8.5.3.1 Lyhyimmän polun jäljittäminen

- Lisätään Floydin algoritmiin mahdollisuus jäljittää (jokin) lasketun pituinen $D[i][j]$ polku solmusta i solmuun j.
- Lisätään algoritmiin toinen matriisi $\Pi[1\ldots|V(G)|][1\ldots|V(G)|]$:
  - Samassa roolissa kuin solmun kenttä $v.\pi$ kalvoilla 8.4 ja 8.5.2.
  - Myös toteutettavissa lisäkenttinä $D[i][j].\pi$.
  - $\Pi[i][j] =$ se k jolla $D[i][j]$ sai nykyisen arvonsa.
  - Rivien 1–9 alustuksissa $k = 0$.
- Näillä lisätiedoilla voidaan jäljittää se, miten $D[i][j]$ sai arvonsa laskusäännöllä (32).

procedure PrintPath(i, j: $V(G)$)
  if $D[i][j] = +\infty$ **then**
    print "Ei ole!"
  else if $i = j$ **then**
    print i
  else
    PrintEdges(i, j)
  end if.

procedure PrintEdges(i, j: $V(G)$)
  if $\Pi[i][j] = 0$ **then**
    print $i \xrightarrow{D[i][j]} j$
  else
    PrintEdges($i, \Pi[i][j]$);
    PrintEdges($\Pi[i][j], j$)
  end if.

- Aikavaatimus on

  $$\mathcal{O}(\text{kaarten lukumäärä polulla}) = \mathcal{O}(|V(G)|)$$

  askelta, koska lyhyimmät polut nähtiin yksinkertaisiksi kalvoilla 8.5.3.

- $\Pi[i][j]$ voidaan määritellä hieman toisinkin:
  - Solmun j edeltäjäksi lyhyimmällä polulla solmusta i
    aliverkoissa G_k kun $k = 0, 1, 2, \ldots, |V(G)|$.
  - Eli oleellisesti samoin kuin kalvoilla 8.4 ja 8.5.2.
  - Rekursio yksinkertaistuu 1-haaraiseksi.
  - Rivien 3–7 alustuksessa

    $$\Pi[i][j] = \begin{cases} i & \text{kun } i \neq j \text{ ja} \\ & \quad i \xrightarrow{w} j \in E(G) \\ \text{NULL} & \text{muuten.} \end{cases}$$

  - Rivin 13 laskusäännössä

    $$\Pi[i][j] = \Pi[k][j]$$

    kun $D[i][j]$ pienenee.
  - Oikeellisuus vaatisi erillisen perustelun.

8.5.3.2 Warshallin algoritmi transitiiviselle sulkeumalle

- Painottamattoman verkon G *refleksiivinen transitiivinen sulkeuma* (reflexive transitive closure) on se verkko G' jonka

  solmut ovat samat $V(G') = V(G)$

  kaaret esittävät saavutettavuuden:

  $$i \to j \in E(G')$$

  jos ja vain jos

  verkossa G pääsee jotakin polkua pitkin solmusta i solmuun j.

- Intuitio: jos verkossa G on polku

  $$p \to q \to r$$

  niin lisätään suora *oikopolku*

  $$p \to r.$$

- Tämä "refleksiivisyys" ja "transitiivisuus" on sama kuin kalvoilla 8.5.2.2.

- Polun (35) tapaan nähdään:

  Jos on jokin polku solmusta i solmuun j, niin on myös yksinkertainen polku.

- Silloin voidaan käyttää kalvojen 8.5.3 algoritmia.

- Lasketaan lukujen sijaan pelkillä *totuusarvoilla*

  "onko $D[i][j] < +\infty$ vaiko ei?"

- Rivien 3–7 alustukset saavat muodon

  $$D[i][j] := B[i][j] \ \mathbf{or}\ \underline{(i = j)}.$$

- Ilman alleviivattua osaa tuloksena onkin transitiivinen muttei refleksiivinen sulkeuma.

- Rivin 13 laskusääntö saa muodon

  $$D[i][j] := D[i][j] \ \mathbf{or}\ (D[i][k] \ \mathbf{and}\ D[k][j]).$$

8.6 Virittävä puu

- Tarkastellaan esimerkkinä Suomen (julkista) rautatieverkkoa:
  - Se on yhtenäinen:
    * Miltä tahansa asemalta on jokin reitti kiskoja pitkin mille tahansa muulle asemalle.
    * Eli rautateitse voi matkustaa kaikkialle maassa.
  - Rataosuuksien ylläpito maksaa:
    * Halutaan lopettaa rataosuuksia säästösyistä.
    * pitäen silti rautatieverkko yhtenäisenä.
  - Tarvitaan siis kaikkien rataosuuksien osajoukko
    * joka edelleen yhdistää kaikki asemat toisiinsa
    * jonka ylläpitokustannukset ovat mahdollisimman pienet.

