7.4 Lyhimmät polut

- 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
- esitellään ongelmalle ratkaisuksi *Dijkstran algoritmi* joka olettaa että kaaripainot ovat *positiivisia*
- Negatiivisten kaaripainojen tapauksessa lyhimmät polut voidaan ratkaista esim. Bellman-Ford algoritmilla, joka kuitenkin sivuutetaan tällä kurssilla

- Olkoon $G = (V, E)$ suunnattu verkko ja s eräs verkon solmu
- oletetaan että jokaiseen kaareen $(u, v) \in E$ on liitetty pituus $w(u, v)$ joka on ei-negatiivinen reaaliluku
- 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
- Dijkstran algoritmi pitää yllä joukkoa S joka muodostuu solmuista joiden lyhin etäisyys solmuun s on jo selvitetty
- jokaiseen solmuun v liittyy kaksi attribuuttia $d[v]$ ja $p[v]$

- $d[v]$ kertoo mikä on solmun v *etäisyysarvio* solmusta s
- $p[v]$ osoittaa minkä solmun kautta v:hen saavutaan lyhimmällä solmusta s lähtevältä polulta

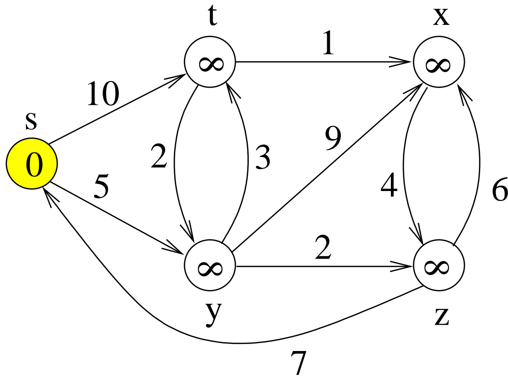
- aluksi algoritmi asettaa kaikkien solmujen (paitsi solmun s) etäisyysarvioksi äärettömän, sekä attribuutin $p[v]$ arvoksi NIL
- algoritmi valitsee rivillä 9 toistuvasti solmun $u \in V - S$, jonka etäisyys-arvio solmuun s on pienin
  - valittu solmu u lisätään joukkoon S, ja
  - kaikkien solmun u vierussolmujen etäisyysarvio solmuun s sekä attribuutti $p[u]$ päivitetään
- ensimmäisessä toistoaskeleessa tulee valituksi aloitussolmu s sillä $d[s] = 0$
  - solmu s siis lisätään joukkoon S
  - kaikille s:n vierussolmuille v asetetaan uusi etäisyysarvio, joka on nyt $d[s] + w(s, v) = 0 + w(s, v) = w(s, v)$, eli sama kuin etäisyys solmujen s ja v välillä
  - attribuuttien $p[v]$ arvoksi tulee s, sillä lyhin polku solmusta s solmuun v saapuu solmun s kautta
- toisessa toistoaskeleessa tulee valituksi se solmu u jonka etäisyys solmuun s on lyhin
  - solmu u siis lisätään joukkoon S
  - kaikille u:n vierussolmuille v joiden entinen etäisyysarvio solmuun s on suurempi kuin $d[u] + w(u, v)$:
  - asetetaan uusi etäisyysarvio, joka on nyt $d[u] + w(u, v)$, eli sama kuin etäisyys $s \rightarrow u$ + etäisyys $u \rightarrow v$, ja

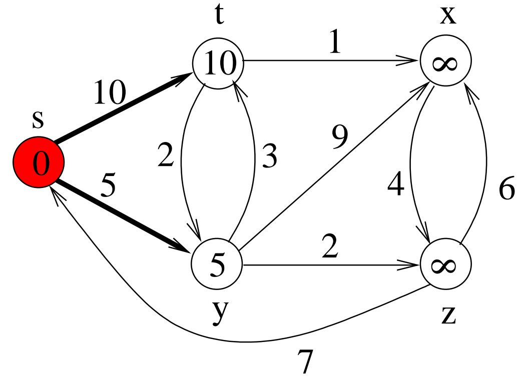
  - attribuuttien $p[v]$ arvoksi päivitetään u, sillä lyhin polku solmusta s solmuun v saapuu solmun u kautta

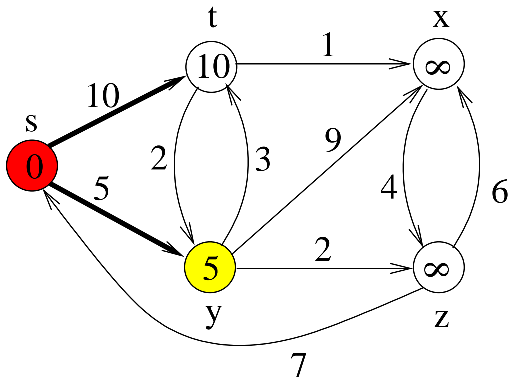
- sama jatkuu kolmannessa toistoaskeleessa ...
- algoritmi käyttää aputietorakenteena minimikekoa H
  - solmut $v \in V - S$ pidetään keossa
  - solmun v avaimena sen etäisyysarvioattribuutin $d[v]$ arvo
  - solmun etäisyysarvion päivityksen yhteydessä kutsutaan heap-decrease-key-operaatiota joka asettaa solmun taas oikealle kekopaikalle

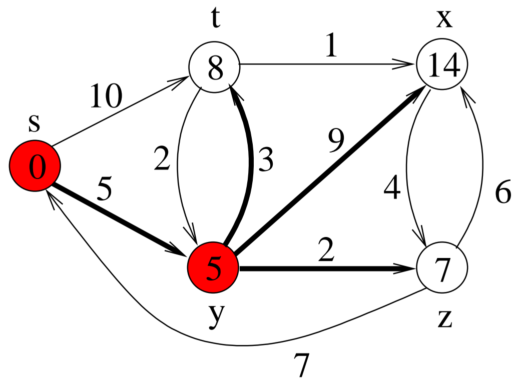
```
Dijkstra(G,w,s)
1   for kaikille solmuille v ∈ V do
2       d[v] ←∞
3       p[v] ←NIL
4   d[s] ←0
5   S ←∅
6   for kaikille solmuille v ∈ V do
7       heap-insert(H,v,d[v])
8   while ( not empty(H) ) do
9       u ←heap-del-min(H)
10      S ←S ∪ {u}
11      for jokaiselle solmulle v ∈ Adj[u] do
12          if d[v] > d[u] + w(u,v) then
13              d[v] ←d[u]+w(u,v)
14              p[v] ←u
15              heap-decrease-key(H,v,d[v])
```

