen vaatiessa

- **Dijkstran algoritmin** tapaan jokaiselle solmulle ylläpidetään myös etäisyysarviota lähtösolmuun a

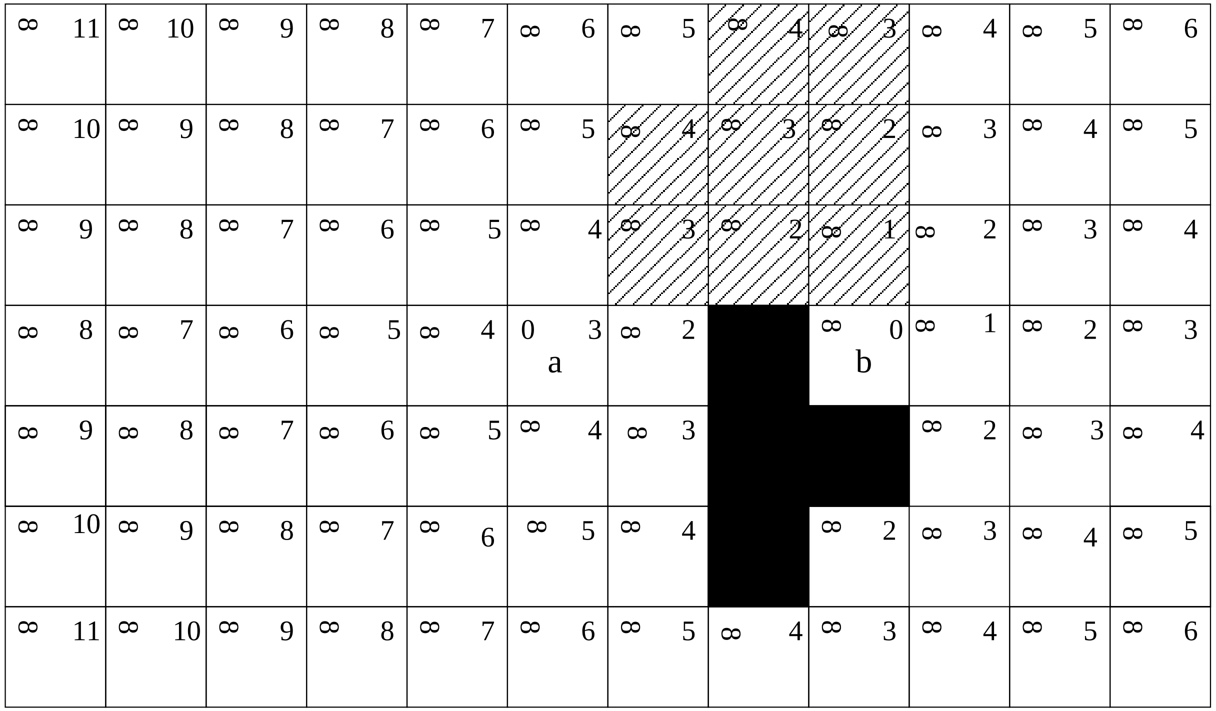
- **A*** siis ylläpitää jokaiselle solmulle v kahta tietoa
  - $alkuun[v]$: lähtösolmusta solmuun v johtavan polun $a \rightsquigarrow v$ etäisyysarvio (tähän mennessä tiedossa oleva lyhin etäisyys, kuten **Dijkstran algoritmissa**)
  - $loppuun[v]$: solmusta v maalisolmuun johtavan polun $v \rightsquigarrow b$ etäisyysarvio
- Algoritmi pitää kirjaa jo käsitellyistä solmuista joukon S avulla
  - alussa S on tyhjä
  - aluksi asetetaan kaikille solmuille v paitsi lähtösolmulle $alkuun[v] = \infty$ ja $alkuun[a] = 0$

    eli alussa etäisyysarvio lähtösolmusta muihin solmuihin on tuntematon
  - algoritmi käsittelee solmuja yksi kerrallaan lisäten joukkoon S aina solmun v jolle summa $alkuun(v) + loppuun(v)$ on pienin
  - kun solmu v tulee käsitellyksi, sen vierussolmujen etäisyysarviot lähtösolmuun päivitetään tarvittaessa (kuten **Dijkstran algoritmissa**)
- Algoritmi lopettaa kun se on käsitellyt kohdesolmun b
- **Dijkstran algoritmin** tapaan algoritmi muistaa kullekin solmulle mistä lyhin polku siihen saapui

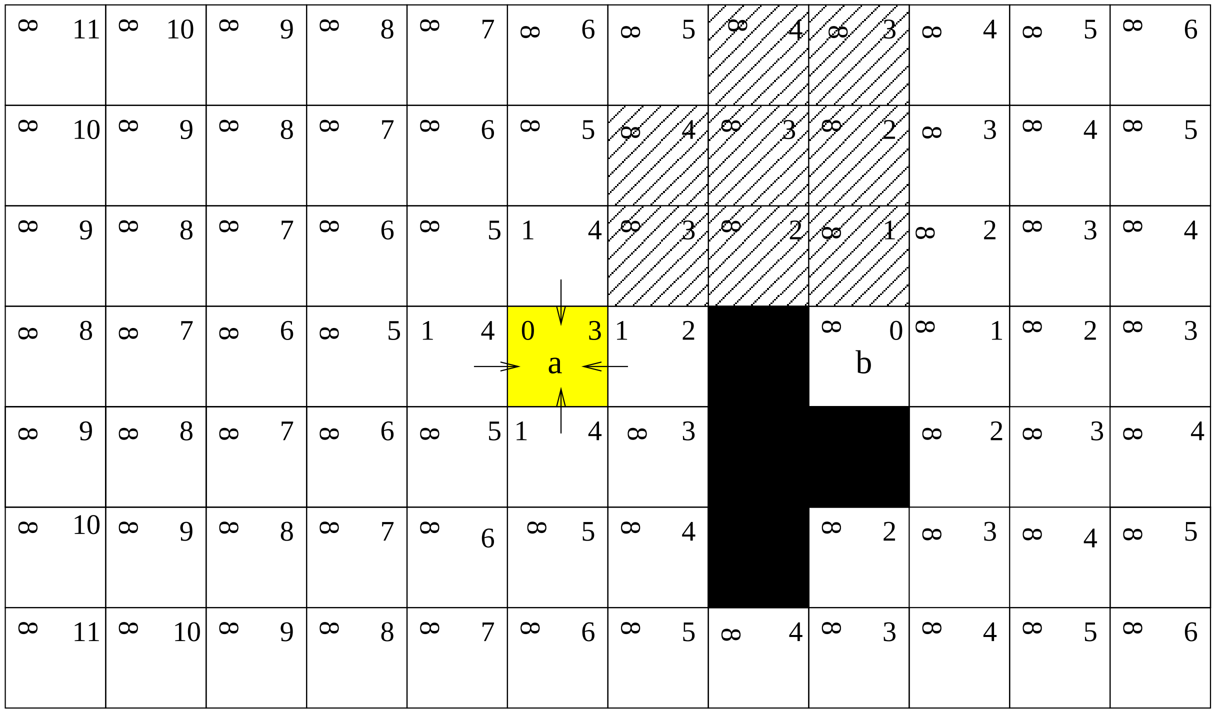
- Lähtötilanteessa kaikkien paitsi lähtösolmun etäisyysarvio $alkuun$ on ääretön

- Alussa käsiteltyjen solmujen joukko S on tyhjä

- Joukkoon S lisätään aina solmu v jolle summa $alkuun(v) + loppuun(v)$ on pienin, eli alussa lisättäväksi valitaan lähtösolmu



