Tietotekniikan kisavalmennus-materiaalia

19. huhtikuuta 2006

Matti Nykänen
Tietojenkäsittelytieteen laitos
PL 68 (Gustaf Hällströmin katu 2b)
00014 Helsingin yliopisto

`matti.nykanen@cs.helsinki.fi`

Lähteet

1. T.H.Cormen, C.E.Leiserson, R.L.Rivest, C. Stein: *Introduction to Algorithms. Second Edition.* MIT Press, 2001.

   (Kansainvälisen tietotekniikkaolympialaistoiminnan `http://ioinformatics.org/` suosittelema peruslähde.)

2. N.Davies: *The Ancient Kindoms of Mexico.* Penguin, 1983.

   (Yksi esimerkkikuva kalvoille 5.4.5.)

3. L.R.Ford (Jr.), D.R.Fulkerson: *Flows in Networks.* Princeton University Press, 1962.

   (Kalvojen 9.4.3 virtauslaskennan vanha perusteos.)

4. M.R.Garey, D.S.Johnson: *Computers and Intractability.* W.H.Freeman and Company, 1979.

   (Kirja sellaisten ongelmien tunnistamisesta, joille luultavasti ei ole olemassa tehokasta ratkaisualgoritmia.)

5. J.Kleinberg, É.Tardos: *Algorithm Design.* Addison Wesley, 2005.

6. A.Levitin: *Introduction to the Design & Analysis of Algorithms.* Addison Wesley, 2003.

7. S.Russell, P.Norvig: *Artificial Intelligence: A Modern Approach. Second Edition.* Prentice Hall, 2003.

   (Lisämateriaalia kalvoille 7.3.)

8. S.S.Skiena, M.A.Revilla: *Programming Challenges: The Programming Contest Training Manual.* Springer-Verlag, 2003.

Tekstissä lähteisiin viitataan merkinnällä [lähteen numero, luvun tai sivujen numero sen sisällä].

Sisältö

1 Johdanto

- Tämä kalvosarja perustuu ohjelmointikilpailukirjaan [8].

- Kirjaan liittyy verkkopalvelin

  `http://www.programming-challenges.com/`

  jossa sen ongelmia voi ratkoa.

- Myös toista palvelinta

  `http://online-judge.uva.es/`

  voi käyttää.

  Siellä on enemmän tehtäviä, mutta se on hankalampi käyttää.

- Opiskelutapana onkin

  1. tutustua johonkin lukuun
  2. ratkoa sen tehtäviä.

1.1 Olympialaisista

- Itämeren alueen tietotekniikkaolympialaiset (Baltic Olympiad in Informatics, BOI)
  - 6-henkiset joukkueet
  - 2 peräkkäistä kisapäivää
  - aikaa 5h / päivä
  - 3 tehtävää / päivä (1h 40 min. / tehtävä), kaikki tehtävät samanarvoisia

- Kansainväliset tietotekniikkaolympialaiset (International Olympiad in Informatics, IOI)
  - 4-henkiset joukkueet
  - kisapäivien välissä lepopäivä.
  - muuten kuin (suurempi) BOI

- Palkintoja jaetaan: 1/12 kultaa, 1/6 hopeaa, 1/4 pronssia (ja loput 1/2 jäävät ilman).

- Algoritminen painotus (ohjelmointinäppäryyden sijaan):
  - Yksinkertaiset syöte- ja tulostusasut yms.
  - Testit on pyritty suunnittelemaan siten, että
    * yksinkertaisellakin ratkaisulla saa osan pisteistä
    * mutta täysiin pisteisiin tarvitaan tehokas ohjelma.
  - Kirjan ja kisatehtävien välillä on joitakin teknisiä eroja:
    * Kisassa käytetään nimettyjä syöte- ja tulos*tiedostoja*, kirjassa nimeämättömiä oletus*virtoja*.
    * Kisassa yksi tiedosto sisältää *yhden* testisyötteen, kirjassa voi tulla *monta* syötettä peräkkäin.

Eräajotehtävä kuten Datatähti-kisassa — tavallisin.

Avoin tehtävä:

- Testisyötteet annetaan etukäteen.
- Laske kysytty tulos haluamallasi tavalla:
  - kynällä ja paperilla (!)
  - nokkelalla algoritmilla nopeasti
  - yksinkertaisella algoritmilla tausta-ajona, kun mietit muita tehtäviä,...

  Vain ratkaisu pisteytetään, tapaa ei.

On-line tehtävä:

- Ohjelmasi "keskustelee" testausrobotin kanssa.
- Pelaa lautapeliä sitä vastaan tms.
- Pisteytys voi riippua keskustelun määrästä.

1. Aloita tutustumalla *kaikkeen* jaettuun materiaaliin!
   - Saat tehtävistä 2 versiota:

     englanninkielinen virallinen versio, joka pätee jos versiot poikkeavat toisistaan

     suomenkielinen käännös, joka on epävirallinen.
   - Voit pyytää täsmennyksiä tehtäviin
     - *vain ensimmäisen tunnin* ajan!
     - muotoiltuna *kyllä/ei*-kysymyksiksi.
2. Jatka kokeilemalla *testausrobottia:*
   - Voit lähettää sille ratkaisuehdotuksiasi testattavaksi.
   - Se palauttaa vastauksena testien tulokset.
   - *Viimeinen* ehdotuksistasi pisteytetään.
   - Pisteytys tehdään vasta *kisapäivän jälkeen eri syötteillä* kuin testaus!

1.2 Tehtävien ja ratkaisujen kokoluokista

- Tehtävistä näkee jonkin suureen $n \approx$ syötteen määrä.

  Esimerkiksi syöterivien lukumäärän.
- Nykyaikaisilla kilpailukoneilla voi tehdä suunnilleen muutaman *miljoonan* $\approx 2^{20}$ *perusoperaatiota* / CPU-sekunti:
  - Perusoperaatiot riippuvat tehtävästä: silmukkakierrokset, rekursiokutsut,...
  - Kellotaajuus on useita satoja ...muutama tuhat MHz eli miljoonia käskyjä / CPU-sekunti.
  - Ohjelmakoodi sisältää kymmeniä ...satakunta käskyä / perusoperaatio.
- Näistä voidaan arvioida, kuinka tehokasta algoritmia tehtävän laatija hakee.

Jos suureen n koko on joitakin...

kymmeniä (esim. $n \approx 20$), niin raakaan laskentavoimaan perustuvaa *kaikkien vaihtoehtojen läpikäyntiä*.

Perusmenetelmiä ja tehostuksia kalvoilla 7.

satoja (esim. $n \approx 100$), niin *kuutiollista* $O(n^3)$ algoritmia.

Esimerkiksi kalvot 9.1.2 missä $n =$ solmujen määrä.

tuhansia (esim. $n \approx 1\,000$), niin *neliöllistä* $O(n^2)$ algoritmia.

Esimerkiksi kalvojen 3 hidasta järjestämistä.

kymmeniä tuhansia (esim. $n \approx 2^{16} \approx 65\,000$), niin $O(n \cdot \log_2(n))$ algoritmia, kuten...

- kalvojen 3.2 nopeaa järjestämistä
- kalvojen 2.3 tehokkaiden kekojen käyttöä
- kalvojen 10.1.1 binäärihaun käyttöä.

- Merkintä

  O(suureen n lauseke)

  tarkoittaa intuitiivisesti

  > "Algoritmi tekee *oleellisesti* tämän verran perusoperaatioita, kun sille annetaan syöte jonka koko on n."

- Se häivyttää näkyvistä
  - laite- ja ohjelmointikielikohtaiset erot yhden perusoperaation kestossa
  - käyttäytymisen pienillä n, joilla alustuksiin saattaa kulua suuri osa ajasta
  - algoritmin muiden kuin hitaimman vaiheen vaikutukset.

- Tarkempi määritelmä sivuutetaan.