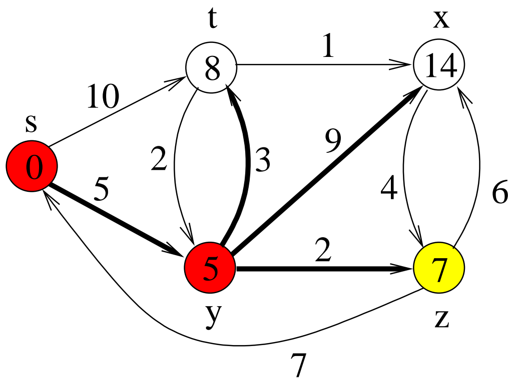
• esimerkki algoritmin toiminnasta

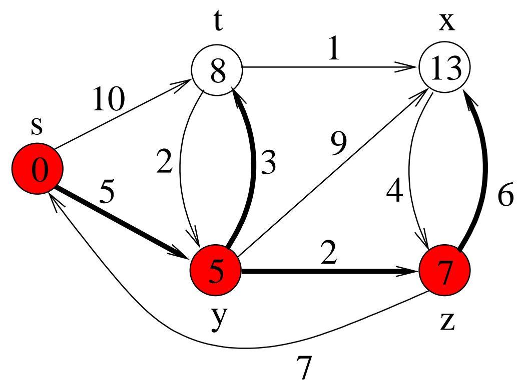


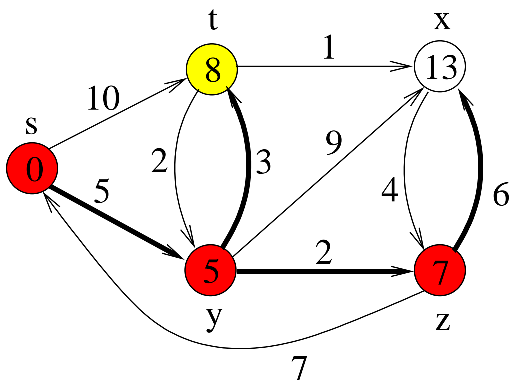


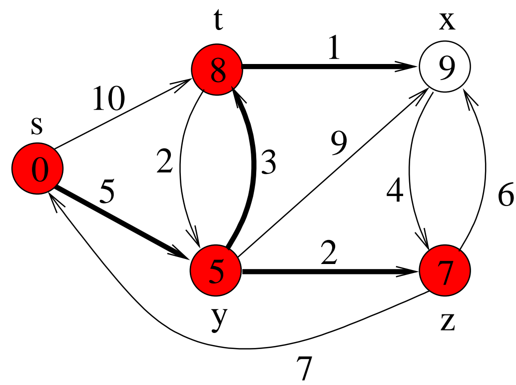


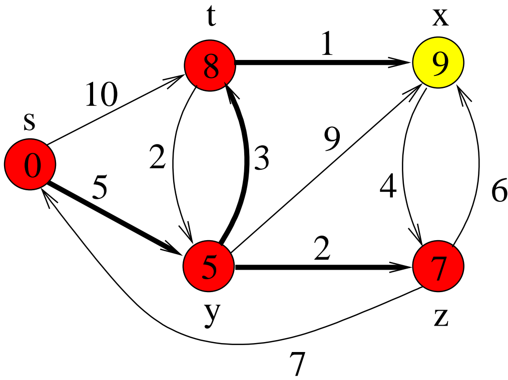


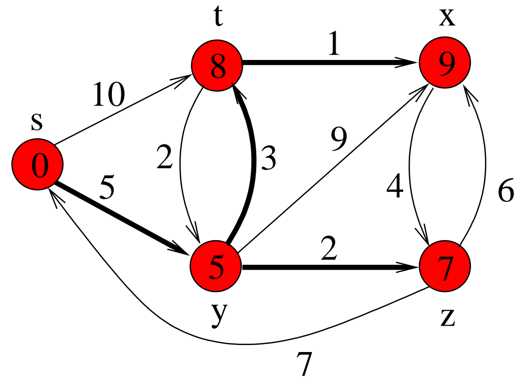










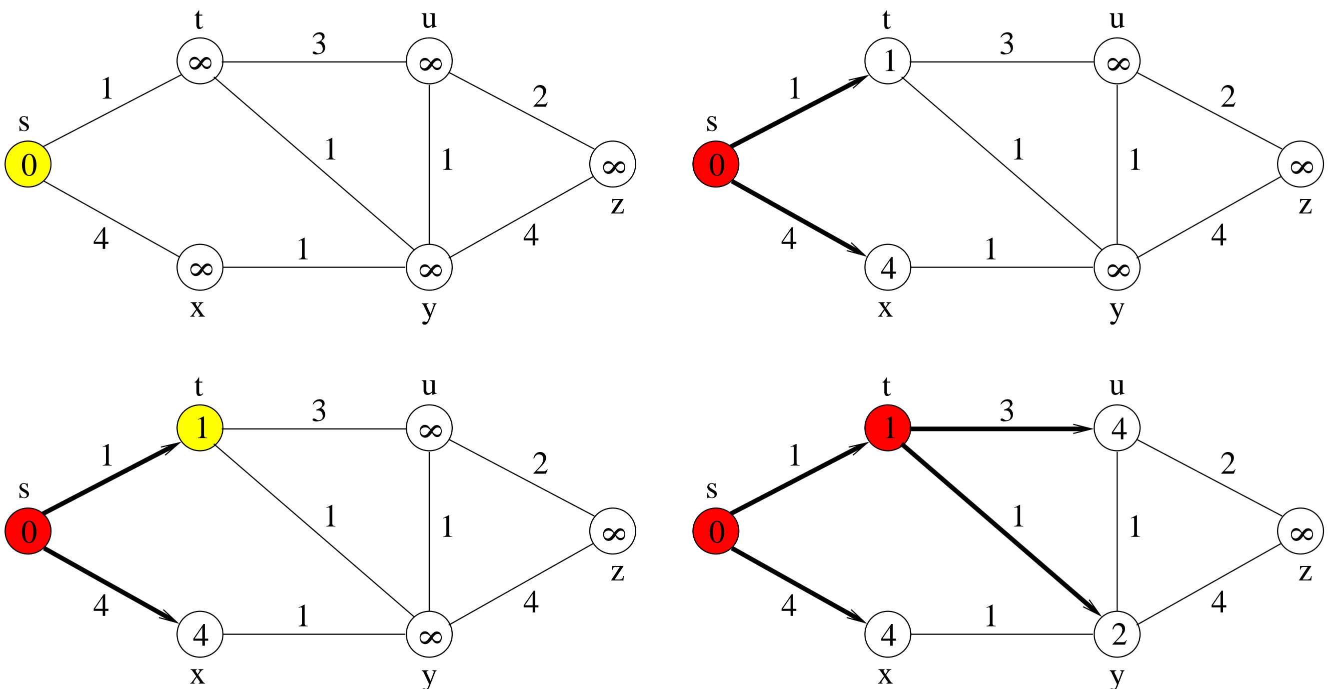


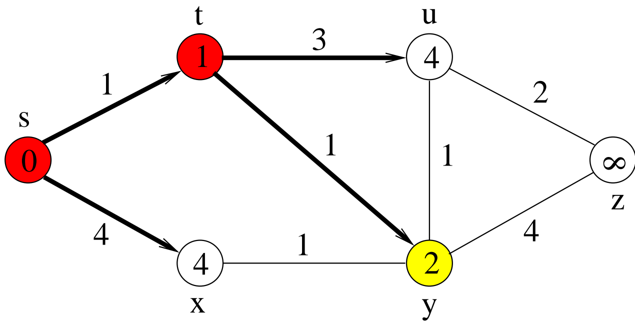
- kuvassa attribuuttien $p[v]$ arvo on kuvattu paksunnettuina kaarina
- algoritmin suorituksen jälkeen lyhin polku $s \rightsquigarrow v$ saadaan selville seuraavasti:
  - $p[v]$ kertoo minkä solmun kautta lyhin polku $s \rightsquigarrow v$ saapuu solmuun v
  - solmuun $p[v]$ lyhin polku saapuu solmun $p[p[v]]$ kautta, jne
  - laitetaan pinoon $p[v], p[p[v]], p[p[p[v]]]$ ja tulostetaan pinon sisältö
  - näin saadaan tulostettua polulla koko polku $s \rightsquigarrow v$ alusta loppuun