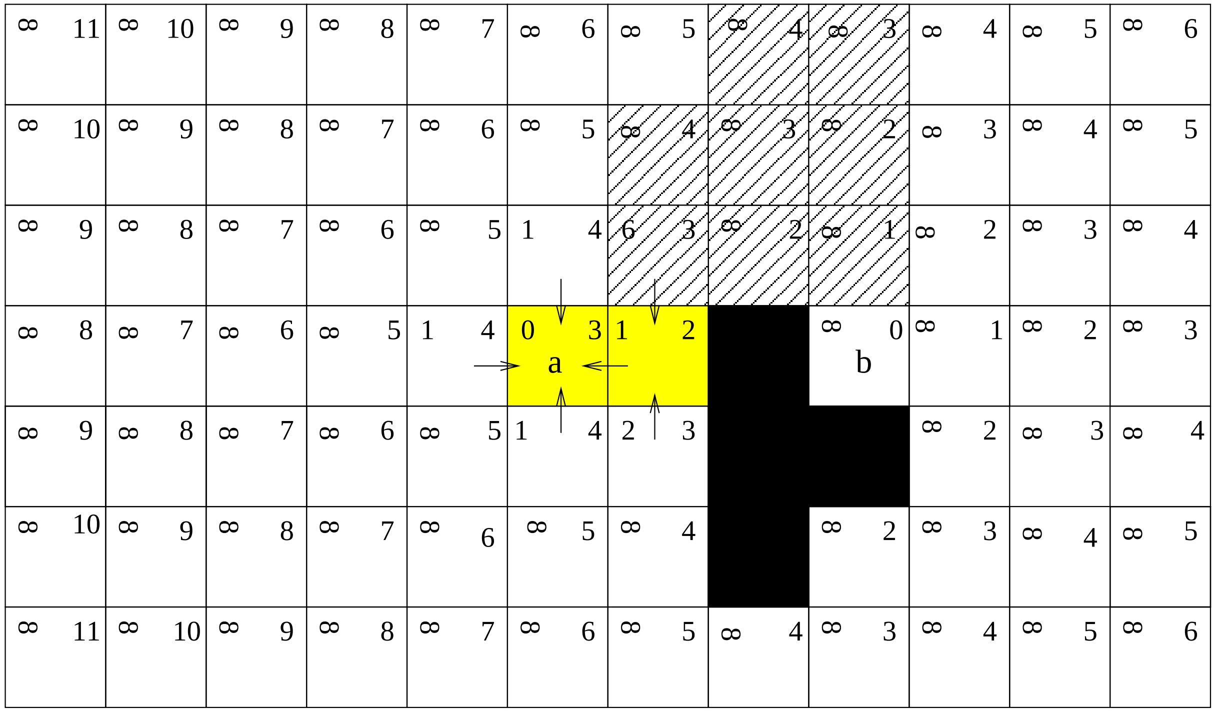
- Lähtösolmu on nyt käsitelty eli viety joukkoon S, kuvassa se on muutettu harmaaksi
- Lähtösolmun vierussolmujen v arvot $alkuun[v]$ on päivitetty
- Vierussolmuihin on myös merkitty että niihin saavuttiin alkusolmusta, tätä tietoa hyväksikäyttäen pystytään lopuksi generoimaan etsitty lyhin polku



- Jälleen käsittelyyn valitaan solmu v jolle summa $alkuun(v) + loppuun(v)$ on pienin, eli lähtösolmun oikealla puolella oleva solmu

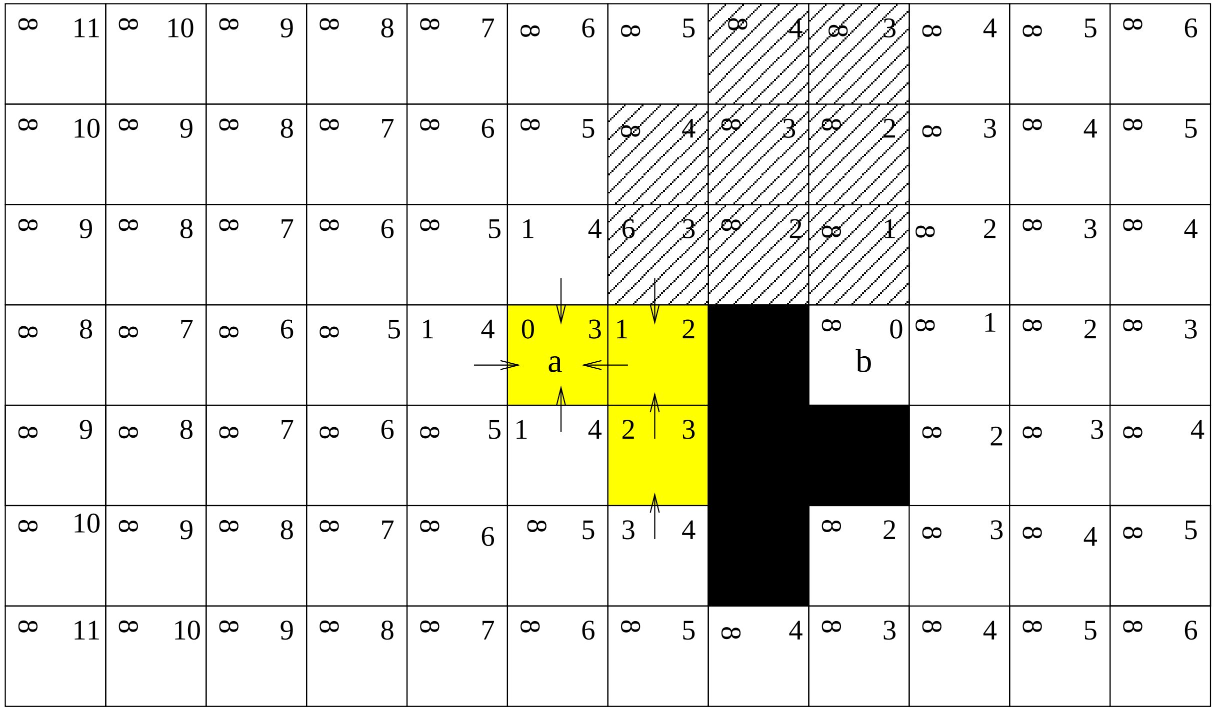
- Huom: valitun solmun yläpuolella on vaikeakulkuinen maasto, joten kaari yläpuolella olevaan solmuun on pituudeltaan 5

- Solmun käsittelyn jälkeen verkossa on 4 solmua joilla $alkuun(v) + loppuun(v) = 5$, eli algoritmi valitsee seuraavaksi käsittelyyn jonkun näistä



- Oletetaan, että algoritmi valitsee juuri käsitellyn solmun alapuolella oleva solmu seuraavaksi käsiteltäväksi