  [1, luku 3.1; 6, luvut 2.1 ja 2.2]

2 Tietorakenteista

[8, luku 2]

- Jos ohjelmointikielellä on standardoitu tietorakennekirjasto kuten

  C++ Standard Template Library (STL)

  Java pakkaus `java.util`

  niin käytä sitä!

- Muuten tietorakenteet joutuu toteuttamaan kisatilanteessa käsin.

- Suoraviivainen toteutus taulukolla on usein helpointa.

  Dynaamisen muistin varaus ja vapautus käsin on virhealtista.

- Tavallisesti kisatehtävistä "näkee" suureen $M =$ montako tietuetta tietorakenteeseen pitää enintään mahtua kerrallaan.

- Suure M riippuu syötteen koosta.

- Esitellään, miten perustietorakenteet

  pino (kalvot 2.1)

  jono (kalvot 2.2)

  keko (kalvot 2.3)

  voidaan toteuttaa M-paikkaisella taulukolla A.

2.1 Pinot

[1, luku 10.1]

- *Pino*lla (stack) on 1 pää, johon kohdistuvat kaikki perusoperaatiot:
  - uuden tietueen *lisäys* eli "PUSH"
  - vanhan tietueen *poisto* eli "POP"
  - päällimmäiseen tietueeseen *kurkistaminen* eli "TOP"

  Lisäksi voidaan kysyä, onko pino *tyhjä* vai ei.

- Viimeiseksi lisätty tietue poistuu ensimmäisenä.

  (**L**ast **I**n, **F**irst **O**ut)

- Esimerkki: lautaspino astiakaapissa.
  - Kattaukseen otetaan aina päällimmäinen lautanen.
  - Astianpesukoneesta laitetaan uusi lautanen aina päällimmäiseksi.
  - Vain päällimmäisestä lautasesta näkee, onko se pesty kunnolla.
- Toteutus:
  - Taulukkona on $A[1 \ldots M]$.
  - Lisäksi muuttuja koko muistaa pinon *syvyyden*, eli siinä nyt olevien tietueiden lukumäärän

    Jos koko $= 0$, niin pino on tyhjä.
  - Tietueet talletetaan *takaperin:*
    1. päällimmäinen on A[koko]
    2. sen alla on A[koko -1]
    3. ...

Lisäys:

koko **:=** koko $+1$**;**
A[koko] **:=** lisättävä tietue t**.**

Poisto:

koko **:=** koko -1**.**

2.2 Jonot

[1, luku 10.1]

- *Jono*lla (queue) on 2 päätä:

  alkupää josta otetaan tietueita pois

  loppupää johon lisätään uusia tietueita.
- Tietueet poistuvat jonosta saapumisjärjestyksessä.

  (**F**irst **I**n, **F**irst **O**ut)
- Esimerkki: kassajono kaupassa.
- Voidaan sallia myös
  - lisäykset alkupäähän
  - poistot loppupäähän

  jolloin saadaan *pakka* (double-ended queue, dequeue).

- Taulukkona on $A[0 \ldots M - 1]$.
- Indeksointiperiaate:
  - $A[i]$ tehdäänkin $A[i \bmod M]$
  - eli *jakojäännös* pituuden M suhteen ("modulo M")
  - eli taulukko onkin *kehä*, jossa "viimeisen" paikan $A[M-1]$ jälkeen onkin jälleen "ensimmäinen" paikka $A[0]$.
- Käytetään kahta apumuuttujaa:
  - alku $=$ jonon ensimmäisen tietueen indeksi
  - koko $=$ jonossa olevien tietueiden lukumäärä.
- Jonon alkiot ovat peräkkäin taulukkopaikoissa

  $$A[\text{alku}], A[\text{alku}+1], A[\text{alku}+2], \ldots, A[\text{alku}+\text{koko}-1] \qquad (\bmod M)$$

  (siis tyhjällä jonolla koko $= 0$).

- Ensimmäisen tietueen A[alku] poisto:

  alku := (alku + 1) mod M;
  koko := koko − 1.

- Viimeisen tietueen A[(alku + koko − 1) mod M] perään lisäys:

  koko := koko + 1;
  A[(alku + koko − 1) mod M] := lisättävä tietue t.

- Ensimmäisen tietueen eteen lisäys:

  if alku = 0 **then**
  alku := $M - 1$
  else
  alku := alku − 1
  end if;
  koko := koko + 1;
  A[(alku] := lisättävä tietue t.

- Viimeisen tietueen poisto:

  koko := koko − 1.

2.3 Keot

- Tietorakenne *keko* (heap) eli *prioriteettijono* (priority queue) [1, luvut 6.1–6.3; 6, luku 6.4] on sellainen tietuevarasto
  - jossa tietueiden t avainkenttien avain[t] välillä on jokin *järjestys*relaatio $\leq$
  - josta järjestyksessä *pienimmän* tietueen löytäminen ja poisto on nopeaa.
- Esimerkki: päivystyspoliklinikka, jossa potilaat hoidetaan *kiireellisyys*järjestyksessä.
- Yksinkertainen *binääri*keko (binary heap) on
  - taulukko $A[1 \ldots M]$
  - josta on *käytössä alkuosa* A[1 . . . koko] (siis tyhjällä keolla koko = 0)
  - jossa vallitsee *kekoehto* (heap property)

  $$\text{avain}[A[i]] \leq \min(\text{avain}[A[\text{vasen}(i)]], \text{avain}[A[\text{oikea}(i)]])$$

  kaikkialla (eli indekseillä $1 \leq i \leq$ koko).

- Tarvittavat apulausekkeet ovat

$$\text{vasen}(i) = \begin{cases} 2 \cdot i & \text{kun } 2 \cdot i \leq \text{koko} \\ \text{NULL} & \text{muuten} \end{cases}$$

$$\text{oikea}(i) = \begin{cases} 2 \cdot i + 1 & \text{kun } 2 \cdot i + 1 \leq \text{koko} \\ \text{NULL} & \text{muuten} \end{cases}$$

$$\text{pappa}(i) = \begin{cases} \left\lfloor \frac{i}{2} \right\rfloor & \text{kun } i > 1 \\ \text{NULL} & \text{muuten} \end{cases}$$

  (siis alaspäin pyöristäen).

- Apulausekkeet voitaisiin johtaa
  - ajattelemalla kekoa *binääripuuna*
  - joka on toteutettu taulukkona A.

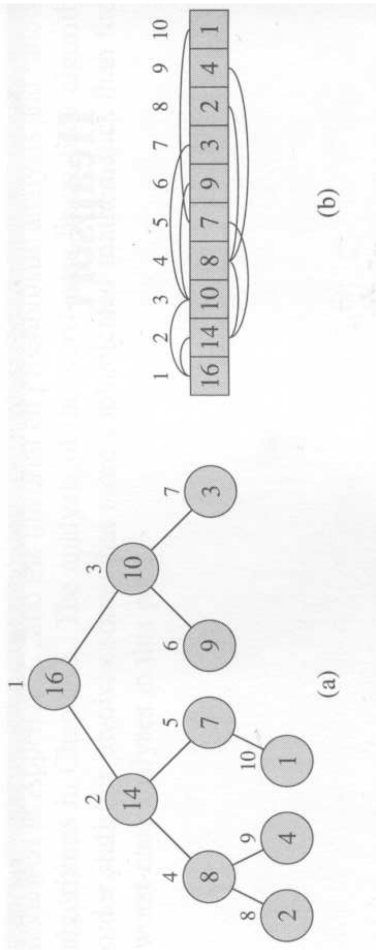
  [1, kuva 6.1]

- Apulausekkeet "hyppäävät" taulukossa A aina *kaksi kertaa kauemmas:*

$$1 \xrightarrow{\text{vasen}} 2 \xrightarrow{\text{vasen}} 4 \xrightarrow{\text{vasen}} 8 \xrightarrow{\text{vasen}} \cdots$$

  Siitä tehokkuus: tarvittavan hyppyjonon pituus on

$$O(\log_2(\text{koko})) \quad \text{askelta.}$$



2.3.1 Pienimmän tietueen haku ja poisto

- Pienin tietue on taulukkopaikassa $A[1]$ kekoehdon nojalla.