- Mallinnetaan rataverkko suuntaamattomana verkkona G jonka

  solmuina $V(G)$ ovat asemat

  särminä $p \overset{w}{—} q \in E(G)$ ovat sellaiset rataosuudet asemien p ja q välillä joiden välillä ei ole muita asemia r

  särmäpainoina w ovat vastaavan rataosuuden ylläpitokustannukset.

- Tämä verkko on *yhtenäinen* (connected):
  - Jokaisesta solmusta on jokin polku jokaiseen muuhun solmuun.
  - Suuntaamattoman verkon G yhtenäisyys on helppo varmistaa:
    1. Ajetaan kalvojen 8.4.1 kutsu DFSVisit(u) mielivaltaisesti valitulle solmulle $u \in V(u)$.
    2. Tarkistetaan että kaikki (muutkin) solmut $v \in V(G)$ mustuivat samalla.

    Perusteluna valkopolkulause 8.4.2 kutsun alussa.

- *Suunnatulla* verkolla puhutaan *vahvasta* (strong) yhtenäisyydestä:
  - se ei yleensä ole *kokonaan* vahvasti yhtenäinen
  - mutta siinä on vahvasti yhtenäisiä *komponentteja* (components) eli solmuvieraita aliverkkoja
  - jotka voidaan tunnistaa kalvojen 8.4.1 (kahdella) syvyyssuuntaisella läpikäynnillä
  - mutta algoritmi(t) ohitetaan tällä kurssilla

- Suuntaamattoman verkon G *virittävä puu* (spanning tree) on sellainen aliverkko $A \subseteq G$
  - jossa on kaikki solmut eli $V(A) = V(G)$
  - joka on edelleen yhtenäinen
  - jossa ei ole syklejä.

- *Eräs* virittävä puu on esimerkiksi yhtenäisyystestin tuottama π-puu.

- Kun verkon G kaaripainot ovat numeerisia, halutaan (kokonaispainoltaan) *pienin* (minimum(-weight)) virittävä puu:

  Sellainen jonka *kokonaispaino*

  $$\sum_{u \xrightarrow{w} v \in E(A)} w$$

  on mahdollisimman pieni.

- Näin on vaikkapa rataverkkoesimerkissämme.

- Verkon G pienin(kään) virittävä puu T ei ole 1-käsitteinen:

  – Jos särmä $p \xrightarrow{w} q \in E(T)$ pääsi puuhun T, mutta samanpainoinen särmä $u \xrightarrow{w} v \in E(G) \setminus E(T)$ ei, niin nämä särmät voidaan *vaihtaa keskenään*.

  – Ongelma ei kuitenkaan ole "valitse $|V(G)| - 1$ kevyintä särmää"...

8.6.1 Algoritmiskeema

- Lähdetään tarkentamaan algoritmiskeemaa

  1: Alusta (suuntaamattoman yhtenäisen numeerisesti painotetun) syöteverkon G aliverkko $A \subseteq G$ tyhjäksi;
  2: **while** A ei vielä ole mikään virittävä puu **do**
  3: Valitse jokin sellainen uusi särmä $u \xrightarrow{w} v \in E(G) \setminus E(A)$ joka on *turvallinen* (safe) aliverkolle A;
  4: Lisää valittu särmä $u \xrightarrow{w} v$ aliverkkoon A
  5: **end while.**

- Skeeman

  invariantti on

  "nykyinen A on syöteverkon G jonkin pienimmän virittävän puun T jokin aliverkko"

  rivin 3 turvallisuus on

  "myös rivillä 4 laajennettu A täyttää yhä invariantin".

- Skeema laskee mitä pitääkin:

  Alustus rivillä 1 asettaa invariantin triviaalisti voimaan:

  – Syöteverkolla G on virittäviä puita T, koska G on yhtenäinen ja suuntaamaton.

  – Tyhjä A on jokaisen T aliverkko, siis myös pienimpien.

  Valinta rivillä 3 takaa *implikaation*

  "*jos* rivin 3 alussa vanha A toteuttaa invariantin *niin* myös rivin 4 lopussa uusi A toteuttaa sen".

  Implikaation *jos*-osan takaa invariantti edellisen silmukkakierroksen lopussa.

  Lopputulos on silloin haluttu:

  – A on invariantin mukaan jonkin pienimmän virittävän puun T aliverkko.

  – A on itsekin kasvanut virittäväksi puuksi rivin 2 silmukkaehdon mukaan.

  – Siis $A = T$.

- Miten rivillä 3 valitaan seuraava särmä $u \xrightarrow{w} v \in E(G) \setminus E(A)$ turvallisesti mutta tehokkaasti?

- Jatkossa esitellään 2 algoritmia, jotka valitsevat hieman eri tekniikoilla:

  Primin algoritmi kalvoilla 8.6.2

  Kruskalin algoritmi kalvoilla 8.6.3.

- Määritellään joitakin apukäsitteitä:

  – Verkon G *halkaisu* (cut) on sen solmujen osajoukko $S \subseteq V(G)$ joka jakaa solmut *2 osaan* S ja $V(S) \setminus S$.