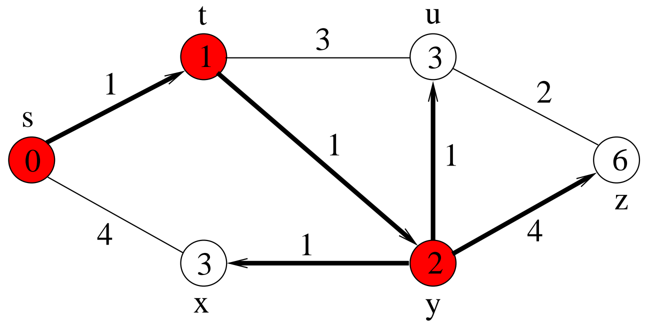
- algoritmina:

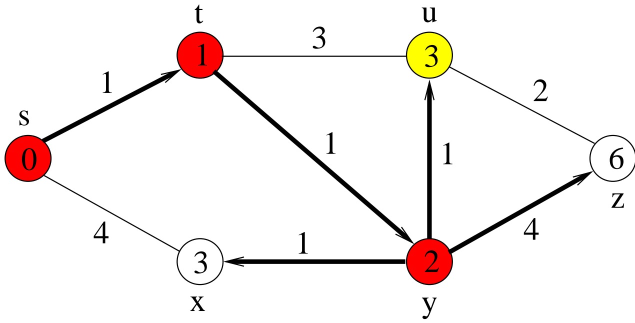
  shortest-path(G,v)
  1 u ←p[v]
  2 **while** u ≠ s **do**
  3   push(S,u)
  4   u ←p[u]
  5 print(lyhin polku solmusta s solmuun v)
  6 **while** not empty(S)
  7   u ←pop(S)
  8   print(u)

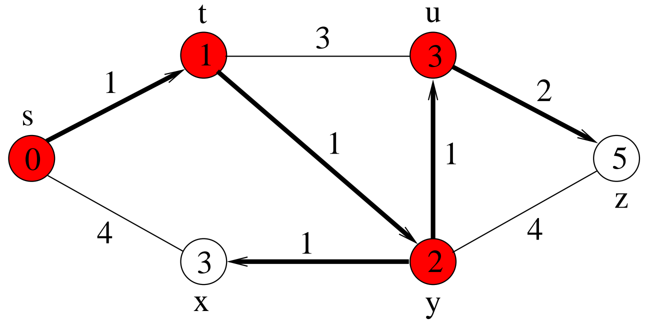
- toinen esimerkki Dijkstran algoritmin toiminnasta, tällä kertaa kyseessä suuntaamaton verkko

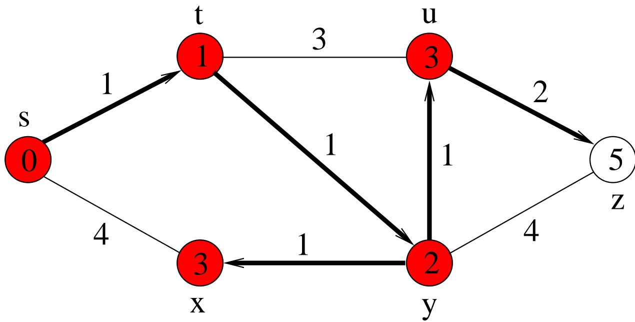


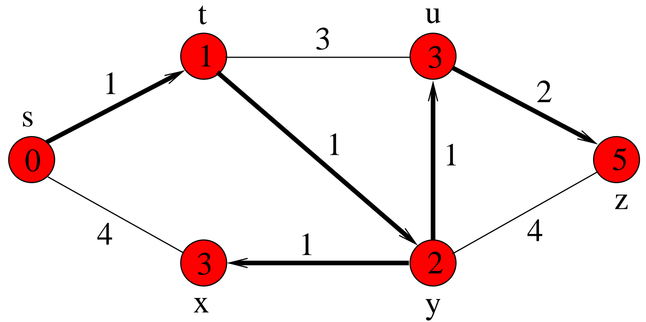












- Dijkstran-algoritmin vaativuus
  - rivien 1-5 alustustoimet vievät aikaa selvästi $\mathcal{O}(|V|)$
  - käytössä siis minimikeko, ja kutsutaan keko-operaatioita heap-insert, heap-del-min ja heap-decrease-key
  - keko-operaatioiden vaativuus on $\mathcal{O}(log\ n)$ jos keossa n alkiota
  - rivillä 7 tehdään $|V|$ kappaletta heap-insert-operaatioita, aikaa siis kuluu $\mathcal{O}(|V|\ log\ |V|)$

- rivien 8-15 toistolauseessa 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 rivillä 12 kerran, eli 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 $\mathcal{O}(|V|)$

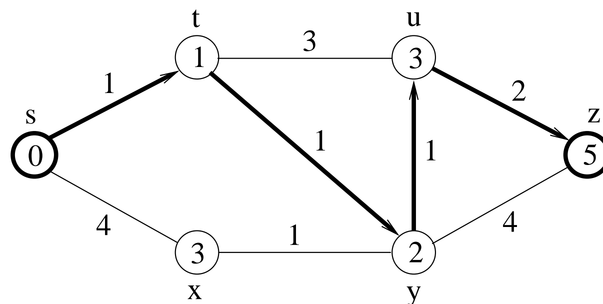
- Dijkstran algoritmi noudattaa ns. *ahneen algoritmin* (greedy algorithm) strategiaa:
  - rivillä 9 valitaan aina käsiteltäväksi se solmu mikä on lähimpänä aloitussolmua s
  - ahneudella tarkoitetaan tässä sitä että algoritmi pyrkii joka hetkellä juuri silloin "parhaalta" vaikuttavaan ratkaisuun
  - ei ole itsestäänselvää että ahne strategia tuottaa nimenomaan lyhimmät polut, mutta Dijkstran algoritmin tapauksessa näin on, todistus löytyy sekä Karvin monisteesta että Cormenin kirjasta
  - todistus perustuu invarianttiin: *toistolauseen alussa kaikille solmuille $v \in S$ pätee: $d[v]$ on sama kuin lyhimmän*

polun $s \rightsquigarrow v$ *pituus*

  - eli kun solmu on lisätty joukkoon S, sen lyhin etäisyys aloitussolmuun on selvinnyt

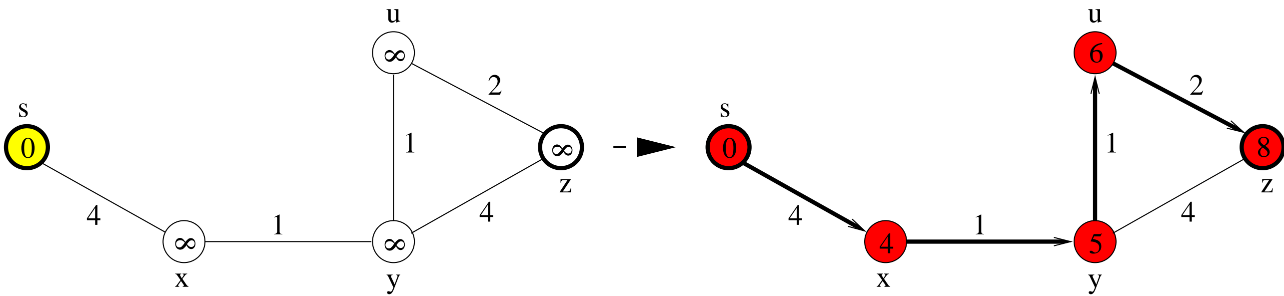
- joskus meille riittää tietää pelkästään kahden solmuparin s ja v välinen lyhin polku
- ajetaan Dijkstran algoritmia lähtösolmuna s siihen asti kunnes solmu v lisätään joukkoon S
- tämän jälkeen voidaan lopettaa sillä lyhin polku $s \rightsquigarrow v$ on jo selvinnyt