- Taulukkopaikassa $A[i]$ olevan alkion poisto:
  procedure delete(i: **indeksi**)
    if $i =$ koko **then**
      koko **:=** koko -1**;**
    else
      vaihda(i, koko)**;**
      koko **:=** koko -1**;**
      heapify(i)
    end if.

- Aliohjelma heapify(i)

  olettaa aluksi että kekoehto on voimassa indeksistä i vasemmalle ja oikealle mentäessä

  takaa lopuksi että kekoehto on voimassa myös indeksissä i.

  Se saadaan ohjelmoimalla kekoehdon rekursiivinen määritelmä.

procedure heapify(i: **indeksi**)
  l **:=** vasen(i)**;**
  r **:=** oikea(i)**;**
  if $l \neq$ NULL **and** avain[$A[l]$] $<$ avain[$A[i]$]
  then
    j **:=** l
  else
    j **:=** i
  end if;
  if $r \neq$ NULL **and** avain[$A[r]$] $<$ avain[$A[j]$]
  then
    j **:=** r
  end if;
  if $j \neq i$ **then**
    vaihda(i, j)**;**
    heapify(j)
  end if.

- Aliohjelma vaihda(i, j) vaihtaa taulukkopaikkojen $A[i]$ ja $A[j]$ sisällöt keskenään.

- Rekursiokutsu heapify(j) voidaan ohjelmoida myös **while**-silmukalla, koska se tehdään *viimeisenä*.

2.3.2 Uuden tietueen lisäys ja keon luonti

- Uusi tietue t lisätään kekoon tekemällä kalvojen 2.3.1 poisto *toisin päin:*

    koko **:=** koko $+1$**;**
    A[koko] **:=** t**;**
    siftup(koko)**.**

  procedure siftup(i: **indeksi**)
    j := pappa(i);
    if $j \neq$ NULL **and** avain[$A[i]$] $>$ avain[$A[j]$]
    then
      vaihda(i, j)**;**
      siftup(j)
    end if.

- Tietueista $t_1, \ldots, t_n$ saadaan keko seuraavasti:

  1. Varaa tarpeeksi suuri keko $A[1 \ldots n]$, joka on aluksi tyhjä.
  2. Lisää kekoon kukin tietue t_k.

  $$O(n \cdot \log_2(n)) \quad \text{askelta.}$$

- On nopeampikin tapa:

    Varaa tarpeeksi suuri taulukko $A[1 \ldots n]$**;**
    Kopioi tietueet $t_1, \ldots, t_n$ taulukkoon A**;**
    koko **:=** n**;**
    for k **:=** n **down to** 1 **do**
      heapify(k)
    end for.

- Voidaan laskea (ei ihan helposti) että vain

  $$O(n) \quad \text{askelta.}$$

- Kalvojen 2.3.1 aliohjelmaa heapify(k) ei tarvitse kutsua, jos vasen(k) $=$ oikea(k) $=$ NULL.

  Siis **for**-silmukan voi aloittaa vasta arvosta

  $$k := \left\lceil \frac{n}{2} \right\rceil.$$

2.3.3 Kahvat keon tietueisiin

- Joskus on päästävä käsiksi myös *muihin* keossa oleviin tietueisiin t kuin vain pienimpään.
- Hankaluuksia:
  - Keko ei ole kokonaan järjestyksessä, joten tietueen t etsintä olisi hidasta.
  - Tietueen t indeksi keossa muuttuu keko-operaatioiden myötä.
- Eräs yksinkertainen ratkaisu:
  - Itse tietueet t talletetaankin kekotaulukon *ulkopuolelle*, jolloin niihin pääsee käsiksi.
  - Kekotaulukon paikkaan $A[i]$ talletetaankin pelkkä *viite* (osoitin tms.) vastaavaan tietueeseen t.
  - Tietueeseen t lisätään uusi kenttä kahva[t] (handle) [1, luku 6.5].

- Kenttä kahva[t] =

  joko se kekotaulukon A indeksi j, jossa tietue t tällä hetkellä on

  tai NULL, jos tietue t ei tällä hetkellä ole keossa.

- Muistisääntö:

$$\text{kahva}[A[i]] \equiv i$$

- Kahvoja ylläpidetään, kun
  - kutsutaan aliohjelmaa vaihda(i, j):

    vaihdetaan myös kenttien

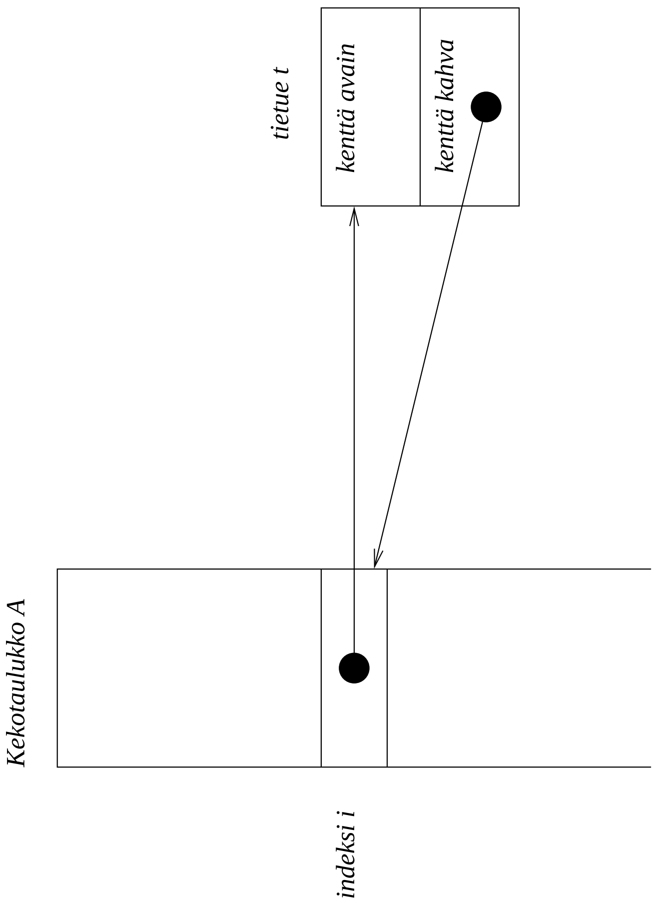
$$\text{kahva}[A[i]] = i$$
$$\text{kahva}[A[j]] = j$$

    sisällöt keskenään.

  - keon koko muuttuu:

    lisäyksessä kahva[A[koko]] = koko

    poistossa kahva[A[koko]] = NULL.



- Kentän kahva[t] avulla voi

  pienentää kenttää avain[t]:

  siftup(kahva[t]) kalvoilta 2.3.2.

  suurentaa kenttää avain[t]:

  heapify(kahva[t]) kalvoilta 2.3.1.

  poistaa koko tietueen t keskeltä kekoa:

  delete(kahva[t]) kalvoilta 2.3.1.

2.3.4 Pidentyvä taulukko

[1, luku 17.4]

- Jos kalvojen 2.3.2 lisäyksessä
  - onkin jo koko $= M$
  - eli taulukko A onkin jo täynnä?
- *Kaksinkertaistetaan* ensin *dynaamisesti* varatun taulukon pituus $M \geq 0$:

  varataan dynaamisesti uusi taulukko $B[1 \ldots \underline{2 \cdot M}]$

  kopioidaan vanha taulukko $A[1 \ldots M]$ uuden taulukon B alkuun

  asetetaan taulukon alkuosoittimen uudeksi arvoksi B.

  vapautetaan taulukon alkuosoittimen vanhan arvon A osoittama dynaamisesti varattu muisti.