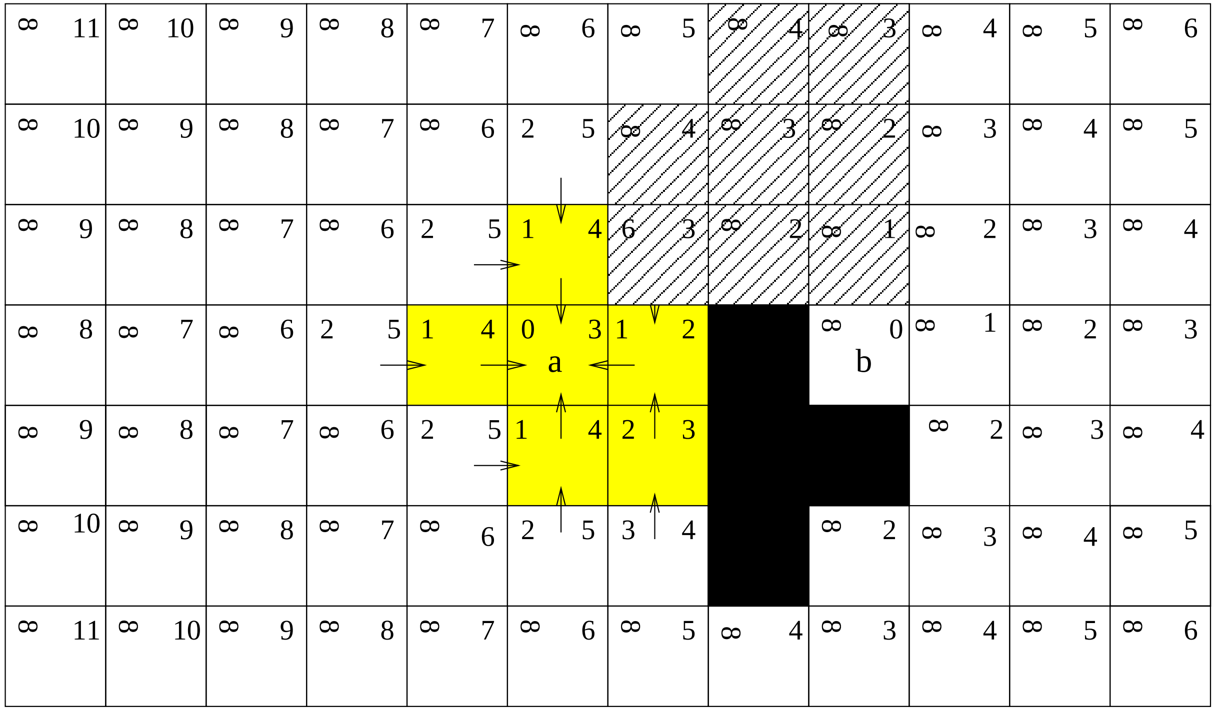
- Päivityksen jälkeen tilanne näyttää seuraavalta



- Jälleen käsittelyyn valitaan solmu v jolle summa $alkuun(v) + loppuun(v)$ on pienin

- Verkossa on edelleen 3 solmua joille $alkuun(v) + loppuun(v) = 5$, eli vaikka silmämääräisesti huomaamme, että ne ovat väärässä suunnassa, tutkii algoritmi ne ennen kuin lähtee oikeaan suuntaan

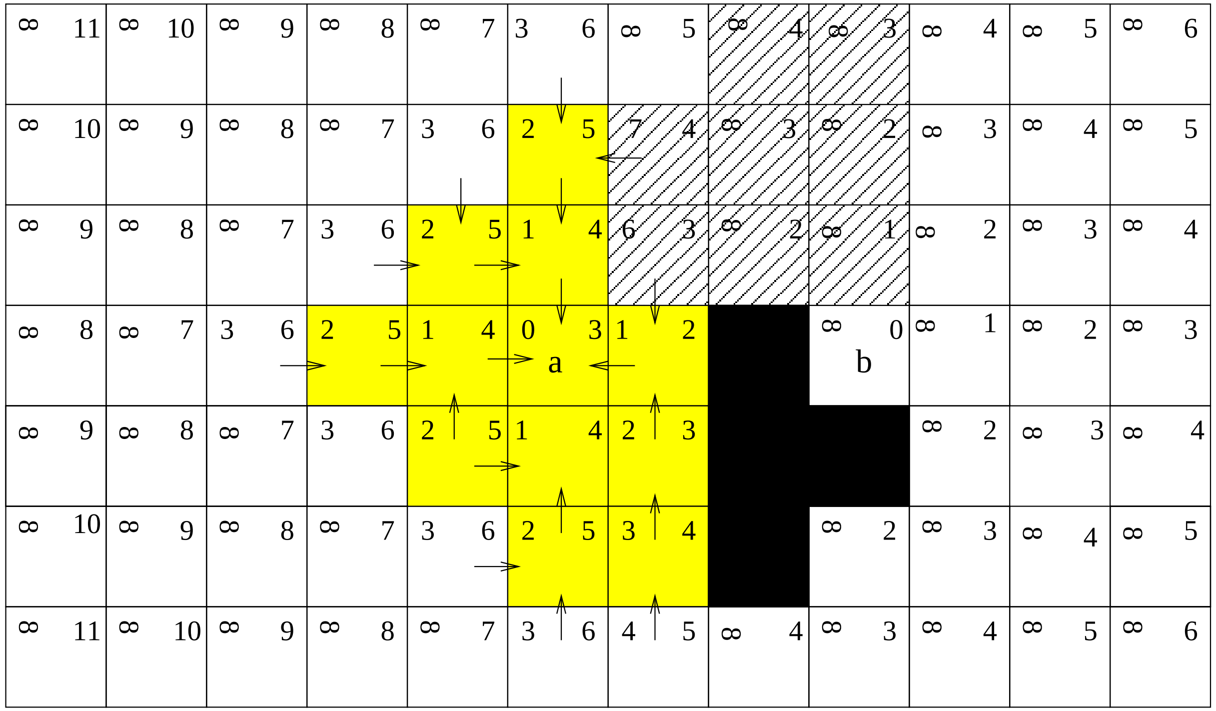
- Seuraavassa tilanne kolmen "väärässä suunnassa" olevan solmun tutkimisen jälkeen



- Verkossa on nyt 6 solmua joille $alkuun(v) + loppuun(v) = 7$

- Vaikka osa näistä on väärässä suunnassa, tutkii algoritmi ne seuraavissa askeleissa

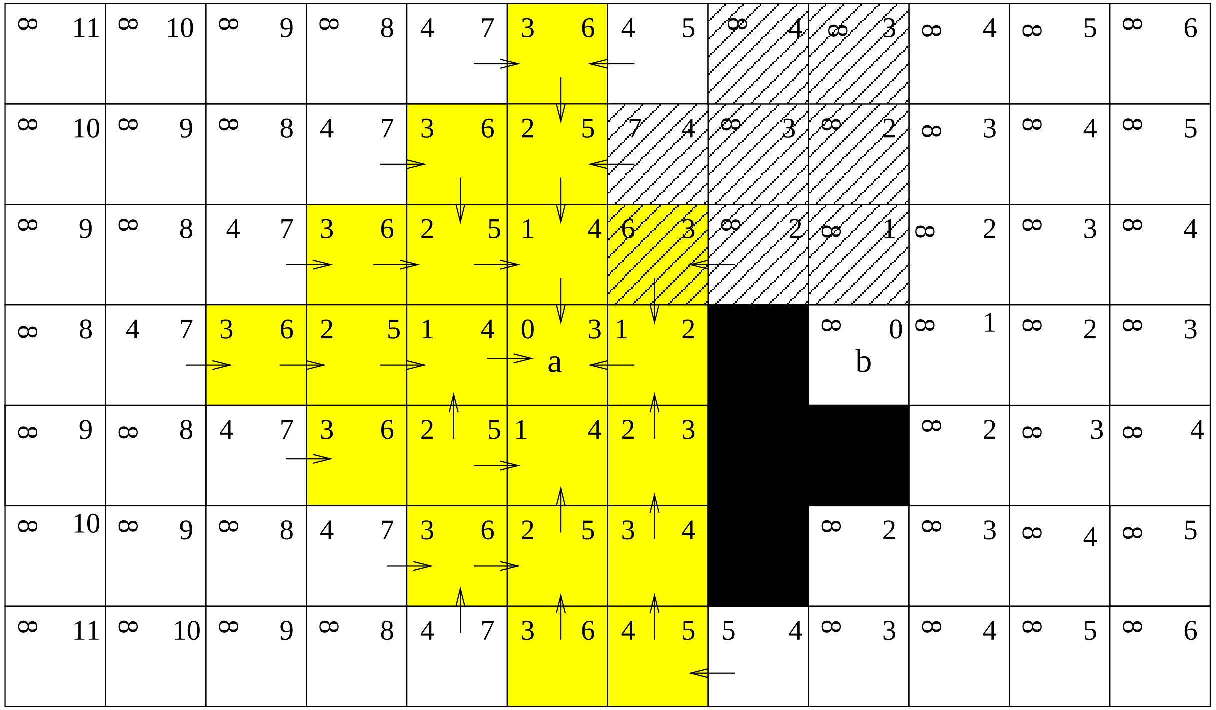
- Päädytään seuraavaan tilanteeseen



- Seuraavissa vaiheissa vuorossa olevat solmut, joille $alkuun(v) + loppuun(v) = 9$, näitä solmuja on 9