  – Särmä *ylittää* (crosses) halkaisun jos yksi sen päätepisteistä on yhdessä ja toinen toisessa osassa.

  – Halkaisu *kunnioittaa* särmäjoukkoa jos mikään joukon särmä ei ylitä halkaisua.

  – Halkaisun ylittävä särmä on *kevyt* (light) jos sen kaaripaino on minimaalinen kaikkien niiden kaarten joukossa jotka ylittävät halkaisun.

Lause 8.6.1. *Oletetaan:*

- *Olkoon G luvallinen syöte*

  eli yhtenäinen suuntaamaton numeerisesti painotettu verkko.

- *Olkoon nykyinen A verkon G jonkin pienimmän virittävän puun aliverkko*

  eli algoritmiskeeman tila on OK valintarivin 3 alussa.

- *Olkoon S aliverkon A särmiä kunnioittava verkon G halkaisu.*

- *Olkoon halkaisun S ylittävä särmä $u \overset{w}{—} v \in E(G)$ kevyt.*

Silloin särmä $u \overset{w}{—} v \in E(G)$ on turvallinen nykyiselle A

eli algoritmiskeeman tila on OK myös lisäysrivin 4 lopussa, kunhan valittiin jokin kevyt särmä.

Todistus.

- Olkoon T syöteverkon G jokin pienin virittävä puu jonka aliverkko A on.

- Jos T sisältää särmän $u \overset{w}{—} v$ niin hyvä: mitään ei tarvinnut osoittaa.

  Muuten meidän on luotava toinen pienin virittävä puu T' joka sisältää sekä aliverkon A että särmän $u \overset{w}{—} v$.

- Olkoon

  $$\mathcal{P} = u \overset{w_1}{—} \bigcirc \overset{w_2}{—} \bigcirc \overset{w_3}{—} \cdots \overset{w_m}{—} v$$

  puun T yksinkertainen polku, joka yhdistää solmut u ja v.

- Puussa T on tällaisia yhdistäviä polkuja, koska u ja v ovat sen verkon G solmuja, jonka pienin virittävä puu T on.

- Näiden yhdistävien polkujen joukossa on yksinkertaisia samalla argumentilla kuin yhtälössä (35).

- Yksinkertaisuus takaa, että polku $\mathcal{P}$ on myös *yksikäsitteinen:*
  - Olkoot $\mathcal{P}_1$ ja $\mathcal{P}_2$ 2 erilaista yhdistävää yksinkertaista polkua.
  - Kuljetaan silloin solmusta u polkuja $\mathcal{P}_1$ ja $\mathcal{P}_2$ pitkin ensimmäiseen solmuun p, jossa ne eroavat kulkemaan eri kaaria.
  - Jatketaan solmusta p polkua $\mathcal{P}_1$ pitkin ensimmäiseen solmuun q joka on jälleen myös polulla $\mathcal{P}_2$.

    (Viimeistään $q = v$.)
  - Saadaan sykli palaamalla solmusta q takaisin solmuun p kulkemalla tällä kertaa polkua $\mathcal{P}_2$ takaperin.
  - Puussa T ei saisi määritelmänsä mukaan olla syklejä.

- Koska solmut u ja v ovat halkaisun S eri osissa, on polulla $\mathcal{P}$ jokin särmä $x \overset{w_i}{—} y$ joka myöskin ylittää halkaisun S.
  - Särmäpainoille pätee $w \leq w_i$ koska särmä $u \overset{w}{—} v \in E(G)$ on kevyt.
  - Särmä $x \overset{w_i}{—} y$ ei kuulu aliverkkoon A, koska halkaisu S kunnioittaa aliverkkoa A.

- Jos puusta T poistetaan yksikin polun $\mathcal{P}$ kaarista, niin puu T
  - hajoaa kahteen sirpaleeseen joista
  - toinen sisältää solmun u
  - ja toinen solmun v

  koska yksikäsitteisyyden nojalla silloin katkeaa ainoa solmujen u ja v välinen yhteys.

- Puu T' saadaan puusta T korvaamalla särmä $x \xrightarrow{w_i} y$ särmällä $u \xrightarrow{w} v \in E(G)$:
  - sirpaleisuuden nojalla muutos ei luo syklejä
  - $A \subseteq T'$ koska puusta T poistuva särmä $x \xrightarrow{w_i} y$ ei kuulunut aliverkkoon A.
  - itse konstruktio lisää särmän $u \xrightarrow{w} v \in E(G)$ puuhun T'
  - kokonaispainokaan ei noussut, joten myös T' on syöteverkon G pienin virittävä puu

    (itse asiassa osoitimme $w_i = w$). □

8.6.2 Primin algoritmi

- Aliverkkoa A kasvatetaan *yhtenä puuna*.
- Kasvatus alkaa mielivaltaisesti valitusta alkusolmusta $s \in V(G)$.
- Kasvatusperiaate on

  "valitse aina sellainen särmä $u \xrightarrow{w} v \in E(G)$ jonka
  - ensimmäinen päätepiste u *on jo* aliverkossa A
  - toinen päätepiste v *ei vielä* ole aliverkossa A
  - paino w on tällaisista särmistä mahdollisimman pieni".
- Näin saatava algoritmi muistuttaa läheisesti Dijkstran algoritmia kalvoilla 8.5.2.