- *Yksittäinen* pino-operaatio voi silloin hidastua

  $O(M)$ askeleeseen.

- *Koko ohjelman* suorituksessa näkyy hidastusta vain

  $O(1)$ askelta/pino-operaatio

  koska jokainen hidastunut operaatio tekee työn, joka hyödyttää kaikkia sen jälkeen tehtäviä operaatioita.

- Kisatehtävissä ei yleensä tarvitse *lyhentää* taulukkoa, vaikka se alkaisikin tyhjentyä.

  Jos tarvitsee, niin silloin käänteinen periaate: Taulukko *puolitetaan* vasta kun se on enää $\frac{1}{4}$ *täynnä*.

- Samaa ideaa voi käyttää muissakin taulukkopohjaisissa tietorakenteissa.

3 Järjestämisestä

- Ohjelmointikielten vakiokirjastoissa on usein järjestämisalgoritmeja valmiina:

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

  Niitä kannattaa käyttää!

- Tutustutaan muutamaan perusalgoritmiin siltä varalta että kisatehtävässä on hyödyllistä
  - käyttää
  - muokata

  juuri tiettyä algoritmia [8, luku 4].

- Olkoon järjestettävä aineisto taulukkona

  $A[0 \ldots N - 1]$.

- Käytetään aliohjelmaa vaihda(i, j) kalvoilta 2.3.1.

3.1 Yksinkertaisia menetelmiä

- Esitellään 2 yksinkertaista järjestämismenetelmää:
  - valintajärjestäminen (kalvoilla 3.1.1)
  - sijoitusjärjestäminen (kalvoilla 3.1.2)
- Ne vievät

  $O(N^2)$ askelta.

  Kalvoilla 3.2 esitellään niitä nopeampi menetelmä.

- Näillä menetelmillä on kuitenkin muita hyviä piirteitä.

  Esimerkiksi ne ovat *vakaita* (stable): yhtä suurten syötetietueiden keskinäinen järjestys pysyy samana.

- Molemmat menetelmät käyttävät seuraavaa perusideaa:
  - Taulukon alkuosa
    $$A[0 \ldots i-1]$$
    on jo järjestyksessä.
  - Taulukon loppuosa
    $$A[i \ldots N-1]$$
    koostuu niistä tietueista, jotka täytyy vielä järjestää.
  - Joka silmukkakierroksella loppuosasta valitaan jokin tietue
    $$A[j]$$
    joka viedään oikealle paikalleen alkuosaan.
  - Näin alkuosa kasvaa
    $$i \mapsto i+1$$
    ja loppuosa lyhenee yhdellä tietueella.

- Esimerkki hyödyllisestä tavasta varmistua silmukan oikeellisuudesta *invariantilla:*
  - Jollakin (oleellisella) väitteellä ϕ, jonka silmukka toteuttaa.
  - Hyödyllisiä myös muistisääntöinä!

  [1, luku 2.1]
- Nyt ϕ on taulukon jako alku- ja loppuosaan.
- Silmukan

  alussa ϕ on totta alustuksen vuoksi.
  Nyt koska alustetaan $i = 0$.

  aikana ϕ pidetään totena kirjoittamalla silmukan runko sen mukaisesti.
  Nyt koska $A[j]$ käsitellään sopivasti ennen kuin i kasvaa.

  lopussa ϕ on yhä totta, kun loppuehto laukeaa.
  Nyt loppuehto on $i = N$, ja siis koko syötetaulukko A on saatu lajiteltua.

3.1.1 Valintajärjestäminen

- *Valintajärjestäminen* (selection sort) [6, luku 3.1]

  valitsee loppuosan *pienimmän* tietueen

  siirtää sen alkuosan *perään*.
- *Ei vaihda turhaan* tietueita keskenään.
- Jos sisäsilmukkaa tehostetaan kalvojen 2.3 kekotietorakenteella, niin saadaan nopea *kekojärjestäminen* (heapsort) [1, luku 6.4; 6, sivut 223–224].

  Tosin vakaus menetetään.

```
for i := 0 to N − 1 do
  j := i;
  for k := i + 1 to N − 1 do
    if A[k] < A[j] then
      j := k
    end if
  end for;
  vaihda(i, j)
end for.
```

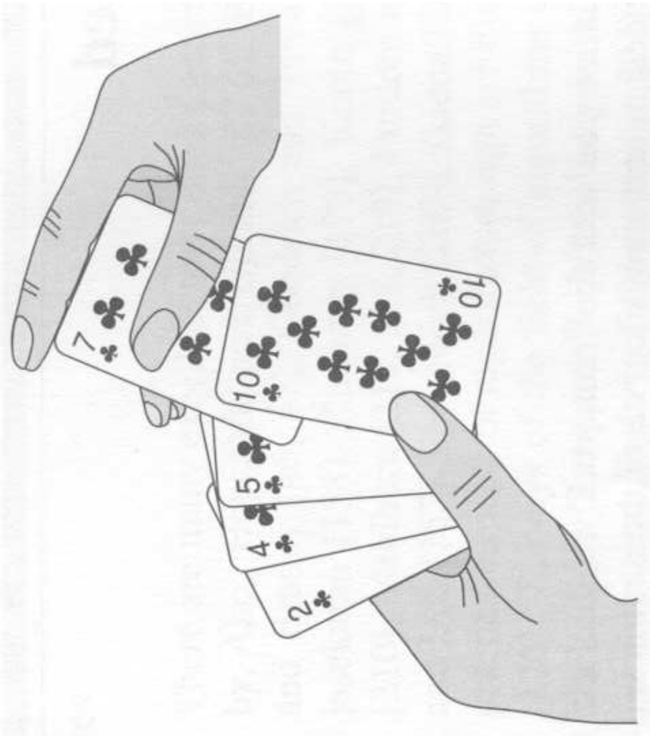

3.1.2 Sijoitusjärjestäminen

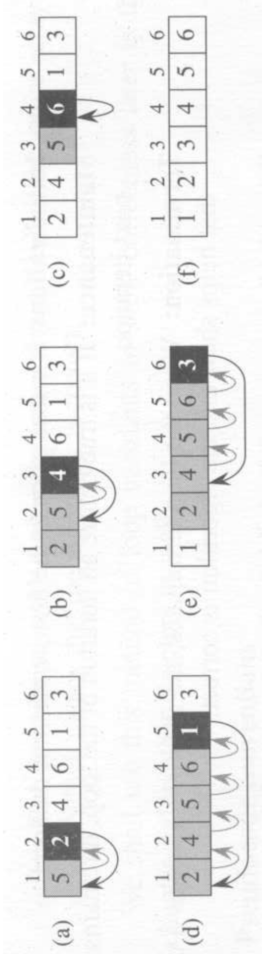
- *Sijoitusjärjestäminen* (insertion sort) [1, luku 2.1 sekä kuvat 2.1 ja 2.2; 6, luku 5.1]

  valitsee loppuosan *ensimmäisen* tietueen

  siirtää sen alkuosan sopivaan *väliin*.
- *Ei valitse turhaan* tietueita.
- Tehokas, jos A on *melkein* järjestyksessä.

```
for i := 0 to N − 1 do
  j := i;
  while j > 0 and A[j] < A[j − 1] do
    vaihda(j, j − 1);
    j := j − 1
  end while
end for.
```





3.2 Pikajärjestäminen

- *Pikajärjestäminen* (quicksort) [1, luku 7.1; 6, luku 4.2] on *käytännössä tehokkain* ja *yleisin* järjestämismenetelmä.

- Se vie

  tyypillisesti

  $O(N \cdot \log_2(N))$ askelta

  joka on *paras mahdollinen!*

  pahimmillaan

  $O(N^2)$ askelta.