- joskus voi olla tilanne että haluaisimme lyhimmän polun lisäksi tietää myös muita "hyviä" polkuja
- tälle *etsi k lyhintä polkua solmusta s solmuun v* on olemassa paljon erilaisia spesialisoituneita ratkaisumenetelmiä
- hahmotellaan tässä Dijkstran algoritmia käyttävä menetelmä millä saamme selvitettyä myös muutamia muita hyviä polkuja:
- tarkastellaan edellisen esimerkkimme verkkoa ja polkuja $s \rightsquigarrow z$

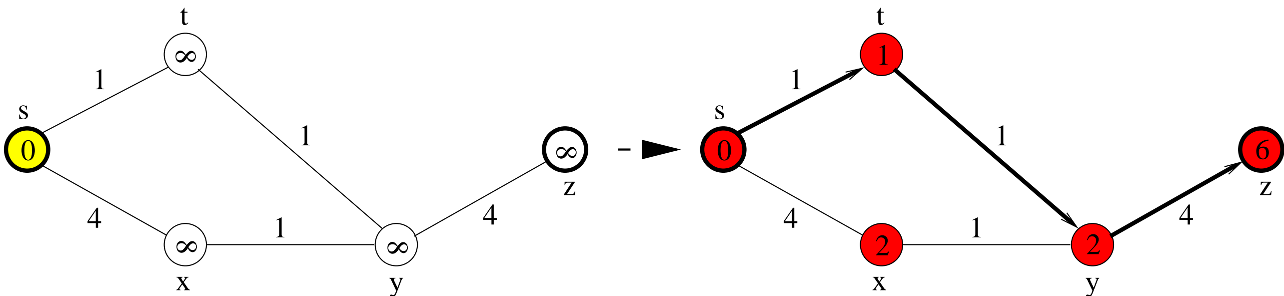


- paras polku kulkee nyt solmujen t, y ja u kautta

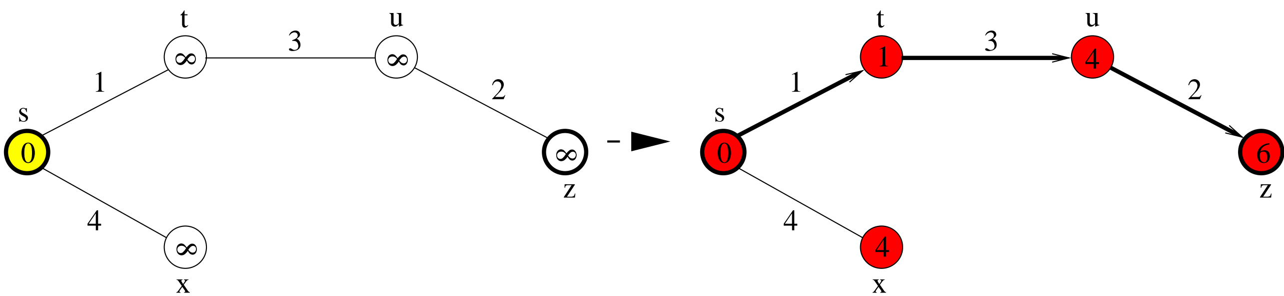
- vaihtoehtoinen hyvä polku saadaan poistamalla verkosta joku solmuista t, y, u, ja suorittamalla Dijkstran algoritmi tuloksena olevalle verkolle:
  - verkosta poistettu solmu t, ja näin saatu lyhin polku



  - verkosta poistettu solmu y, ja näin saatu lyhin polku



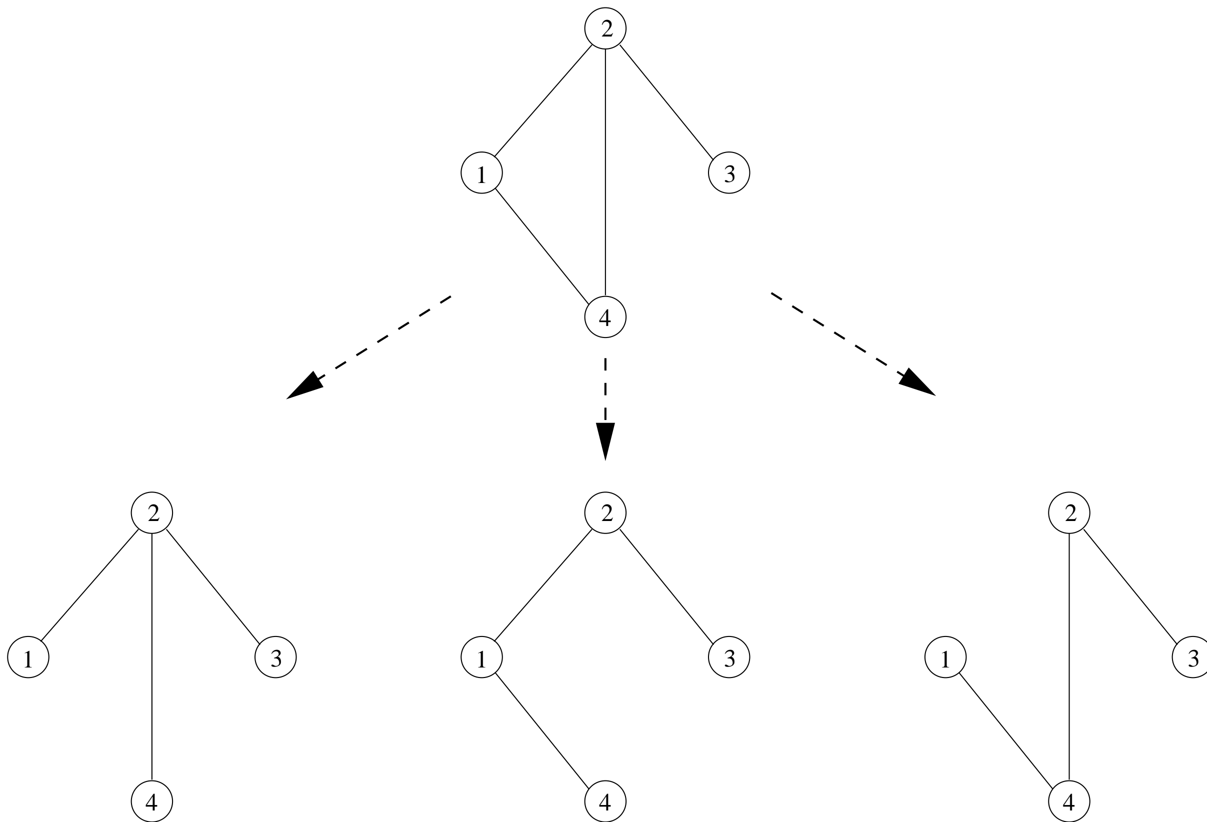
  - verkosta poistettu solmu u, ja näin saatu lyhin polku



- ei ole kuitenkaan taattua että tämä menetelmä tuottaa nimenomaan k parasta polkua, kyseessä on siis *heuristinen menetelmä*

7.5 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
- verkko ja sen virittäviä puita



- seuraavassa yksinkertainen algoritmi joka muodostaa verkolle $G = (V, E)$ jonkin virittävän puun
- algoritmin lopussa joukossa E' olevat kaaret muodostavat virittävän puun

spanning-tree(G)