- Tutkitaan nämä solmut

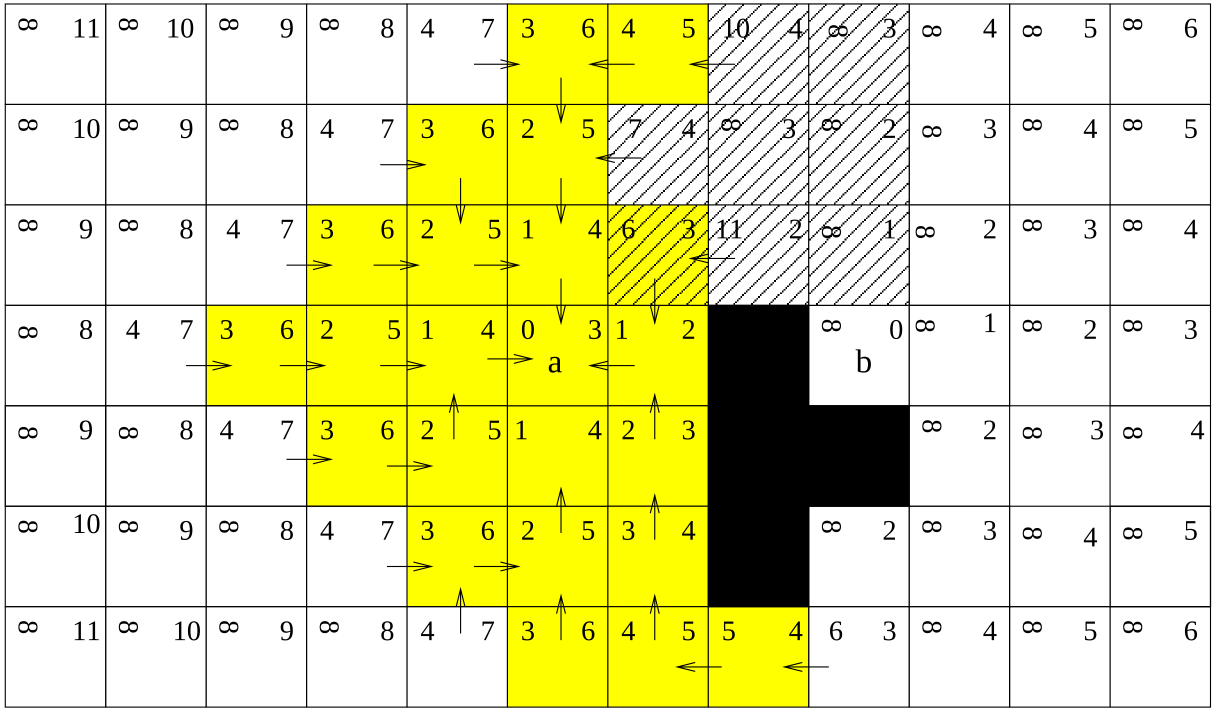
- Päädytään seuraavaan tilanteeseen



- Huomaamme, että algoritmi alkaa vihdoin kääntää etsintää oikeaan suuntaan sillä nyt "väärässä suunnassa" oleville solmuille $alkuun(v) + loppuun(v) = 11$

- Seuraavissa vaiheissa siis käsitellään ne 2 solmua, joille $alkuun(v) + loppuun(v) = 9$