- Esimerkiksi kalvoilla 3.1.1 mainittu kekojärjestäminen vie pahimmillaankin vain

  $O(N \cdot \log_2(N))$ askelta

  mutta on käytännössä

  – hitaampi

  – vaikeampi ohjelmoida.

- Pikajärjestäminen saadaan kalvojen 10.1 hajoita ja hallitse -periaatteella.

- Syötetaulukon $A[p \ldots r]$ jako osiin:

  – Vaihdellaan sen alkioita keskenään

  – kunnes on olemassa sellainen indeksi $p \leq q \leq r$ jolla pätee:

  – alkuosan $A[p \ldots q-1]$ alkiot ovat *korkeintaan* yhtä suuria kuin *jako-* eli *napa-alkio* (pivot) $A[q]$

  – loppuosan $A[q+1 \ldots r]$ alkiot ovat *vähintään* yhtä suuria kuin $A[q]$.

  Sitten lajitellaan rekursiivisesti alku- ja loppuosa.

- Rekursio ei tarvitse jälkikäsittelyä:

  – $A[q]$ on jo oikealla paikallaan

  – alkuosa järjestyy paikallaan, samoin loppuosa.

procedure quicksort(p, r: **indeksi**)

```
if p < r then
    q := partition(A, p, r);
    quicksort(A, p, q − 1);
    quicksort(A, q + 1, r)
end if.
```

function partition(p, r: **indeksi**): **indeksi**

1: Valitse jollakin strategialla jokin alkioista $A[p \dots r]$ jakoalkioksi x;
2: Vaihtele alkioita $A[p \dots r]$ siten, että jaolta vaaditut ehdot tulevat voimaan;
3: **return** se indeksi q johon rivillä 1 valittu jakoalkio x päätyi rivin 2 vaihtojen seurauksena.

- Rivillä 1 kannattaisi valita sellainen x jonka q on mahdollisimman *keskellä* taulukkoa $A[p \dots r]$.

- Mutta se pitäisi tehdä oleellisesti yhdellä pyyhkäisyllä, eli

$$O(r - p) \qquad \text{askeleessa.} \tag{1}$$

3.2.1 Aineiston partitiointi

[1, kuvat 7.1–7.3]

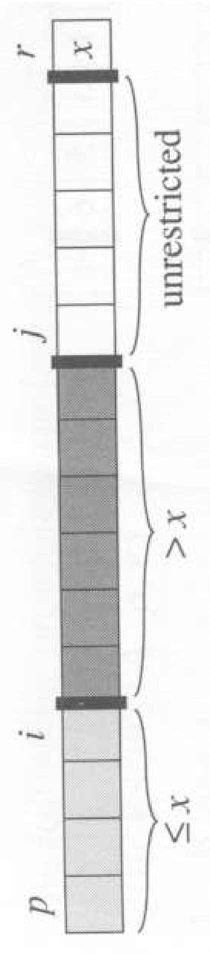
- Kehitetään eräs sellainen funktio partition(A, p, r) joka toteuttaa yhtälön (1).

- Jaetaan syöte $A[p \dots r]$ 4 osaan:

  1. Osan $A[p \dots i]$ alkiot ovat $\leq x$.
  2. Osan $A[i + 1 \dots j - 1]$ alkiot ovat $> x$.
  3. Osan $A[j \dots r - 1]$ alkioita ei ole vielä käsitelty.
  4. Osa $A[r] = x$ on jakoalkio. Se ei muutu.

- Tämä 4-jako olkoon silmukkainvariantti.

- Joka kierroksella käsitellään osan 3 seuraava alkio $A[j]$ osaan 1 tai 2.



function partition(p, r: **indeksi**): **indeksi**

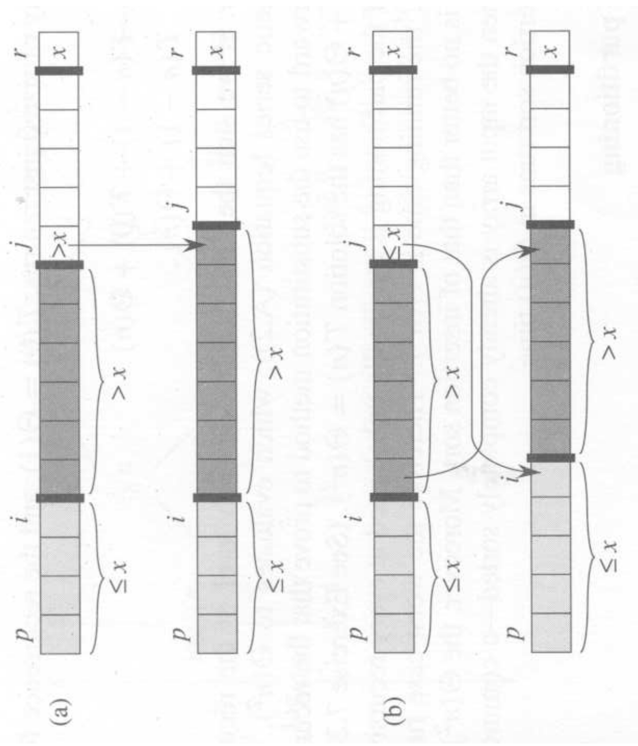
```
x := A[r];
i := p − 1;
for all j := p to r − 1 do
    if A[j] ≤ x then
        i := i + 1;
        vaihda(i, j)
    end if
end for;
vaihda(i + 1, r);
return i + 1.
```

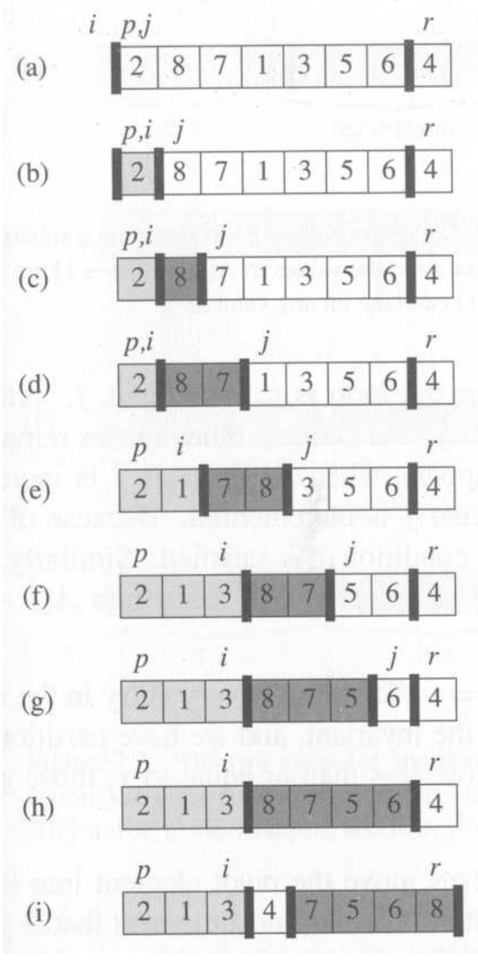
- Voitaisiin parantaa jakoalkion x valintastrategiaa:

  Viimeisen valinta on huono strategia:
  Jos A on jo valmiiksi lajiteltu?

  Jakoalkio laidassa on hyvä ohjelmoinnissa:
  Se ei ole muiden alkioiden siirtelyssä tiellä.