1 valitaan mielivaltainen solmu $v \in$ V

2 V'←$\{v\}$

3 E' ←$\emptyset$

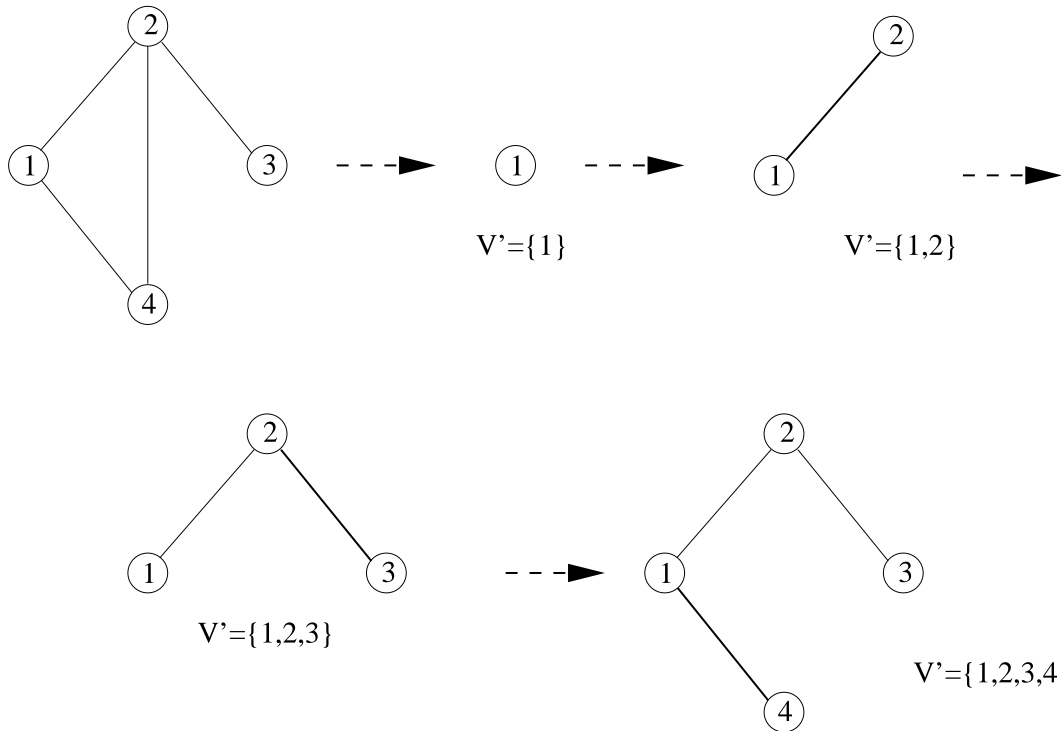
4 **while** V' $\neq$ V **do**

5 valitse kaari $(u, v) \in E$ siten että $u \in V'$ ja $v \in V - V'$

6 V'←V' $\cup$ $\{v\}$

7 E' ←E' $\cup$ $\{(u, v)\}$

- esimerkki algoritmin toiminnasta:



- algoritmin vaativuus riippuu paljolti rivin 5 operaation toteutuksesta

- pienellä miettimisellä päästään toteutukseen joka toimii ajassa $\mathcal{O}(|V| + |E|)$

- onko varmaa että algoritmi toimii oikein, eli virittävä puu löytyy?
- todistaaksemme toiminnan oikeellisuuden, meidän on osoitettava että seuraavat asiat ovat voimassa
  - muodostettavaan virittävään puuhun (V', E') ei tule sykliä toistolauseen sisällä
  - jos $V' \neq V$ niin rivillä 5 on aina mahdollista löytää ehdot täyttävä kaari

- virittävät puut tulevat käyttöön monissa sovelluksissa
- esim. tietokoneverkkoprotokollissa ja hajautetuissa algoritmeissa koneet joutuvat usein jakamaan tietoa keskenään, tämä hoidetaan muodostamalla virittävä puu ja käyttämällä sitä sanomien välittämiseen

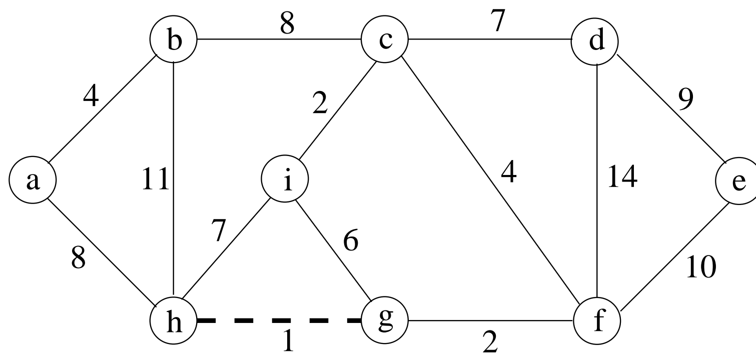
- äsken esitetty algoritmi muodostaa verkosta jonkin virittävän puun, jos kyseessä on painotettu verkko, olemme yleensä kiinnostuneita minimaalisen virittävän puun muodostamisesta
- Olkoon $G = (V, E)$ suuntaamaton yhtenäinen painotettu verkko, jonka kaaripainot määrää funktio w
- verkon G *minimaalinen virittävä puu* (engl. minimal spanning tree) on G:n virittävistä puista se jonka kaaripainojen summa on pienin
- esittelemme seuraavassa kaksi hieman eri periaatteella toimivaa algoritmia minimaalisen virittävän puun muodostamiseen

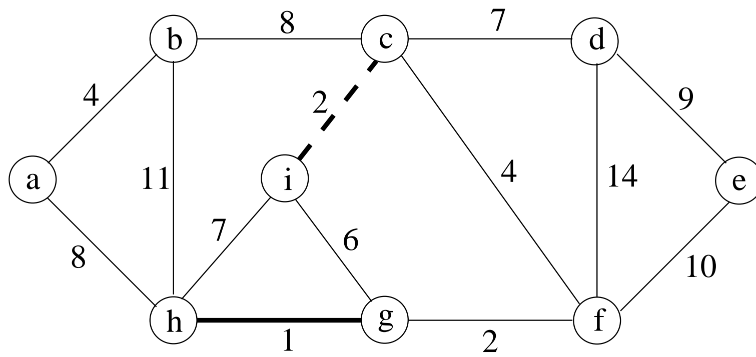
7.5.1 Kruskalin algoritmi

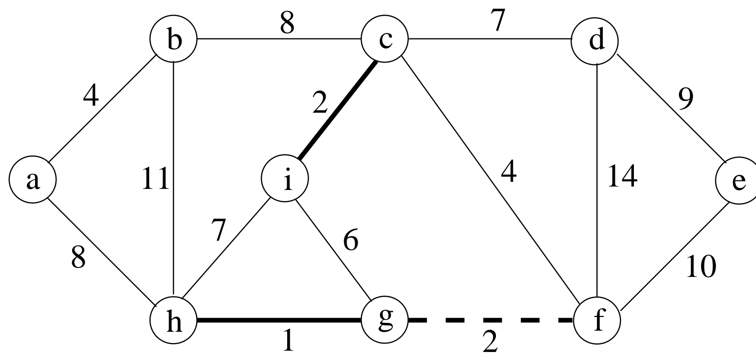
- algoritmi alkaa muodostamaan virittävää puuta siten että muodostusvaiheessa koossa on useita erillisiä paloja puusta
- merkitsemme attribuutilla $P[v]$ palaa johon solmu v kuuluu
- algoritmin lopussa nämä palat yhdistyvät yhtenäiseksi puuksi joka on pienin virittävä puu
- algoritmi kerää virittävän puun kaaria joukkoon A ja palauttaa lopuksi joukon
- algoritmin lopussa verkon $G = (V, E)$ minimaalinen virittävä puu siis on (V, A)