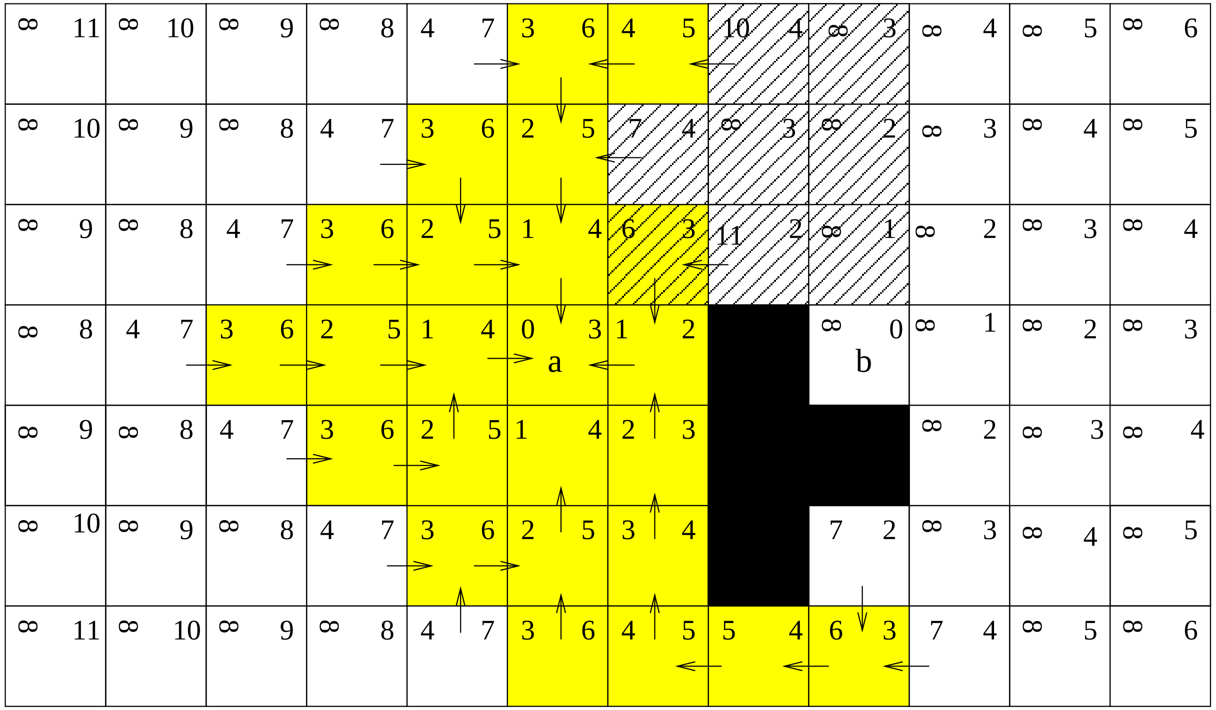
- Päädytään seuraavaan tilanteeseen



- Jälleen löytyy uusi solmu, jolle $alkuun(v) + loppuun(v) = 9$

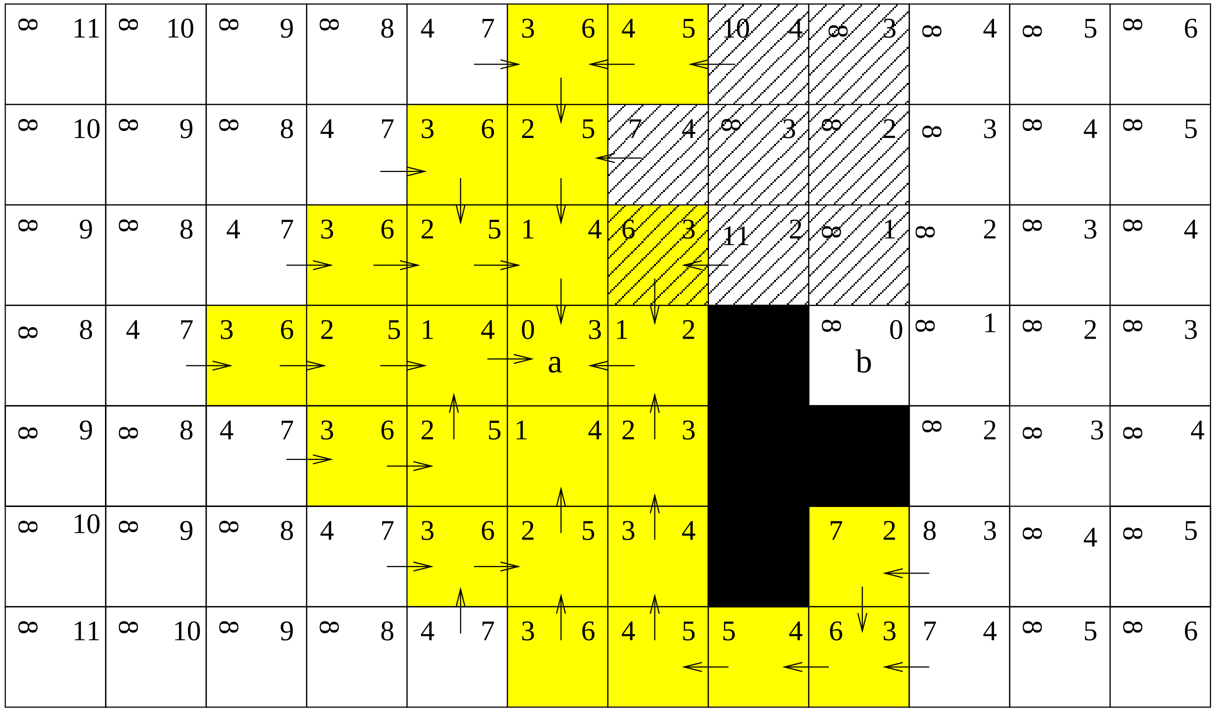
- Valituksi tuleva solmu on askeleen lähempänä maalisolmua, joten algoritmi jatkaa oikeaan suuntaan

- Päädytään seuraavaan tilanteeseen



- Jälleen löytyy uusi solmu, jolle $alkuun(v) + loppuun(v) = 9$
- Algoritmi valitsee solmun käsittelyyn

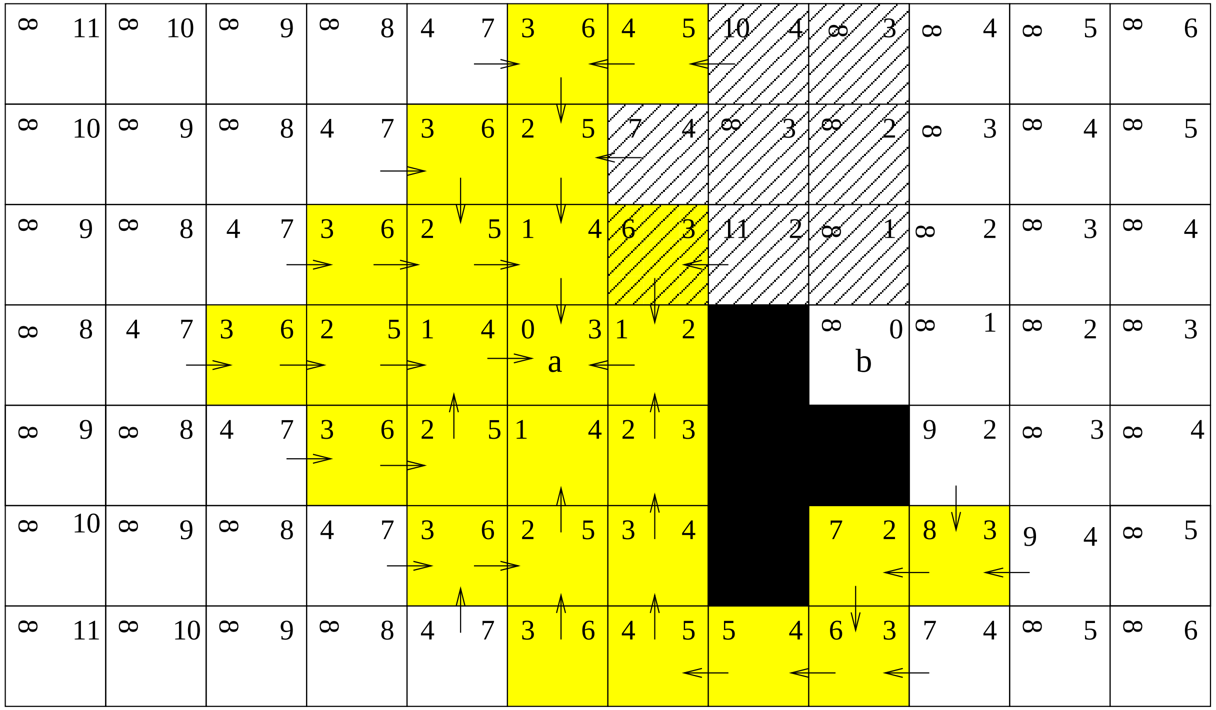
- Päädytään seuraavaan tilanteeseen



- Seuraavana käsittelyvuorossa yksi solmuista, jolle $alkuun(v) + loppuun(v) = 11$

- Oletetaan, että algoritmi valitsee näistä sattumalta käsittelyyn lähempänä maalia olevan