- Tehokkaan valinnan toteutus:
  - Pidetään mahdolliset toiset päätepisteet $v \notin V(A)$ kalvojen 5.4.3 (minimi)prioriteettijonossa H.
  - Päätepisteen v avain $v.\mathrm{dist}$ = pienin särmäpaino w jolla siihen tulee särmä $u \xrightarrow{w} v \in E(G)$ jostakin ensimmäisestä päätepisteestä $u \in V(A)$.

    Tai $+\infty$ jos sellaisia särmiä ei vielä ole.
  - Solmujoukko $V(A)$ = ne solmut $u \in V(G)$ jotka ovat jo poistuneet prioriteettijonosta H.

    Ei siis erillistä tietorakennetta.
  - Kaarijoukko

    $$E(A) = \left\{ v.\pi \xrightarrow{v.\mathrm{dist}} v : v \in V(G) \setminus \{s\} \right\}$$

    voidaan jälleen jäljittää lopuksi kentistä $v.\pi$ = se u jolta kenttä $v.\mathrm{dist}$ sai arvonsa.


```
procedure Prim(G: verkko, s: solmu)
 1: for all u ∈ V(G) \ {s} do
 2:    u.dist := +∞;
 3:    u.π := NULL
 4: end for;
 5: s.dist := 0;
 6: s.π := NULL;
 7: H := buildHeap(V(G));
 8: while not isEmptyHeap(H) do
 9:    u := heapExtractMinimum(H);
10:    for all u -w-> v ∈ E(G) do
11:       c := w;
12:       if v ∈ H and c < v.dist then
13:          v.dist := c;
14:          v.π := u;
15:          shiftup(H, v)
16:       end if
17:    end for
18: end while.
```

Ainoat muutokset Dijkstran algoritmiin kalvoilla 8.5.2 ovat riveillä 10–12.

- Primin algoritmi toimii oikein:
  - Olkoon halkaisu $S =$ ne solmut jotka ovat jo poistuneet työkeosta H

    eli nykyiset $V(A)$.
  - Algoritmi valitsee mahdollisimman kevyen särmän

    $$u.\pi \xrightarrow{u.\mathrm{dist}} u$$

    joka ylittää tämän halkaisun S
  - ja päivittää tietorakenteensa H vastaamaan tämän valinnan seurauksia.
  - Lauseen 8.6.1 mukaan tämä valinta on turvallinen
  - joten vedotaan algoritmiskeeman oikeellisuusperusteluun.
- Aikavaatimus on (parannuksineen) sama kuin Dijkstran algoritmilla kalvoilla 8.5.2.

8.6.3 Kruskalin algoritmi

- Aliverkkoa A kasvatetaan *palasista:*
  - Palaset *osittavat* syöteverkon G solmut $V(G)$ erillisiin osiin.
  - Solmut ovat samassa palasessa

    jos ja vain jos

    niiden välillä on polku nykyisessä aliverkossa A.
  - Algoritmi *yhdistää* aikaisemmin erilliset
    - solmun u palasen
    - solmun v palasen

    yhdeksi uudeksi palaseksi kun se lisää niiden väliin valitun särmän $u \xrightarrow{w} v \in E(G)$.
  - Syklit vältetään, kun lisätään vain särmiä palasesta toiseen.
  - Yhdistelyn alussa jokainen syötesolmu on yksinään omassa palasessaan.

- Yhdistelyperiaate on

  "valitse aina sellainen särmä $u \xrightarrow{w} v \in E(G)$ jonka
  - päätepisteet u ja v kuuluvat *eri* palasiin
  - paino w on tällaisista särmistä mahdollisimman pieni".
- Tehokkaan valinnan toteutus:
  - Järjestetään etukäteen syöteverkon G särmät painojensa mukaan jollakin tehokkaalla algoritmilla kalvoilta 5.
  - Käytetään tehokasta tietorakennetta
    - vastaamaan kysymykseen

      "kuuluvatko *ehdotetun* särmän $u \xrightarrow{w} v$ päätepisteet u ja v samaan vai eri palasiin?"
    - yhdistämään nämä palaset kun vastaus on "eivät kuulu"

      jolloin ehdotettu särmä voidaan valita.

    Tietorakenne kuvaillaan vasta kalvoilla 8.7.