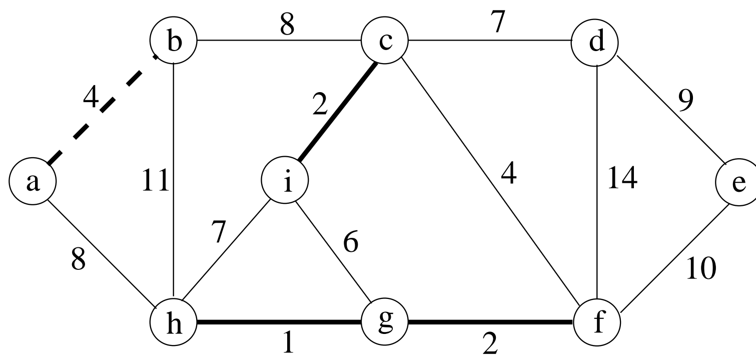
Kruskal(G,w)
1 A $\leftarrow \emptyset$
2 **for** kaikille solmuille $v_i \in$ V **do**
3   solmu v_i muodostaa oman palan $P[v_i]$
4 järjestetään kaaret painon w mukaan kasvavaan järjestykseen
5 **for** jokaiselle kaarelle $(u, v) \in E$ kasvavassa järjestyksessä **do**
6   **if** P[u] $\neq$ P[v] **then**
7   A $\leftarrow$A $\cup \{(u, v)\}$
8   yhdistä palat P[u] ja p[v]
0 **return** A

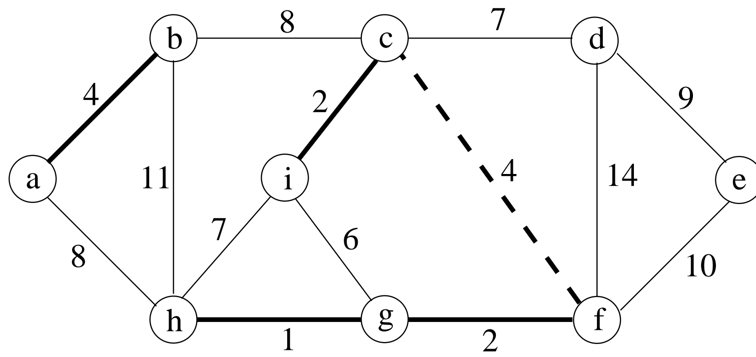
- esimerkki algoritmin toiminnasta:

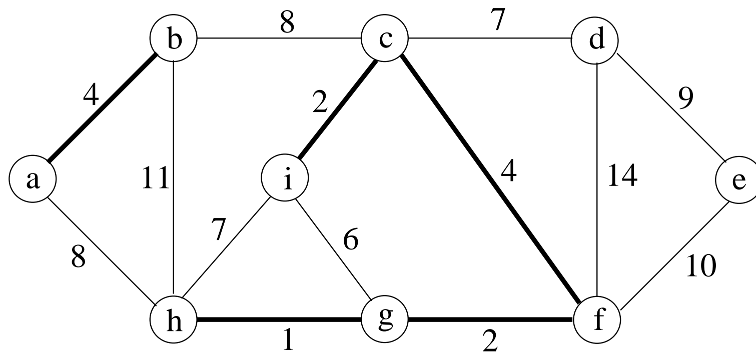

b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2

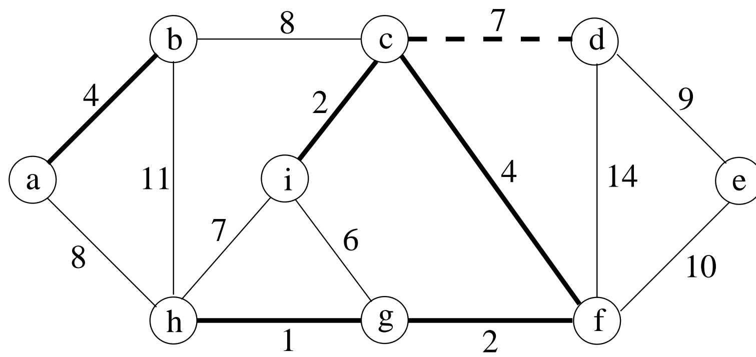

b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2

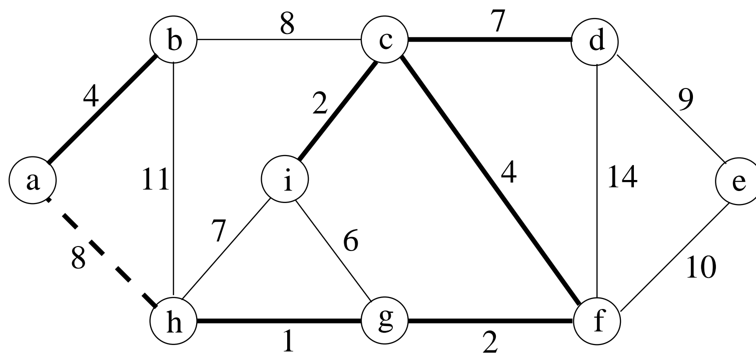

b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2

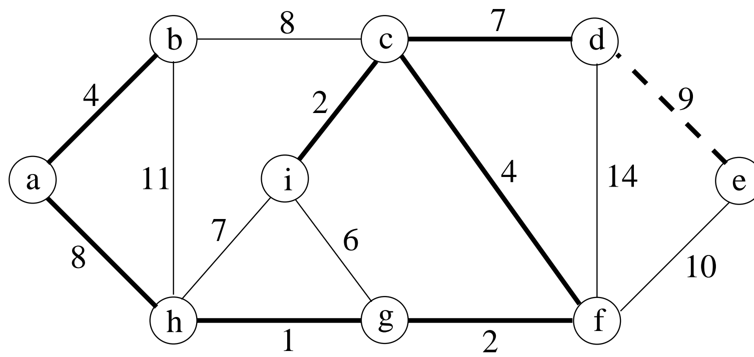

b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2

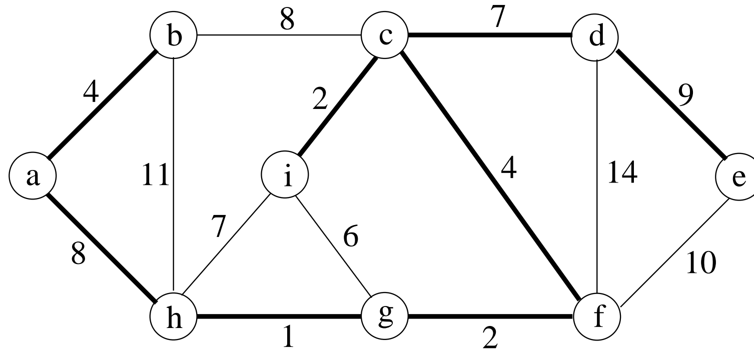

a
b
c
d
e
f
g
h
i
4
8
7
2
11
4
14
9
7
6
8
10
1
2


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


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


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





- Algoritmi itsessään on hyvin yksinkertainen mutta sen toteuttaminen ei välttämättä ole aivan yksinkertaista
- jotta toteutuksesta tulee tehokas, on erityisesti kiinnitettävä huomiota siihen miten solmuihin liittyvä pala-attribuutti toteutetaan
- suoraviivainen tapa palan toteuttamiseen:
  - talletetaan palatieto $|V|$ paikkaiseen taulukkoon
  - 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 $P[v]$ uudeksi palanumeroksi $P[u]$
- palat tallettava taulukko on siis käytävä läpi ja näin rivin 8 vaativuus on luokkaa $\mathcal{O}(|V|)$