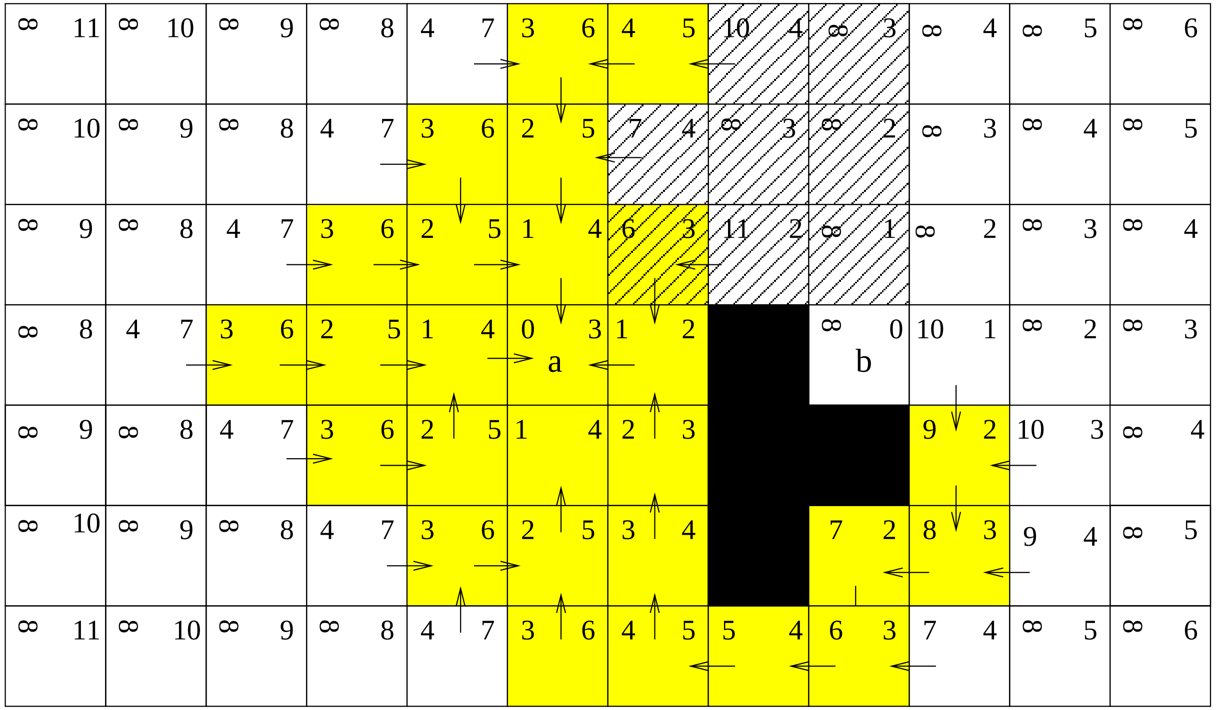
- Päädytään seuraavaan tilanteeseen



- Jälleen käsitteyvuorossa yksi solmuista, jolle $alkuun(v) + loppuun(v) = 11$

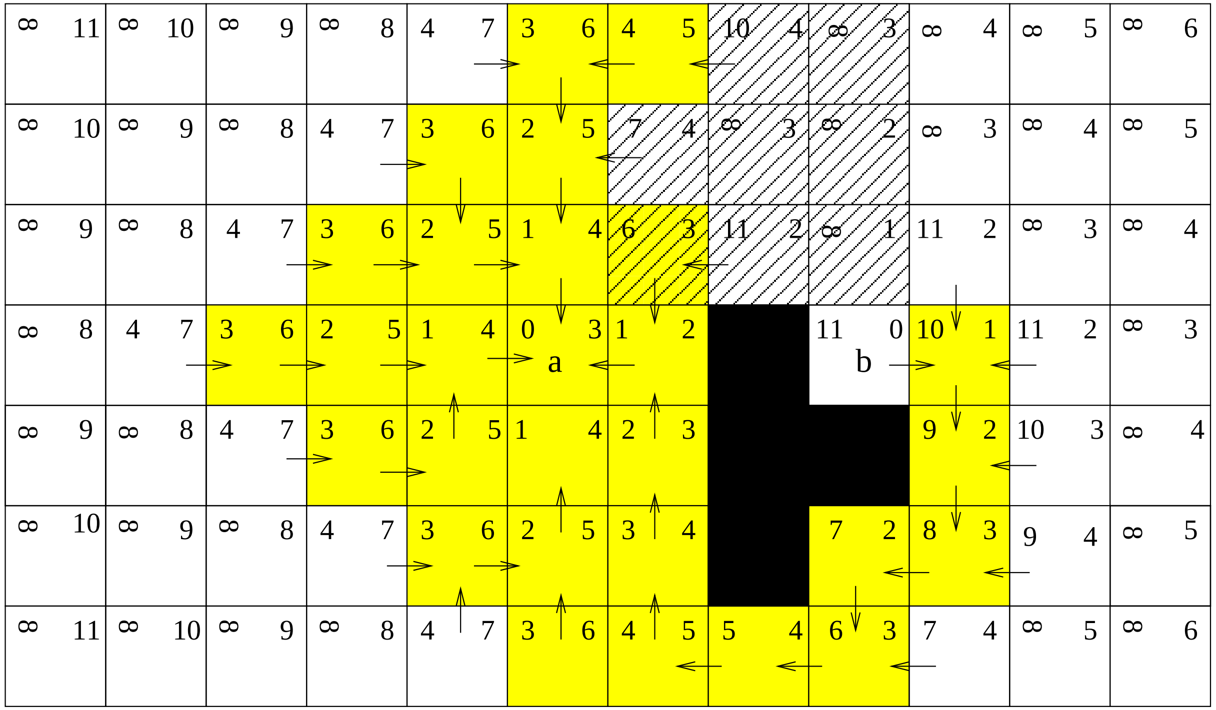
- Oletetaan, että algoritmi valitsee jälleen näistä maalia lähimpänä olevan

- Päädytään seuraavaan tilanteeseen



- Oletetaan taas, että algoritmi sattumalta valitsee käsittelyyn lähempänä maalisolmua olevan solmun

- Maalisolmu b löytyy vihdoin



- Kun maalisolmu tulee lisätyksi joukkoon S, algoritmi voi lopettaa ja lyhin polku on löytynyt

- Kuten huomaamme, algoritmi löytää lyhimmän polun huomattavasti nopeammin kuin **Dijkstran algoritmi**, jonka olisi täytynyt tutkia verkon kaikki polut joiden pituus on korkeintaan 11

- Seuraavalla sivulla algoritmi pseudokoodimuodossa

```
Astar(G,w,a,b)
// G tutkittava verkko, a lähtösolmu, b kohdesolmu ja w kaaripainot kertova funktio
for kaikille solmuille v ∈ V
    alkuun[v] = ∞
    loppuun[v] = arvioi suora etäisyys v ⇝ b
    polku[v] = NIL
alkuun[a] = 0
S = ∅
while ( solmu b ei ole vielä joukossa S )
    valitse solmu u ∈ V \ S, jolle alkuun[v]+loppuun[v] on pienin
    S = S ∪ {u}
    for jokaiselle solmulle v ∈ Adj[u]        // kaikille u:n vierussolmuille v
        if alkuun[v] > alkuun[u] + w(u,v)
            alkuun[v] = alkuun[u]+w(u,v)
            polku[v] = u
```

- Lyhin polku muodostuu nyt taulukkoon *polku* ja se on tulostettavissa samaan tapaan kuin **Dijkstran algoritmin** yhteydessä

- Jos oletetaan, että etäisyysarvio $loppuun[v]$ on laskettavissa vakioajassa, on algoritmin pahimman tapauksen aikavaativuus sama kuin **Dijkstran algoritmilla** eli $\mathcal{O}((|E| + |V|) \log |V|)$, jos toteutuksessa käytetään minimikekoa joukossa $V \setminus S$ olevien solmujen tallettamiseen

- Etäisyysarvio solmusta v maalisolmuun, eli arvo $loppuun(v)$ voidaan laskea monilla eri tavoilla riippuen sovelluksesta

- Esimerkissämme oli käytössä ns. Manhattan-etäisyys, eli oletettiin että eteneminen voi tapahtua vain ylös, alas ja sivuille

- Muita mahdollisuuksia esim.
  - diagonaalinen etäisyys joka sallii myös siirtymisen "väli-ilmansuuntiin"
  - euklidinen etäisyys eli viivasuora etäisyys

- Etäisyysarvio voi olla erilainen eri osissa verkkoa

- Jos polun loppuosan etäisyysarvioksi määritellään kaikille solmuille $loppuun(v) = 0$ toimii **A*** täsmälleen kuten **Dijkstran algoritmi**!

- Terminologiasta:
  - kirjallisuudessa etäisyysarviota $loppuun(v)$ kutsutaan heuristiikkafunktioksi ja siitä merkitään usein $h(v)$
  - alkumatkan etäisyysarviosta $alkuun(v)$ taas käytetään usein merkintää $g(v)$
  - joukossa S olevia jo käsiteltyjä solmuja sanotaan suljetuiksi solmuiksi
  - jo löydettyjä, mutta ei vielä joukkoon S vietyjä solmuja sanotaan avoimiksi solmuiksi

- On huomioinarvoista, että algoritmi löytää varmasti lyhimmän polun vain jos heuristiikkafunktion arvo eli loppuosan etäisyysarvio ei ole millekään solmulle suurempi kuin solmun todellinen etäisyys maalisolmusta ja heurstiikkafunktio on monotoninen
  - heuristiikkafunktio on monotoninen, jos mille tahansa vierekkäisille solmuille u ja v pätee $h(u) \leq w(u, v) + h(v)$
  - jos etäisyysarvio liiottelee joidenkin solmujen etäisyyksiä, löytää algoritmi jonkun polun, mutta polku ei välttämättä ole lyhin
  - tälläiset polut löytyvät keskimäärin nopeammin kuin lyhimmät polut
  - algoritmia on siis mahdollisuus joissain tilanteissa nopeuttaa, jos polun ei tarvitse olla täysin optimaalinen pituuden suhteen