1: $V(A)$:= $V(G)$;
2: $E(A)$:= $\emptyset$;
3: Alusta jokainen syötesolmu $u \in V(G)$ kuulumaan yksin omaan palaseensa;
4: Järjestä särmät $E(G)$ kasvavaan järjestykseen kaaripainojen suhteen;
5: **for all** särmä $u \xrightarrow{w} v \in E(G)$ edellä mainitussa järjestyksessä **do**
6:     **if** solmut u ja v kuuluvat eri palasiin **then**
7:         Lisää tämä särmä $u \xrightarrow{w} v \in E(G)$ aliverkkoon A;
8:         Yhdistä solmujen u ja v palaset yhdeksi palaseksi
9:     **end if**
10: **end for.**

- Kruskalin algoritmi toimii oikein:
  - Olkoon algoritmin suoritus aloittamassa lisäysriviä 7 valittuaan särmän $u \overset{w}{—} v$.
  - Olkoon lauseen 8.6.1 halkaisu $S =$ päätepisteen u palanen.
    * Kaikki muut syöteverkon G solmut laitetaan halkaisun toiselle puolelle.
    * Erityisesti päätepiste v joka ei kuulunut samaan palaseen kuin u.
    * Siis valittu särmä ylittää halkaisun S.
    * S kunnioittaa nykyistä aliverkkoa A:

      Jos päätepisteestä u pääsee johonkin solmuun x aliverkkoa A pitkin, niin x on samassa palasessa kuin u, eli samalla puolella leikkausta S kuin u.
  - Valittu särmä on halkaisun S kevyin:
    * Särmät joiden paino $< w$ on käsitelty
    * joten ne eivät kulje enää palasten välillä
    * joten ne eivät kulje yli halkaisun S.

- Rivin 4 järjestäminen vie aikaa

  $\mathcal{O}(|E(G)| \cdot \log |E(G)|) = \mathcal{O}(|E(G)| \cdot \log |V(G)|)$ askelta

  (koska $|E(G)| = \mathcal{O}(|V(G)|^2)$).
- Rivien 5–10 silmukka
  - pyörähtää nykyisellään täydet $|E(G)|$ kertaa.
  - voidaan katkaista heti kun rivin 7 lisäys on tehty $|V(G)| - 1$ kertaa
  - mutta silloinkin pyörähdyksiä voi tulla $\Theta(|E(G)|)$ kappaletta.
- Tavoitteena on, että rivien 5–10 silmukka ei olisi hitaampi kuin rivin 4 järjestäminen.
  - Silloin palasten käsittely riveillä 6–8 saa viedä $\mathcal{O}(\log |V(G)|)$ askelta operaatiota kohden.
  - Kalvoilla 8.7 tullaan pääsemään tähän...

    ...ja *vielä tehokkaampaan* ratkaisuun!

- Rivin 4 järjestäminen voidaan vielä upottaa rivien 5–10 silmukkaan:
  - Käytetään kalvojen 5.4 kekojärjestämisen inkrementaalisuutta.
  - Rivillä 4 vain alustetaan keko H särmistä $E(G)$ kalvojen 5.4.1.4 tapaan $\mathcal{O}(|E(G)|)$ askeleessa.
  - Rivillä 5 otetaan keosta H seuraava ehdokassärmä $u \overset{w}{—} v$ $\mathcal{O}(\log |E(G)|)$ askeleessa.
  - Kun silmukka katkaistaan $|V(G)| - 1$ lisäykseen, jää kekoon H vielä kaaria, joiden järjestäminen näin vältettiin.
  - Ei silti paranna $\mathcal{O}$-arviota.

8.6.4 Prim vai Kruskal?

- Primin algoritmi kalvoilta 8.6.2 ja Kruskalin algoritmi kalvoilta 8.6.3 näyttävät yhtä hyviltä, kun tarkastellaan niiden suoritusaikojen $\mathcal{O}$-arvioita.
- Silti *Prim*in algoritmi on käytännössä parempi.
- Muistin käyttö:
  - Primin algoritmi voi käyttää suoraan kalvojen 8.3 vieruslistaesitystä.
  - Kruskalin algoritmi järjestää särmiät kasvavan kaaripainon mukaan.
    * Tämä järjestys täytyy tallettaa jonnekin muualle kuin syöteverkon G muistiesitykseen
    * ellei muistiesitys ole lista kaikista särmistä

      (tai vastaava).

- Primin algoritmin aikavaatimuksen oletusarvoksi on osoitettu (sivuutetaan) vain

$$\mathcal{O}(|E(G)|)$$

askelta

  - kun syöteverkko G ei ole harva
  - ja särmäpainot arvotaan tietyllä tavalla.

- Tämä vastaa käytännön havaintoja:
  - Primin algoritmi on hyvä tiheillä satunnaisilla verkoilla
  - ja kilpailukykyinen vielä muissakin tilanteissa
  - uudempiakin kuin Kruskalin algoritmia vastaan.

- Primin algoritmia voi vielä nopeuttaa:
  - Vaihdetaan keoksi kalvojen 8.5.2.1 huomiossa 9 mainittu Fibonaccin keko.
  - Saadaan sama hyöty kuin kalvojen 8.5.2 Dijkstran algoritmissa.