- koko algoritmin aikavaativuus:
  - oletetaan edellä kuvailtu suoraviivainen tapa toteuttaa palat
  - alkutoimet riveillä 1-3 vievät aikaa $\mathcal{O}(|V|)$
  - rivin 5 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 operaatio
  - saamme siis vaativuudeksi $\mathcal{O}(|E|\ log\ |E| + |V|^2)$
- käyttämällä ns. union-find -rakennetta palojen toteuttamiseen, päästään algoritmissa aikavaativuuteen $\mathcal{O}(|E|\ log\ |E|)$
- vielä kun huomioidaan että $|E| \leq |V|^2$ josta taas seuraa $log\ |E| = \mathcal{O}(log\ |V|)$, saamme union-find-rakenteen avulla toteutetun Kruskalin algoritmin aikavaativuuden muotoon $\mathcal{O}(|E|\ log\ |V|)$
- myös Kruskalin algoritmi noudattaa ahnetta strategiaa: yritetään aina lisätä virittävään puuhun solmuista kevein, eikä ole aivan itsestään selvää että algoritmi tuottaa nimenomaan minimaalisenvirittävän puun
- todistus sivuutetaan tässä mutta se löytyy sekä Karvin monisteesta että Cormenin kirjasta

7.5.2 Primin algoritmi

- myös Primin algoritmi käyttää ahnetta strategiaa mutta erona Kruskaliin on se että algoritmin suorituksen aikana on olemassa vain yksi palanen jota kasvatetaan vähitellen kokonaiseksi virittäväksi puuksi
- algoritmin syötteenä on suuntaamaton verkko G, kaaripainot määrittelevä funktio w ja jokin verkon solmu r josta virittävän puun muodostaminen aloitetaan
- solmuihin liittyy kaksi attribuuttia, avain $key[v]$ ja vanhempi $p[v]$
- solmun v avainkentän arvona on sen särmän paino minkä paino on pienin niiden särmien joukossa jotka yhdistävät v:n konstruktion alla olevaan virittävään puuhun
- algoritmin suoritusaikana muodostettavaan virittävään puuhun vielä kuulumattomia solmuja pidetään minimi-keossa H järjestettynä avainkentän mukaan
- algoritmin suorituksen jälkeen virittävän puun kaaret ovat $A = \{(v, p[v]) | v \in V - \{r\}\}$
- eli algoritmissa ei muodosteta eksplisiittisesti virittävän puun kaarijoukkoa vaan asia hallitaan solmujen p-attribuuttien avulla

```
Prim(G,w,r)
1   for kaikille solmuille v ∈ V do
2       key[v] ←∞
3       p[v] ←NIL
4   key[r] ←0
5   for kaikille solmuille v ∈ V do
6       heap-insert(H,v,key[v])
7   while not empty(H) do
8       u ←heap-del-min(H)
9       for jokaiselle solmulle v ∈ Adj[u] do
10          if solmu v on vielä keossa H ja w(u,v) < key[v] then
11              p[v] ←u
12              key[v] ←w(u,v)
13              heap-decrease-key(H,v,d[v])
```

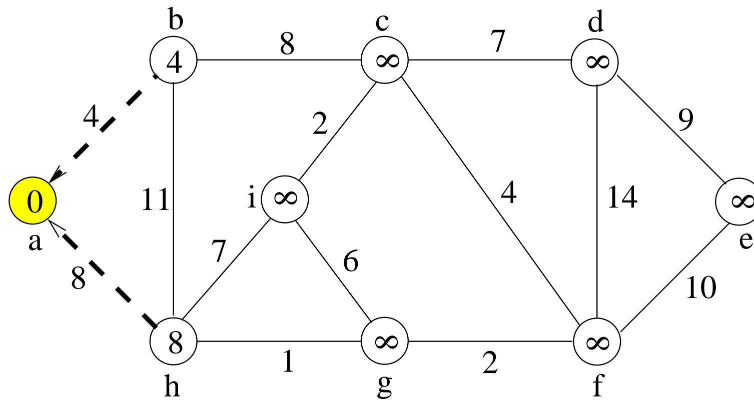
- algoritmin toimintaidea:
  - aluksi virittävä puu on tyhjä ja asetetaan kaikille solmuille paitsi r:lle avaimen arvoksi ääretön
  - ensimmäisessä toistossa tulee valituksi solmu r, joka siis poistuessaan keosta implisiittisesti liittyy virittävään puuhun
  - kun virittävään puuhun on näin liitetty solmu, pidetään ominaisuutta

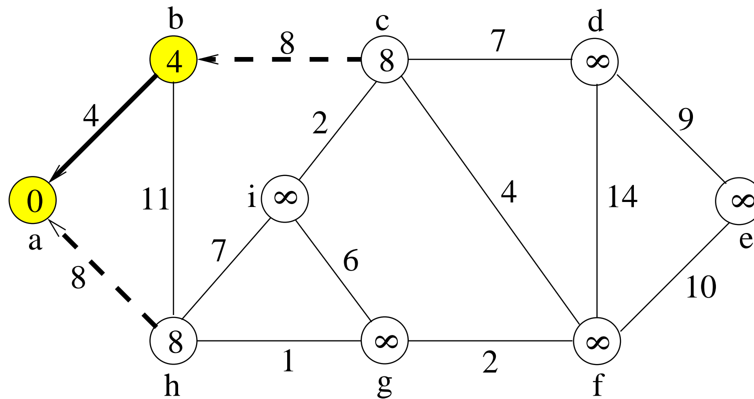
    kekoon kuulumattoman solmun v avainkentän arvona on sen särmän paino minkä paino on pienin niiden särmien joukossa jotka yhdistävät v:n konstruktion alla olevaan virittävään puuhun

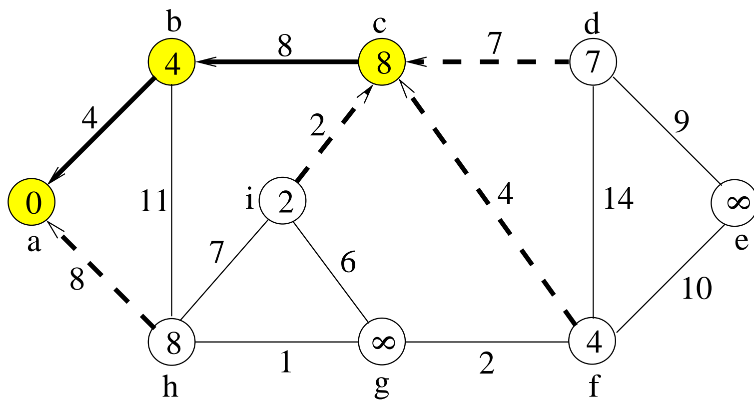
    voimassa

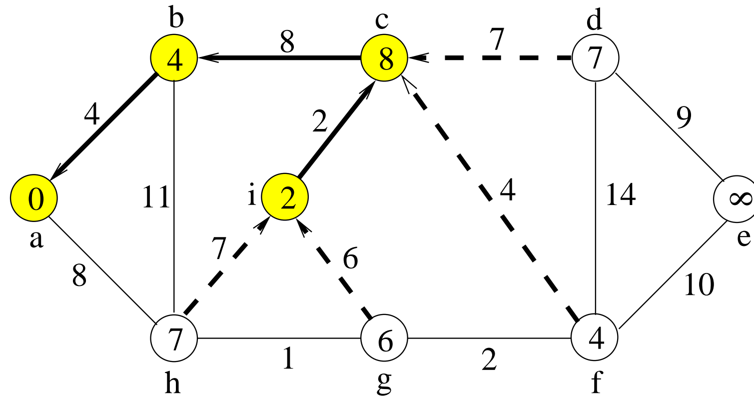
- tämän jälkeen otetaan keosta käsittelyyn solmu jonka avainarvo, eli etäisyys virittävään puuhun on pienin ja liitetään solmu virittävään puuhun
- edelleen huolehditaan että edellä mainittu ominaisuus virittävän puun ulkopuolisten solmujen avainkenttien arvoille on voimassa
- jatketaan niin kauan kun solmuja on keossa jäljellä