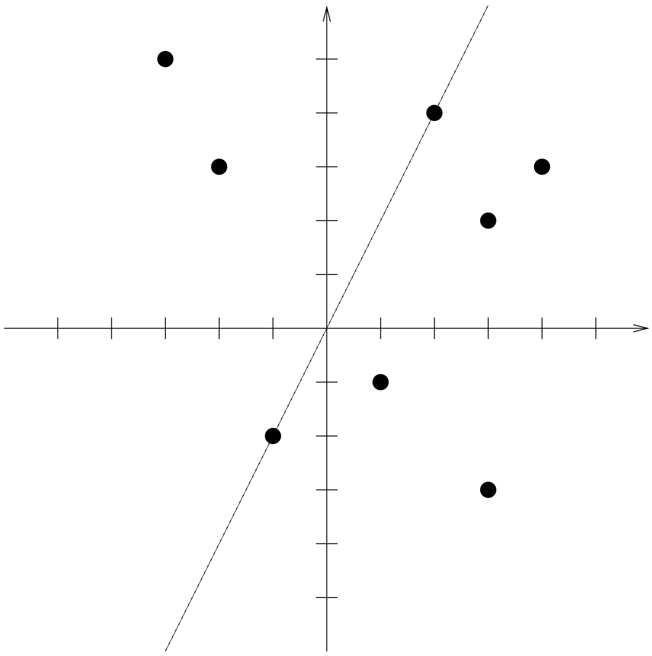
  Hyvä strategia voi aloittaa vaihtamalla valitsemansa paremman jakoalkion taulukon laitaan.





3.3 Järjestäminen algoritmin osana

- Tehokas järjestäminen on usein käyttökelpoinen *aliohjelmana*.
- Esimerkiksi *syötteen* järjestäminen etukäteen helpottaa usein lopputyötä.
- Esimerkki kalvojen 12 laskennallisesta geometriasta:
  - Syötteenä pisteet $p_1, p_2, p_3, \ldots, p_n$ koordinaatistossa kokonaislukupareina $p_i = \langle p_i.x, p_i.y \rangle \in \mathbb{Z} \times \mathbb{Z}$.
  - Ovatko mitkään kaksi pistettä samalla origon $\langle 0, 0 \rangle$ kautta kulkevalla suoralla?
  - Oletetaan että jokainen $x_i \neq 0$.
- Muita esimerkkejä: kalvojen 10.2.6 tehtävä, Grahamin pyyhkäisy kalvoilla 12.2, janojen leikkaaminen kalvoilta 12.4.2,. . .



- Origon ja pisteen p kautta kulkevan suoran yhtälö on

$$y = \text{kulmakerroin}(p) \cdot x$$

missä

$$\text{kulmakerroin}(p) = \frac{p.y}{p.x}$$

joten pisteet p_i ja p_j ovat samalla origon kautta kulkevalla suoralla täsmälleen kun

$$\text{kulmakerroin}(p_i) = \text{kulmakerroin}(p_j).$$

- Ongelma saa siis muodon:

"Toistuuko luvuissa

$$\begin{aligned} K = {} & \text{kulmakerroin}(p_1), \\ & \text{kulmakerroin}(p_2), \\ & \text{kulmakerroin}(p_3), \\ & \vdots \\ & \text{kulmakerroin}(p_n) \end{aligned}$$

mikään arvo?"

Muunnos vie

$$O(n)$$

askelta.

Perusmenetelmä tekee jokaiselle luvulle vertailut

$$\begin{aligned} \text{kulmakerroin}(p_i) \text{ vs } & \text{kulmakerroin}(p_{i+1}), \\ & \text{kulmakerroin}(p_{i+2}), \\ & \text{kulmakerroin}(p_{i+3}), \\ & \vdots \\ & \text{kulmakerroin}(p_n) \end{aligned}$$

sen jälkeisiin lukuihin.

Vie

$$O(n^2)$$

askelta.

Nopeampi menetelmä on:

1. Ensin lajittele (nopeasti) luvut K.
2. Sitten katso toistuuko lajitelluissa luvuissa sama arvo 2 kertaa peräkkäin.

Vie vain

$$O(n \cdot \log(n) + n) = O(n \cdot \log(n))$$

askelta.

- Ratkaisussa on vielä ongelma:

$$\text{kulmakerroin}(p_i) \in \mathbb{Q}$$

joten eikö $\mathbb{Z}$-aritmetiikka riitäkään?

- Kirjoitetaan auki algoritmin testi

$$\begin{aligned} \text{kulmakerroin}(p) &\leq \text{kulmakerroin}(q) \\ \frac{p.y}{p.x} &\leq \frac{q.y}{q.x} \end{aligned}$$

josta ristiin kertomalla saadaan

$$\begin{cases} p.y \cdot q.x \leq p.x \cdot q.y & \text{kun } p.x \cdot q.x > 0 \\ p.y \cdot q.x \geq p.x \cdot q.y & \text{kun } p.x \cdot q.x < 0 \end{cases}$$

joka on $\mathbb{Z}$-aritmetiikkaa.

- Järjestää saa *minkä tahansa järjestysrelaation* suhteen.

Järjestetään siis tämän relaation suhteen.

- Tällaisen vuoksi valmiit kirjastorutiinit järjestämiseen usein ottavatkin käytettävän vertailurutiinin parametrina.

4 Aritmetiikasta

[8, luku 5]

- Kisatehtävässä saattaa joutua *laskemaan hyvin suurilla kokonaisluvuilla*.
- Nämä luvut voivat olla satojen (jopa tuhansien!) numeromerkkien pituisia.
- Silloin ei voikaan käyttää valmiita kokonaislukutyyppejä, joissa *talletettujen bittien lukumäärä* on
  - kiinteä
  - pieni.

- Kisakäytössä yleisin yhdistelmä on

kääntäjänä ohjelmointikielille C ja C++ *the GNU Compiler Collection* eli `gcc` (`http://gcc.gnu.org/`)

keskusyksikkönä Intel Pentium -perheen prosessori.

- Tässä yhdistelmässä on tarjolla

tyyppi	bitit	suurin arvo
`short int`	16	+ 32 767
`int`	32	+ 2 147 483 647
`long int`	32	+ 2 147 483 647
`long long int`	64	+ 9 223 372 036 854 775 807

- Pienin arvo on

$$-(1 + \text{suurin arvo}).$$

- Tyyppi `long long int` ei kuitenkaan ole ANSI-standardin mukainen, joten sen käyttö voi aiheuttaa käännösaikaisen varoituksen.

- On myös valmiita mielivaltaisen laskentatarkkuuden kirjastoja.

- Erityisen yleinen on *GNU Multiprecision Library* eli `gmp` (`http://www.gnu.org/software/gmp/manual/`).

- Nämä kirjastot eivät kuitenkaan kuulu kisoissa käytettävien ohjelmointikielten standardikirjastoihin.

- Niinpä suurten kokonaislukujen käsittelyn joutuu useimmiten ohjelmoimaan itse.

4.1 Liukuluvuista

- Liukuluvut *eivät auta:*
  - Vaikka suurin liukuluku voikin olla esimerkiksi

$$1.79769313486 \times 10^{308}$$

  - niin sen kokoiset numerot onkin talletettu pelkkänä *likiarvona:*

    vain nämä 12 ensimmäistä numeroa ovat tallessa, muut 296 ovat nollautuneet!

- IOI- ja BOI-kilpailutehtävissä ei (vielä) käytetä liukulukulaskentaa
  - juuri tällaisten *pyöristysvirheiden* vuoksi
  - joten syötteen liukuluvut voi skaalata pois.

- Nyt yleisin liukulukustandardi on *IEEE 754* (`http://grouper.ieee.org/groups/754/`).

- Siinä liukuluku esitetään muodossa

$$\overbrace{\pm 1.\underbrace{b_2 b_3 b_4 \ldots b_p}_{\text{mantissabitit}} \times 2^e}^{\text{talletettu } t \text{ tavuun}}$$

- Ohjelmointikielissä C ja C++ tämä standardi tarkoittaa seuraavaa:

tyyppi	p	e	t
`float`	24	$-126 \ldots +127$	4
`double`	53	$-1022 \ldots +1023$	8

- Intel IA64 -prosessoreissa *sisäinen* mantissan tarkkuus onkin $p = 64$ bittiä.
  - Sillä pyritään välttämään pyöristysvirheet lausekkeiden välivaiheissa.
  - Vasta kun lopputulos talletetaan muuttujaan, niin se pyöristetään.
  - Kääntäjässä `gcc` tämä sisäinen esitys on tyyppinä `long double`, ja sen koko on $t = 12$ tavua.