8.7 Joukot ja yhdisteet

- Kruskalin algoritmi kalvoilla 8.6.3 johti seuraavanlaiseen tietorakenneongelmaan:
  - On kokoelma $a_1, a_2, a_3, \ldots, a_n$ *alkioita*.
  - Tehtävänä on pitää yllä tietoa näiden alkioiden *jaottelusta joukkoihin:*
    * Jokainen alkio kuuluu tasan yhteen joukkoon.
    * Jokaisessa joukossa on aina ainakin yksi alkio.
  - Tietorakanteen on pystyttävä reagoimaan

    kysymykseen "Kuuluvatko alkiot a_i ja a_j tällä hetkellä samaan vai eri joukkoon?"

    antamalla oikea vastaus

    komentoon "Ota alkioiden a_i ja a_j joukot ja yhdistä ne!"

    päivittymällä vastaavasti.

- Kruskalin algoritmissa

  alkioina ovat syöteverkon G solmut $V(G)$

  joukkoina ovat aliverkon $A \subseteq G$ palaset.

- Käytetään tietorakenteen pohjana *metsää*:
  - Esitetään jokainen alkio a omana solmunaan.
  - Liitetään jokaiseen solmuun kenttä $a.\pi =$ tämän solmun isäsolmu.
    * Jos tämä solmu a on itse juuri niin $a.\pi = a$.

      Tämä juuren esitystapa yksinkertaistaa hieman erikoistapausten käsittelyä algoritmeissa.
    * Toinen mahdollisuus merkitä solmu a juureksi olisi tuttu $a.\pi =$ NULL.
  - Puut ovat siis kuten kalvojen 8.4.1, 8.4.2, 8.5.2 ja 8.6.2 π-puut:
    * linkitys lapsesta isään
    * haarautumisaste rajoittamaton.

- Puiden ja joukkojen suhde:
  - Jokainen alkioiden joukko esitetään vastaavista solmuista koostuvana puuna.
  - Jokaista puuta (eli joukkoa) *edustaa* sen juurisolmu.

    Eli kun kysytään solmua (eli alkiota) vastaavaa puuta (eli joukkoa) niin *haetaan sen juuri*solmu.
  - Annettujen kahden solmun (eli alkion) u ja v (joukot eli) *puut yhdistetään* seuraavasti:
    1. Ensin haetaan niiden juurisolmut p ja q.
    2. Sitten tehdään puusta p puun q juuren uusi alipuu
    3. päivittämällä $p.\pi$ osoittamaan solmuun q.
  - Kukin solmu (eli alkio) on aluksi oman puunsa (eli joukkonsa) juuri.

- Puiden ja joukkojen suhteesta saadaan tietorakenteelle *perusoperaatiot:*
  - Juuren haulle

    function Find(r: **solmu**): **solmu**
      if $r.\pi = r$ **then**
        return r
      else
        return Find($r.\pi$)
      end if.
  - Puiden yhdistämiselle

    procedure Union(u, v: **solmu**)
      Find(u).π := Find(v).

    Esimerkiksi kutsu Union(a, a) toimii järkevästi valitsemallamme juuren esitytavalla.

    (Tämä operaatio tunnetaan myös nimellä 'merge'.)
  - Solmun alustukselle

    procedure MakeSet(u: **solmu**)
      $u.\pi$:= u.
- Tietorakennetta kutsutaankin nimellä *union-find*. (Tai 'merge-find'.)

- Näin saadaan Kruskalin algoritmiin palasten käsittelylle toteutus tietorakenteen perusoperaatioilla:

  Palasten alustus rivillä 3 on silmukka

  for all $u \in V(G)$ **do**
    MakeSet(u)
  end for;

  Palasten erillisyystesti rivillä 6 voidaan johtaa seuraavasti:
  - "solmut u ja v kuuluvat eri puihin"
  - "solmujen u ja v puilla on eri juuret"
  - Find(u) $\neq$ Find(v).

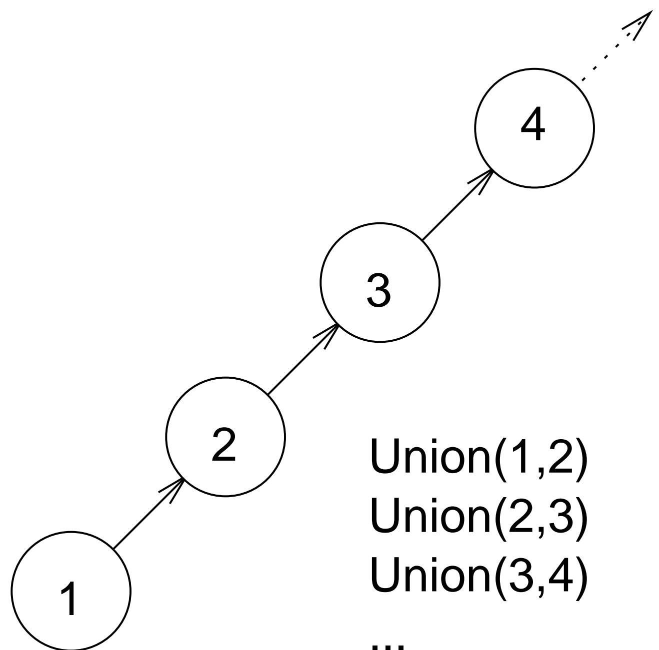
  Palasten yhdistäminen rivillä 8 on kutsu