- **A*-algoritmi** on ylikurssia, eli ei kuulu koealueeseen
- **A*** on kuitenkin suosittu aihe Tietorakenteiden harjoitustyössä
- Algoritmia käsitellään jonkin verran kurssilla Johdatus tekoälyyn
- Lisätietoa **A***:sta:
  - `www.policyalmanac.org/games/aStarTutorial.htm`
  - `theory.standford.edu/~amitp/GameProgramming/`
  - Russell and Norvig: Artifical Inteligence, A modern Approach

Vuo verkossa

- Tärkeä verkko-ongelmaperhe on vuoalgoritmit. Tämä on kuitenkin sekin ylikurssia, eli tästäkään aiheesta ei tule tenttikysymyksiä

- Vuoverkko (engl. flow network) $G = (V, E)$ on suunnattu verkko, joiden kaarilla $(u, v) \in E$ on kapasiteetti (engl. capacity) $c(u, v) \geq 0$. Jos $(u, v) \notin E$, asetamme $c(u, v) = 0$. Verkossa on lähde eli alkusolmu (engl. source) s ja kohde eli loppusolmu (engl. sink) t

- Vuo (engl. flow) on funktio $f : V \times V \to R$ siten, että

$$
\begin{array}{rl}
\text{kaikille } u, v \in V : f(u, v) \leq c(u, v) & , \quad \text{eli vuo ei ylitä missään kapasiteettia,} \\
\text{kaikille } u, v \in V : f(u, v) = -f(v, u) & , \quad \text{eli vuo kulkee yhteen suuntaan, ja} \\
\text{kaikille } u \in V - \{s, t\} : \sum_{v \in V} f(u, v) = 0 & , \quad \text{eli muissa kuin alku- ja loppusolmuissa} \\
 & \text{kaikki mitä tulee solmuun, jatkaa siitä ulos}
\end{array}
$$

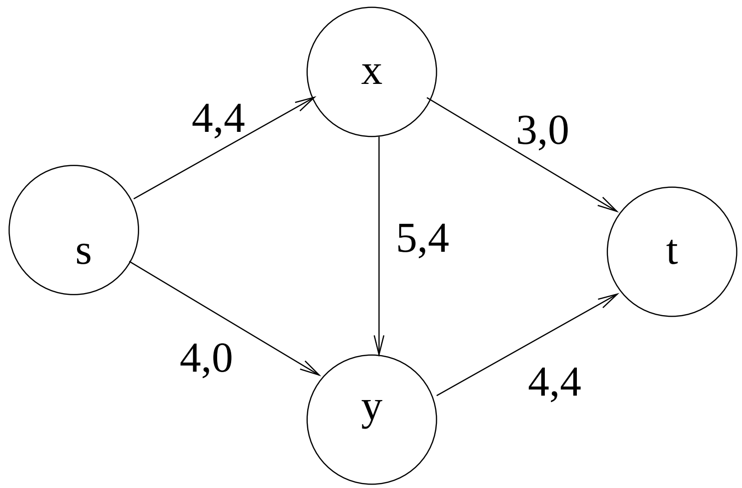
- Vuon arvo on tällöin $\sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$

- Luonnollinen kysymys on nyt: mikä on vuoverkon maksimivuo?

- Esimerkkejä tällaisista ongelmista on jonkin aineen virtaus putkissa, tavarantoimitus eri toimenpisteiden välillä ja tiedonsiirtokapasiteetti verkon yli

- Tarkastelemme nyt yleisellä tasolla, miten tätä voidaan laskea. Ajatelkaamme, että meillä on verkko, jossa on jo jokin vuofunktio olemassa (triviaalisti nollavuo on vuo)

- Huomaamme, että jos löytyy polku s:stä t:hen, niin että siinä on kapasiteettia jäljellä, niin voisimme lisätä vuota sitä reittiä pitkin



- Tässä verkossa kaaren ensimmäinen numero on kapasiteetti ja toinen vuo. Huomaamme, että vuota ei voi lisätä, mutta että verkossa ei kulje maksimivuota

- Jos kuitenkin viemme vuon 3 kulkemaan polkua (s, y, x, t), jossa ajattelemme vuota y:stä x.ään negatiivisena vuona x:stä y:hyn, niin verkon vuo kasvaa arvolla 3

- Kutsumme täydennysreitiksi (engl. augmenting path) sellaista suuntaamatonta polkua verkossa, että vuota voi lisätä, eli jokaiselle oikeaan suuntaan kuljetulla kaarella on kapasiteettia jäljellä ja jokaisella takaperin kuljetulla kaarella vuo ei mene negatiiviseksi

- Voidaan osoittaa, että kun täydennysreittejä ei enää ole, vuo on maksimaalinen

- Tämä menetelmä kutsutaan Ford-Fulkersonin menetelmäksi

- Lisätietoja: Cormenin luku 26

Verkkoalgoritmien yhteenveto tenttiä varten

- Olkoon verkossa $G = (V, E)$ n solmua ja m kaarta (eli $n = |V|$ ja $m = |E|$)

Algoritmi	**Aikavaativuus**	**Tilavaativuus**
BFS	$\mathcal{O}(n + m)$	$\mathcal{O}(n)$
DFS	$\mathcal{O}(n + m)$	$\mathcal{O}(n)$
Bellman-Ford	$\mathcal{O}(nm)$	$\mathcal{O}(n)$
Dijkstra	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}(n)$
Floyd-Warshall	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$
Prim	$\mathcal{O}(m \log n)$ / $\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Kruskal	$\mathcal{O}(m \log n)$	$\mathcal{O}(m)$

- Läpikäyntien suoria sovelluksia (eli aika- ja tilavaativuudet samat kuin läpikäynneillä):
  - Lyhin polku painottomassa verkossa = **BFS**
  - Sylkien löytäminen = **DFS** kohtaa harmaan solmun
  - Topologinen järjestäminen DAGissa = **DFS** + järjestäminen käänteisessä f-arvojärjestyksessä
  - Kriittiset työvaiheet DAGissa = pisin polku DAGissa = topologinen järjestäminen + pituuden laskeminen f-arvojärjesteyksessä = **DFS** + pituus siitä eteenpäin on selvä, kun solmu muuttuu mustaksi
  - Vahvasti yhtenäiset komponentit = **DFS** + verkon transponointi + transpoosiverkon **DFS**, läpikäynnin alkusolmuksi valitaan aina jäljellä olevista se jolla on suurin f-arvo
- Algoritmit soveltuvat kaikki sekä suunnattuihin että suuntaamattomiin verkkoihin, paitsi DAG-sovellukset sekä **Prim** ja **Kruskal**, koska pienin virittävä puu on määritelty vain suunttaamattomiin verkkoihin

- Lyhin polku painotetussa verkossa:
  - Mikään näistä (**Bellman-Ford**, **Dijkstra**, **Floyd-Warshall**) ei toimi, jos on negatiivinen sykli
  - **Dijkstra** ei toimi negatiivisille painoille, muut kylläkin
  - **Dijkstra** toimii sellaisenaan suuntaamattomissa verkoissa
  - **Bellman-Ford** ja **Floyd-Warshall** voidaan käyttää myös suuntaamattomissa verkoissa, kunhan verkossa ei ole negatiivisia painoja, korvaamalla kukin suuntaamaton kaari suunnatulla kaarella molempiin suuntiin

9. Algoritminsuunnittelumenetelmiä

- Tämä luku on lähinnä kertausta kurssin aikana tavatuista algoritminsuunnittelumenetelmistä

- Tässä yhteydessä on hyvä muistuttaa invariantin käsitteestä, josta on iloa myös perusohjelmoinnissa turhien virheiden välttämiseksi. Ylipäätään ohjelmoinnissa on aina syytä tarkastella alku- ja lopetusehtoja, erikoistapauksia jne., ja invariantit ovat hyviä työkaluja erinäisten for- ja while-silmukoiden hallitsemiseksi

Raaka voima (Brute force)

- Helppo lähestymistapa on käydä kaikki mahdolliset tapaukset läpi
- Tämä menetelmä on yleensä tehoton, mutta käyttökelpoinen erityisesti pienikokoisille ongelmille tai jos muuta ei voida tehdä ongelman luonteen takia
- Esimerkki, jossa raaka voima on hyvä lähestymistapa:
  - Etsimme pienintä avainta järjestämättömästä linkitetystä listasta
  - Tässä käymme alkiot läpi yksitellen alusta lähtien, pidetään kirjaa tähän asti pienemmästä avaimesta, ja lopetamme kun lista loppuu
  - Tällainen hakumenetelmä kutsutaan peräkkäishauksi (engl. sequential search)
- Esimerkki, jossa raaka voima ei ole hyvä lähestymistapa:
  - Haluamme löytää tiheästä verkosta lyhimmän polun solmusta s solmuun t
  - Generoidaan kaikki mahdolliset polut s:stä t:hen ja laskemme kullekin polun pituus
  - Valitaan lyhin
- Tällainen ”kaikkia vaihtoehtoja luetteleva” menettely kutsutaan joskus leikkisästi British Museum -etsinnäksi.