4.2 Pitkät luvut muistissa

Taulukkona numeromerkkejä:

- Jos tehtävästä näkee ylärajan käsiteltävien lukujen pituudelle, niin siinä on taulukoiden maksimipituus.
- Muuten pitää varata tuloksille oikean mittainen taulukko dynaamisesti:

lasku	tuloksen maksimipituus
$a+b$	$1+\max(a\text{:n pituus}, b\text{:n pituus})$
$a \cdot b$	$1 + a$:n pituus $+ b$:n pituus
$\vdots$	$\vdots$

- Voi myös ylläpitää tietoa luvun *todellisesta* pituudesta — ja jättää sen alkunollat käsittelemättä.

Numeromerkkinä kussakin taulukkopaikassa:

- $0, 1, 2, \ldots, 9$ desimaali- eli 10-kantaluvuille
- 0 tai 1 binääri- eli 2-kantaluvuille.

Valitaan siten, että vältetään vaivalloiset kantalukumuunnokset (jos mahdollista).

Indeksointi valitaan *kasvavaksi numeroposition mukaan*, eli

indeksi	desimaali	binääri
0	ykköset	ykköset
1	kymmenet	kakkoset
2	sadat	neloset
$\vdots$	$\vdots$	$\vdots$
p	10^p:t	2^p:t
$\vdots$	$\vdots$	$\vdots$

eli indeksissä p on position p monikertojen kantaluku^p lukumäärä koko luvussa.

Etumerkki $+$ tai $-$:

- Talletetaan omana erillisenä kenttänään.
- Asetetaan laskutoimituksissa omana (esi- tai jälki)vaiheenaan.
- Muista käsitellä erikoistapaus -0!

4.3 Yhteenlasku

<u>Saman</u>merkkisten lukujen a ja b yhteenlasku:

> Alusta ylivuotobitti d **:=** 0;
> **for** p **:=** 0 **to** max(a:n pituus, b:n pituus) **do**
> d,$c[p]$ **:=** jakolaskun $\frac{a[p]+b[p]+d}{\text{kantaluku}}$ osamäärä ja jakojäännös
> **end for**

<u>Eri</u>merkkisten lukujen a ja b yhteenlasku onkin $+$-merkkisten lukujen vähennyslaskua kavoilta 4.4.

4.4 Vähennyslasku

- Jos molemmat luvut a ja b ovat $+$-merkkisiä, niin käytetään vastaavaa ideaa kuin kalvojen 4.3 yhteenlaskussa:

> Alusta lainausbitti d **:=** 0;
> **for** p **:=** 0 **to** a:n pituus **do**
> v **:=** lyhyt luku $(a[p] - b[p] - d)$**;**
>   **if** $v < 0$ **then**
> v **:=** $v +$ kantaluku**;**
> d **:=** 1
>   **else if** $a[p] > 0$ **then**
> d **:=** 0
>   **end if;**
> $c[p]$ **:=** jakolaskun $\frac{v}{\text{kantaluku}}$ jakojäännös
> **end for.**

- Tämä idea kuitenkin vaatii, että $a \geq b$.

  Se voidaan taata kalvojen 4.5 algoritmilla ja etumerkkien käsittelyllä.

- Muuten $-$-merkki voidaan vaihtaa $+$-merkiksi ja kalvojen 4.3 yhteenlaskuksi.

4.5 Suuruusvertailu

Erimerkkisten lukujen a ja b suuruusvertailu on niiden etumerkkien vertailua.

Samanmerkkisten lukujen a ja b suuruusvertailu on *suurimman* indeksiposition p etsimistä, jossa lyhyet luvut $a[p]$ ja $b[p]$ eroavat toisistaan.

4.6 Kertolasku

- Kertolasku kouluopein "rivi riviltä":

$$\begin{aligned} a \cdot b &= a \cdot \left(\sum_{p=0}^{b\text{:n pituus}-1} b[p] \cdot \text{kantaluku}^p \right) \\ &= \sum_{p=0}^{b\text{:n pituus}-1} (b[p] \cdot a \cdot \text{kantaluku}^p) \\ &= \sum_{p=0}^{b\text{:n pituus}-1} \sum_{q=1}^{b[p]} a\underbrace{000\ldots0}_{p\text{ kpl}} \end{aligned}$$

- Sama algoritmina:

Alusta vastaus c **:=** 0**;**
Alusta siirtyvä rivi r **:=** a**;**
for p **:=** 0 **to** b:n pituus $-$ 1 **do**
  for q **:=** 1 **to** $b[p]$ **do**
    c **:=**$c + r$ kalvoilta 4.3
  end for;
  r **:=** r perässään vielä uusi numero **0**
end for.

- Etumerkkien käsittely yksinkertaista.

4.7 Jakolasku

- Jakolaskun $\frac{a}{b}$ idea on
  - koulussa opittu jakokulmamenetelmä
  - siirtää riviä kuten kalvojen 4.6 kertolaskussa, mutta *toiseen suuntaan*.
- Intuitio ruutupaperilla:
  1. Kirjoita luvut a ja b alakkain siten, että niiden *vasemmat* reunat ovat tasan.
  2. Laske montako kertaa b voidaan vähentää siitä luvun a alkuosasta a', jonka alla b tällä hetkellä on.
  3. Kirjoita askeleen 2 antama tulos vastauksen c siihen kohtaan, jossa luvun b oikea reuna nyt on.
  4. Korvaa luvun a alkuosa a' sillä luvulla, joka jäi yli askeleen 2 vähennyslaskuista.
  5. Siirrä lukua b yksi sarake oikealle, ja jatka askeleesta 2.

Alusta siirtyvä rivi r **:=** 0**;**
for p **:=** a:n pituus $-$ 1 **down to** 0 **do**
  r **:=** r perässään vielä uusi numero $a[p]$**;**
  $c[p]$ **:=** 0**;**
  while $r \geq b$ kalvoilta 4.5 **do**
    $c[p]$ **:=** $c[p] + 1$**;**
    r **:=** $r - b$ kalvoilta 4.4
  end while
end for.

Silmukan pysähtyessä

osamäärä on tulosmuuttujassa c

jakojäännös on siirtyvällä rivillä r.

4.8 Potenssiin korotus

- Potenssiin korotus a^n, missä
  - kantaluku a on pitkä
  - eksponentti n on lyhyt

  on kalvojen 4.6 kertolaskun toistoa.

- Perustapa

$$\underbrace{a \cdot a \cdot a \cdot \ldots \cdot a}_{n \text{ kpl}}$$

  on hidas — $n-1$ kertolaskua.

- Tehokkaampaa on hajoittaa ja hallita eksponenttia $n > 0$ kalvojen 10.1 tapaan:

$$\begin{aligned} a^1 &= a \\ a^{2\cdot m} &= \left(a^2\right)^m \qquad \text{tai vaihtoehtoisesti} \\ &= (a^m)^2 \\ a^{2\cdot m+1} &= a^{2\cdot m} \cdot a \end{aligned}$$

  Kertolaskuja tarvitaan enää $O(\log_2 n)$.

- Nollat ja etumerkit voi käsitellä erikseen.

function pow(a: $\mathbb{N}$, n: $\mathbb{N}$): $\mathbb{N}$

```
if n = 1 then
  c := a
else if n on parillinen then
  c := pow(a · a, n/2) kalvoilta 4.6
else
  c := a · pow(a, n − 1) kalvoilta 4.6
end if;
return c.
```

- Sama algoritmi sopii myös pitkille eksponenteille n. (Tosin vastaus c on *hyvin* pitkä luku...)