  Union(u, v)

8.7.1 Yhdistäminen korkeuden mukaan

- Kalvojen 8.7 tietorakenteessa puut voivat kehittyä *epä*tasapainoisiksi.
  - Koska yhdistämisoperaatiossa aina jälkimmäinen parametri liitetään edelliseen.
  - Silloin jää saavuttamatta Kruskalin algoritmin kalvoilla 8.6.3 vaatima korkeintaan logaritminen suoritusaika operaatiota kohden.
- Lisätään siis tietorakenteeseen tasapainoehto.
  - Yhdistetään aina puista *matalampi korkeampaan*.
  - Tarvitaan juurisolmuihin lisäkenttä

    $r.\text{rank}$ = tämän puun korkeus
    = sen pisimmän polun pituus
    = pahin juuren hakuaika.



- Uudet tietorakenteen perusoperaatiot:
  - Juuren haku on sama kuin kalvoilla 8.7.
  - Puiden yhdistäminen
    * valitsee uuden juuren ja alipuun lisäkentän perusteella
    * päivittää uuden juuren lisäkenttää tarvittaessa.

procedure Union(u, v: **solmu**)
  Link(Find(u), Find(v)).

procedure Link(p, q: **solmu**)
  if p.rank $>$ q.rank **then**
    $q.\pi$:= p
  else if p.rank $<$ q.rank **then**
    $p.\pi$:= q
  else if $p \neq q$ **then**
    $p.\pi$:= q;
    q.rank := q.rank +1
  end if.

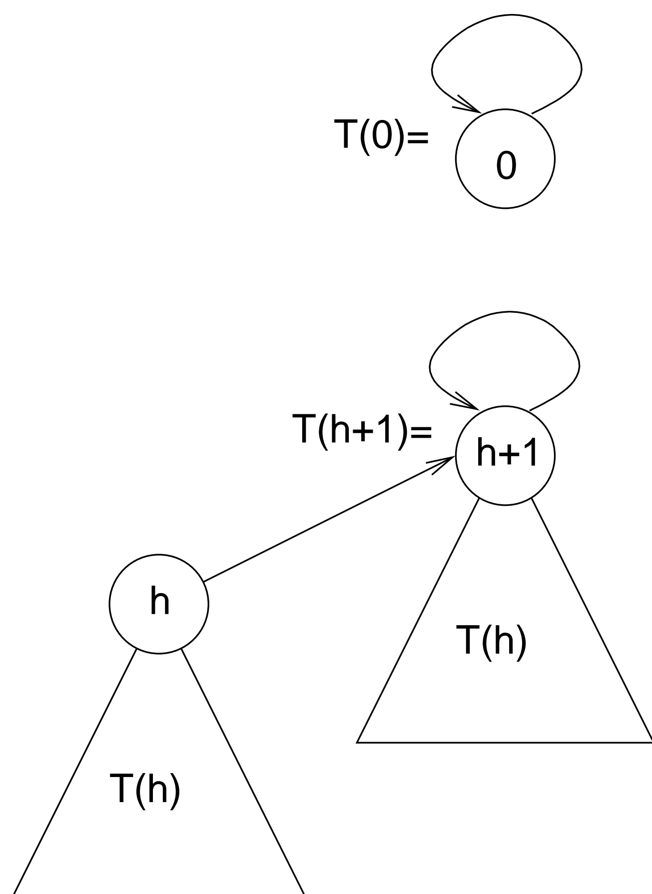
  - Alustetaan myös lisäkenttä:

procedure MakeRoot(u: **solmu**)
  $u.\pi$:= u;
  u.rank := 0.

- Tämä parannus saavuttaa Kruskalin algoritmin vaatiman logaritmisen suoritusajan:
  - Olkoon $T(h)$ = vähäsolmuisin puu jonka juuressa kenttä rank saa arvon h.
    * $T(0)$ = yksinäinen solmu.
    * $T(h+1)$ = kahden puun $T(h)$ yhdiste tällä periaatteella.
  - Puussa $T(h)$ on 2^h solmua:
    * Puussa $T(0)$ on $1 = 2^0$ solmua, kuten pitääkin.
    * Puussa $T(h+1)$ on $2\cdot$ niin monta solmua kuin puussa $T(h)$.

      Induktio-oletuksen mukaan puussa $T(h)$ on 2^h solmua.