Ahne algoritmi (Greedy algorithm)

- Edetään vaiheittain, kussakin vaiheessa siitä kohdasta katsottuna parhaaseen vaihtoehtoon. Kyse on siis paikallisesta optimoinnista
- Olemme nähneet tästä menetelmästä useita esimerkkejä:
  - **Dijkstran**, **Kruskalin** ja **Primin** algoritmit ovat tällaiset ja näissä tapauksissa menetelmä tuottaa todistettavasti optimitulokset
- Yleisesti ottaen tällä menetelmällä pääsee lokaaliin optimiin, joka voi olla globaalia optimia huonompi
- Optimointitehtävissä puhutaan usein naapuruushausta (engl. hill climbing): haluamme korkeimmalle kukkulalle, joten suuntaamme siihen suuntaan, jossa nousu on jyrkintä
- Esimerkki: Haluamme kaupungissa kävellä paikasta A paikkaan B
  - Otamme joka kadunkulmassa sen vaihtoehdon joka näyttää vievän parhaiten paikan B suuntaan

Peruuttava etsintä (Backtracking)

- Edetään kunnes tullaan umpikujaan ja palataan taaksepäin yrittämään uudestaan toista vaihtoehtoa
- Tätä menetelmää tarkastelimme luvun 3 lopussa
- Optimointitehtävissä pystytään myös karsimaan (prune) pitkin matkaa huonoja vaihtoehtoja (eli jätämme ne pois menemättä niihin syvemmin) ja menetelmä kutsutaan silloin nimellä branch-and-bound
- Peruuttavat algoritmit perustuvat rekursion käyttöön eli pinoon

Hajota ja hallitse (Divide and conquer)

- Tämä on tavallinen ongelman lähestymistapa tietojenkäsittelyssä: jaetaan ongelma pienempiin samanlaisiin ongelmiin (''hajota''), joita yhdistämällä (''hallitse'') saadaan lopullinen vastaus

- Tässäkin edetään rekursiivisesti: osaongelmat jaetaan puolestaan rekursiivisesti pienempiin osaongelmiin

- Huomaa: Kaikki rekursio ei ole hajota ja hallitse: tässä viitataan tapauksiin, jossa jaetaan ongelma vähintään kahdeksi (samantyyppiseksi) osaongelmaksi

- Esimerkkejä tällaisista algoritmeista on lomitusjärjestäminen ja pikajärjestäminen

- Binäärihaku on sukua tähän lähestymistapaan, mutta siinä vaan jaetaan ongelma pienemmäksi eikä tarvitse tehdä mitään yhdistämisoperaatiota lopussa. Tällainen tekniikka kutsutaan joskus nimellä yksinkertaista ja hallitse

Dynaaminen ohjelmointi (Dynamic programming)

- Rekursio voi monesti olla tehoton, joten voidaan tallentaa osaongelmien tulokset taulukkoon. Osatulosten taulukointi kutsutaan nimellä dynaaminen ohjelmointi

- Esimerkkejä tästä on mm. Fibonacci-lukujen iteratiivinen laskeminen parin apumuuttujan avulla sekä **Floyd-Warshallin algoritmi** (tai transitiivisen sulkeuman laskeminen), jossa päivitetään kaksiulotteista taulukkoa (matriisia)

Algoritmien suunnittelu

- Lisää algoritmien suunnittelusta seuraa kurssilla Design and Analysis of Algorithms
- Tällä kurssilla päätavoite on ymmärtää esitettyjen algoritmien ideat ja osata soveltaa niitä hieman muuntaen
- Käytännössä algoritmit
  - toteutetaan tietokonelaitteistolla ja
  - liittyvät osaksi sovellusohjelmaa
- Tällä kurssilla on lähinnä tarkasteltu asymptoottista pahimman tapauksen aikavaativuutta ("$\mathcal{O}$-notaatio"), joka kertoo yksinkertaisessa muodossa algoritmin skaalautuvuuden

- Sovelluksessa tarvitaan muutakin:
  - Kuinka isoja syötteet todella ovat? Helppoja vai vaikeita?
  - Mikä on käyttäjien tarvitsema/haluama riittävä suorituskyky? Mikä oikeasti on järjestelmän pullonkaula?
  - Jos algoritmia pitää tosissaan ruveta viilaamaan, miten se suhtautuu käytettävissä olevaan suoritusympäristöön (rinnakkaistuminen, välimuistin käyttö jne.)?
  - Käytännön tekijät: ohjelmiston ylläpidettävyys ja laajennettavuus, virheistä toipuminen jne.

Tietorakenteet ja algoritmit

abstrakti tietotyyppi: mitä datalle halutaan tehdä

tietorakenne: miten se tehdään (talletusrakenne, operaatioiden toteutus)

- Kaikki ei ole ihan yksinkertaista:
  - tietorakenteiden toteuttamisessa voidaan tarvita ei-triviaaleja algoritmeja (esim. AVL-puiden tasapainotus)
  - algoritmien tehokkaassa toteuttamisessa voidaan tarvita ei-triviaaleja tietorakenteita (esim. keko ja union-find)

Millaisia asioita olemme oppineet

- Toisin sanoen mitä kurssista olisi hyvä muistaa vielä ensi vuonna

- Perustietorakenteita ja -algoritmeja, jotka on hyvä osata jopa koodata melko sujuvasti:

  – pino, jono, linkitetty lista, puu

  – syvyyssuuntainen ja leveyssuuntainen läpikäynti

- Yleisesti tarvittavia tietorakenteita ja algoritmeja, joita ei yleensä tarvitse (tai kannata) itse koodata, mutta keskeiset periaatteet pitää tuntea:

  – hakemistorakenteet: puu vs. hajautus

  – järjestäminen: $\mathcal{O}(n)$ vs. $\mathcal{O}(n \log n)$ vs. $\mathcal{O}(n^2)$; pikajärjestämisen vaikeat tapaukset

- Mallinnustekniikoita: etenkin puiden ja verkkojen käyttäminen ongelmanratkaisun mallina
  - tekoäly: pelipuut, puun läpikäynnit, verkko-algoritmit, graafiset mallit (verkkopohjaisia todennäköisyysmalleja)
  - tietoliikenne: esim. reitittäminen **Dijkstran algoritmilla**
  - ohjelmointi ja ohjelmointikielet: jäsennyspuu, verkkopohjaiset kaavioesitykset
  - tietojenkäsittelyteoria: puut ja verkot abstrakteissa laskennan malleissa

Loppusanat

- Kurssi on toivottavasti ollut teille antoisa ja hyödyllinen
- Muistakaa kurssipalautteen antamista. Erityisen arvokkaat ovat suositukset, miten kurssia voisi parantaa

KIITOS!