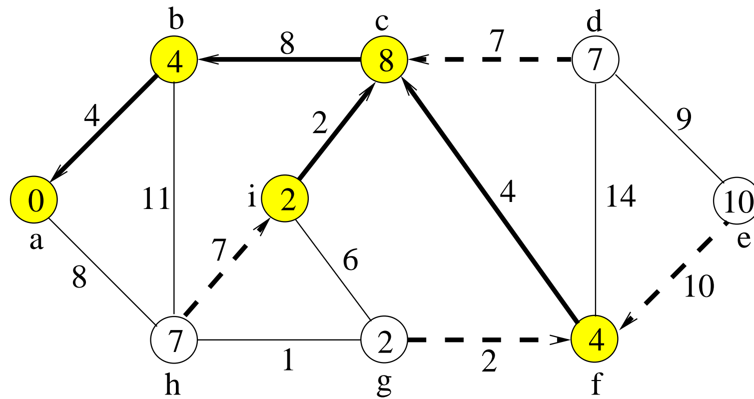
- esimerkki algoritmin toiminnasta:

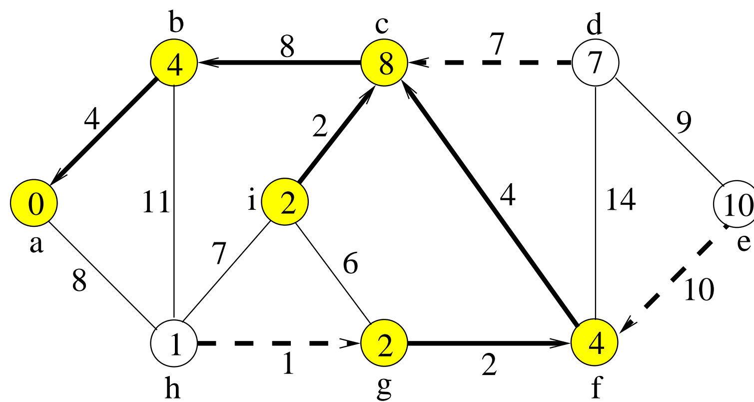


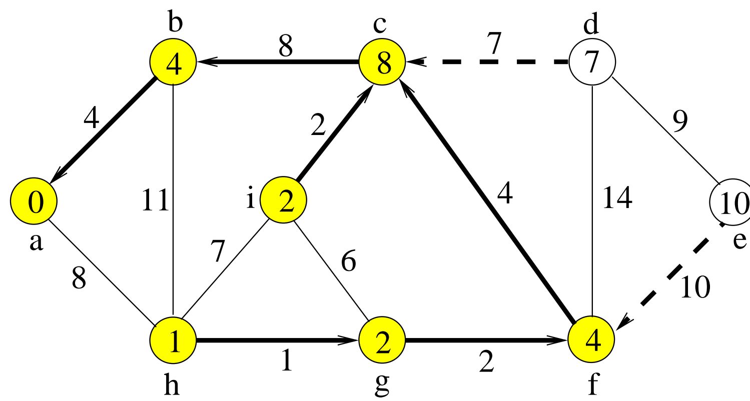


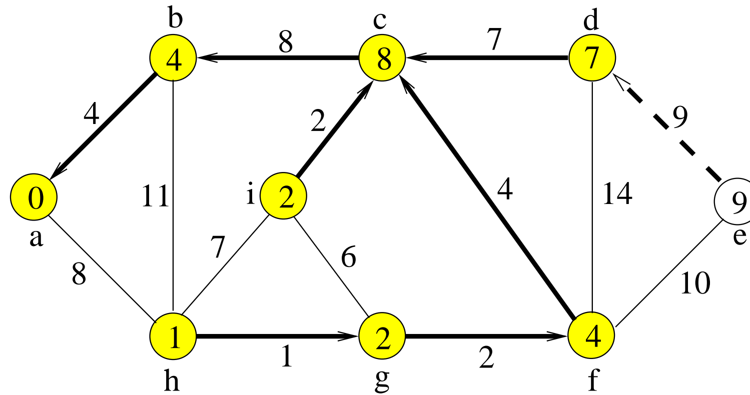

b
c
d
4
8
8
7
7
4
2
9
0
11
i
2
4
14
∞
a
e
8
7
6
10
8
∞
4
h
1
g
2
f

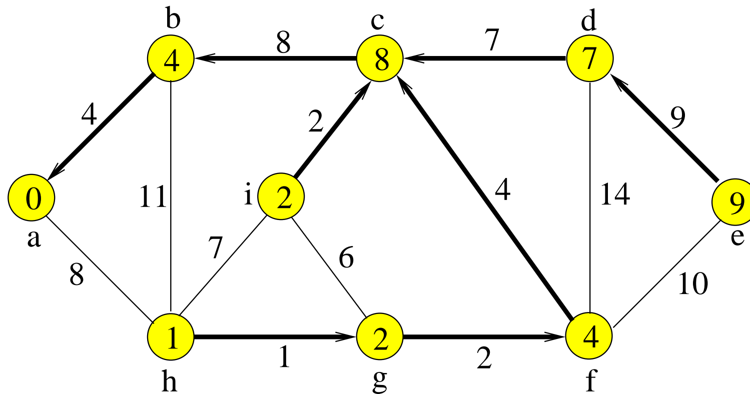

b
c
d
4
8
8
7
7
4
2
9
0
11
i
2
4
14
∞
a
e
8
7
6
10
7
6
4
h
1
g
2
f


b
c
d
4
8
8
7
7
4
2
9
0
11
i
2
4
14
10
a
e
8
7
6
10
7
2
4
h
1
g
2
f


b
c
d
4
8
8
7
7
4
2
9
0
11
i
2
4
14
10
a
e
8
7
6
10
1
2
4
h
1
g
2
f







- algoritmin vaativuus:
  - rivien 1-4 alkutoimet vievät aikaa $\mathcal{O}(|V|)$
  - riveillä 5-6 kutsutaan $|V|$ kertaa heap-insert operaatiota, eli aikaa kuluu $\mathcal{O}(|V|\ log\ |V|)$
  - rivien 7-13 toistolauseessa operaatio heap-del-min suoritetaan kertaalleen jokaiselle solmulle, ja tähän kuluu aikaa

$\mathcal{O}(|V|\ log\ |V|)$

- sisempi toistolause riveillä 9-13 suoritetaan $\mathcal{O}(|E|)$ kertaa sillä suuntaamattomassa verkossa kaikkien vieruslistojen yhteispituus on $|E|$
- sisemmässä toistolauseessa suoritetaan korkeintaan kertaalleen heap-decrease-key operaatio
- näin saamme Primin algoritmin aikavaativuudeksi $\mathcal{O}(|V|\ log\ |V| + |E|\ log\ |V|) = \mathcal{O}(|E|\ log\ |V|)$ eli sama kuin Kruskalin algoritmilla

- aputietorakenteena keko, ja alussa kaikki V solmua ovat keossa eli tilavaativuus $\mathcal{O}(|V|)$

• myöskään Primin algoritmin kohdalla ei ole aivan itsestään selvää että algoritmi tuottaa nimenomaan minimaalisen virittävän puun

• todistus sivuutetaan tässä mutta se löytyy sekä Karvin monisteesta että Cormenin kirjasta

7.5.3 Kruskal vai Prim?

• algoritmit näyttävät $\mathcal{O}$-analyysin valossa yhtä hyviltä

• Prim toimii käytännössä yleensä paremmin

• Primin algoritmia voidaan vielä nopeuttaa jos käytetään normaalin keon sijasta ns. Fibonaccin kekoa (missä decrease-key onnistuu vakioajassa), näin saatu Primin aikavaativuus on $\mathcal{O}(|E| + |V|\ log\ |V|)$