      Siis puussa $T(h+1)$ on $2 \cdot 2^h = 2^{h+1}$ solmua, kuten pitääkin.
  - Siis n-solmuisessa puussa juuren haku vie $\mathcal{O}(\log n)$ kappaletta π-kentän seuraamisia.



8.7.2 Polun tiivistäminen

- Kalvojen 8.7.1 tasapainotukseen voidaan yhdistää vielä polun *tiivistäminen* (compression).
- Kun haetaan annetun solmun r juuri s, niin kuljetaan polku

$$r \xrightarrow{\pi} \underbrace{\bigcirc \xrightarrow{\pi} \bigcirc \xrightarrow{\pi} \cdots \xrightarrow{\pi} \bigcirc}_{\text{polun välisolmut}} \xrightarrow{\pi} s.$$

  - Voimme tallettaa vastaisen varalle *oiko*polun $r.\pi$:= s.
  - Tästä oikopolusta hyötyvät tulevaisuudessa ne juuren haut jotka alkavat
    * solmusta r itsestään
    * kaikista solmuista t joiden polku kulkee solmun r kautta.
  - Polun välisolmutkin hyötyvät kun nekin saavat oman oikopolkunsa silloin kun niiden kautta haetaan.

- *Lopulliset* tietorakenteen perusoperaatiot:
  - Juuren haku tiivistää kulkemansa polun:

    Rekursiivisesti lisäämällä oikopolut paluumatkalla

```
function Find(r: solmu): solmu
  if r.π ≠ r then
    r.π := Find(r.π);
  end if;
  return r.π.
```

Iteratiivisesti kulkemalla polku *kahdesti*

```
function Find(r: solmu): solmu
  s := r;
  while s.π ≠ s do
    s := s.π
  end while;
  while r.π ≠ s do
    t := r.π;
    r.π := s;
    r := t
  end while;
  return s.
```

  - Puiden yhdistäminen ja solmun alustus pysyvät samoina kuin kalvoilla 8.7.1.

- Nyt juuren kenttä s.rank = pisin polun pituus tässä puussa *ellei* polkuja olisikaan tiivistetty.
  - Koska lasketaan edelleen kuten kalvoilla 8.7.1.
  - Oikean arvon laskeminen olisi hankalaa.
  - Ja turhaa: Kenttää käytetään arviona vastaavan puun *koosta*.
- Tietorakenteen aikavaativuuden analysointi on vaikeaa ja ohitetaan tällä kurssilla:
  - Polun tiivistäminen voi muuttaa puurakennetta rajusti.
  - Yhden 'Find'-operaation tekemä tiivistystyö voi auttaa seuraavia operaatioita.

    Reiluuden nimissä päädytään tasoitettuun tarkasteluun kaikkien operaatioiden yli (kuten kalvoilla 7.4.1).
  - Kalvoilla 8.7.1 nähty puun tasapainoisuus takaa, että tehdystä työstä hyötyy hyvin moni solmu.

- Tarkastellaan seuraavanlaista tietorakenneoperaatioiden jonoa:
  - m on kaikkien operaatioiden lukumäärä.
  - $n \leq m$ on alkioiden eli 'MakeSet'-alustusoperaatioiden lukumäärä.
  - 'Link'-operaatioita on korkeintaan $n - 1$ kappaletta:
    * Union = 2 · Find +1 · Link.
    * $n - 1$ riittää vaikka kaikkien alkioiden liittämiseen samaan joukkoon (kuten Kruskalin algoritmissa kalvoilla 8.6.3).
- Tällaisen jonon suoritus vie

$$\Theta(m \cdot \alpha(m, n))$$

askelta, missä $\alpha(m, n)$ kasvaa *hyvin hitaasti:*

  - $\alpha(m, n) \leq 3$ kun $n < 2^{16} = 65\,536$.
  - $\alpha(m, n) \leq 4$ kun $n <$ alkeishiukkasten arvioitu määrä maailmankaikkeudessa.
  - Siis käytännössä vakioaika/operaatio.

- Tarkka määritelmä on

$$\alpha(m, n) = \min\left\{i \geq 1 : A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) > \log_2 n\right\}$$

missä

$$\begin{aligned} A(1, j) &= 2^j \quad \text{kun } j \geq 1 \\ A(i, 1) &= A(i-1, 2) \quad \text{kun } i \geq 2 \\ A(i, j) &= A(i-1, A(i, j-1)) \quad \text{kun } i, j \geq 2 \end{aligned}$$

on *Ackermannin* funktio

  - josta on kirjallisuudessa hieman erilaisia muotoja
  - joille on yhteistä *erittäin nopea* kasvaminen.
- Esimerkiksi

$$\begin{aligned} A(4, 1) &= A(2, 16) \\ &= A(1, A(2, 15)) \\ &= 2^{A(2,15)} \\ &= 2^{2^{A(2,14)}} \\ &= 2^{2^{2^{2^{2^{2^{2^{2^{2^{2^{2^{2^{2^{2^{2^{2}}}}}}}}}}}}}}} \end{aligned}$$