- Rekursiivisessa haarassa $n > 1$ lasketaan

$$\frac{n}{2}.$$

  Helppoa jos n on 2-kannassa; muuten kuten kalvoilla 4.7.

  Jakojäännös ilmaisee, oliko n parillinen vai pariton.

  Osamäärä on se luku, jolla tehdään seuraava rekursiokutsu.

4.9 Kantalukumuunnos

- Jos

  laskenta on helppoa yhdessä, mutta

  syöttö tai tulostus vaaditaankin toisessa

  kannassa, niin tarvitaan kantalukumuunnos.

- Aloitetaan helpommasta erikoistapauksesta:
  - Luku $x > 0$ on *koneen omassa* esitysmuodossa

    jolloin voi käyttää koneen omaa lyhyiden lukujen aritmetiikkaa.
  - Se halutaan pitkään taulukkomuotoon d kannassa b

    oletetaan kantalukujen olevan koneen omassa muodossa.

- Voidaan toistaa koneen omaa jakolaskua:

```
Alusta tuotettava positio p := 0;
while x > 0 do
  x,d[p] := koneen oman jakolaskun x/b
  osamäärä ja jakojäännös;
  p := p + 1
end while.
```

- Nollan ja etumerkin voi käsitellä erikseen.

- Yleisessä tapauksessa myös x on pitkässä taulukkomuodossa kantanaan a.

  Käytetään samaa ideaa:

```
Alusta jakaja q := kohdekannan b
taulukkoesitys lähdekannassa a
tuotettuna yllä olevalla erikoistapauksella;
Alusta tuotettava positio p := 0;
while x > 0 do
  x,d[p] := pitkien lukujen jakolaskun x/q
  osamäärä ja jakojäännös kalvoilta 4.7;
  p := p + 1
end while.
```

5 Lukuteoriasta

[8, luku 7; 1, luku 31]

- Matematiikan haara joka tutkii *kokonaislukujen jaollisuusominaisuuksia*.
- Tietojenkäsittelyssä sillä on sovelluksia esimerkiksi *salakirjoituksessa*.
- Ohjelmointikilpailuissa siihen törmää esimerkiksi sellaisissa tehtävissä, joissa käsitellään
  - alkulukuja (kalvot 5.2)
  - laskutoimituksia luvuilla jotka "pyörähtävät ympäri" takaisin pieniksi kasvaessaan yli tietyn rajan (kalvot 5.4).
- Tällaisissa tehtävissä tarvitaan usein myös kalvojen 4 suurten lukujen käsittelykirjastoja.

5.1 Esimerkki: parillinen vai pariton?

- Tehtävässä [8, Problem 7.6.1]

  annetaan kokonaisluku $n > 0$

  tarkastellaan niitä lukuja $i = 1, 2, 3, \ldots, n$ jotka jakavat luvun n tasan

  kysytään onko sellaisten i lukumäärä parillinen vai pariton.

  Se selviää yksinkertaisella jaollisuuspäättelyllä.
- Luku i jakaa tasan luvun n täsmälleen silloin kun on olemassa luku j jolla
$$i \cdot j = n.$$
- Mutta silloinhan myös tämä luku j on itsekin sellainen luku, joka jakaa luvun n tasan!
- Eli kaksi sellaista lukua ovat aina keskenään pari!

- Onko vastaus siis aina "parillinen"?
- Kyllä, *paitsi*
  - silloin kun $i = j$
  - tämä pari kutistuukin yhdeksi ainoaksi luvuksi.
- Siis vastaus on

  pariton jos n on jonkin toisen luvun *neliö*, eli
$$n = 1, 4, 9, 16, 25, \underbrace{36}_{6\cdot 6}, \ldots$$

  parillinen muuten.

(Tämän ratkaisun löysi Topi Musto syksyn 2004 olympiavalmennusleirillä.)

5.2 Alkuluvuista

- *Alkuluvut* (primes) ovat ne luvut p, jotka eivät jakaudu tasan millään muilla luonnollisilla luvuilla kuin 1 ja p itse:
$$2, 3, 5, 7, 11, 13, 17, \ldots$$
- Annetun luvun $x > 0$ *alkulukukehitelmä* (prime factorization) on tulo
  - joka koostuu alkuluvuista
  - jonka arvo on x.

  Esimerkiksi $315 = 3 \cdot 3 \cdot 5 \cdot 7$.
- Jos alkuluku p esiintyy luvun x kehitelmässä, niin p on luvun x *tekijä* (factor).
- Kehitelmän voi tulostaa yksinkertaisesti
  - yrittämällä jakaa lukua x luvuilla 2 ja parittomilla luvuilla $3, 5, 7, \underline{9}, 11, 13, \underline{15}, \ldots$
  - lopettamalla kun yritys on edennyt ohi kohteen neliöjuuren.

while x on parillinen **do**
Tulosta tekijä 2;
x **:=** $\frac{x}{2}$
end while;
q **:=**3;
while $q \le \sqrt{x}$ **do**
s,r **:=** jakolaskun $\frac{x}{q}$ osamäärä ja jakojäännös;
if $r = 0$ **then**
Tulosta tekijä q;
x **:=** s
else
$q := q + 2$
end if
end while.

- Jos luvut x ja p mahtuvat koneen laskentatarkkuuteen, niin voit käyttää ohjelmointikielen $\sqrt{x}$-kirjastofunktiota.

  Estä silloin pyöristysvirheen vaikutus lisäämällä ylimääräinen kierros: $q \le \sqrt{x} + 1$.

- Jos käytät kalvojen 4 pitkiä lukuja, niin käytä kertolaskua: $q \cdot q \le x$.

  Tämän kertolaskun voi myös korvata yhteenlaskuilla: $(t + 1)^2 = t^2 + 2 \cdot t + 1$.

5.3 Eukleideen algoritmi

- Kahden kokonaisluvun a ja b *suurin yhteinen tekijä (s.y.t.)* (greatest common divisor, gcd) on suurin kokonaisluku z jolla jakolaskut
$$\frac{a}{z} \quad \text{ja} \quad \frac{b}{z}$$
päättyvät tasan.

  Luvun z alkulukukehitelmässä esiintyy tekijä p täsmälleen yhtä monta kertaa kuin p esiintyy siinä lukujen a ja b alkulukukehitelmistä, joissa p esiintyy vähemmän.

- Jos tämä z on triviaalivastaus 1, niin a ja b ovat *suhteellisia* alkulukuja (relatively prime).

  Silloin niiden alkulukukehitelmissä ei ole yhtään yhteistä tekijää.

- Murtoluvun
$$\frac{a}{b}$$
sievin muoto (jossa osoittajalla ja nimittäjällä ei ole yhteisiä tekijöitä) on
$$\frac{a/z}{b/z}$$
koska z on näiden yhteisten tekijöiden tulo.

- Lukujen a ja b *pienin yhteinen monikerta (p.y.m.)* (least common multiple, lcm) on pienin luku u jolla jakolaskut
$$\frac{u}{a} \quad \text{ja} \quad \frac{u}{b}$$
menevät tasan. Se voidaan laskea
$$u = \frac{a \cdot b}{z}$$
koska jakolasku poistaa tulosta yhteisten tekijöiden kaksoiskappaleet.

  – Presidentinvaalit ovat 6 vuoden välein,

  – kunnallisvaalit 4 vuoden välein.

  – Kuinka usein ne ovat yhtä aikaa?

  – Vastaus: p.y.m. eli 12 vuoden välein.

- Jos halutaan selvittää annettujen lukujen a ja b s.y.t. (missä $a \ge b$) niin tehdään jakolasku $\frac{a}{b}$.

  – Jos jako meni tasan, niin vastaus on b.

  – Jos jäi jakojäännös $0 < r < b$, niin selvitetään lukujen b ja r s.y.t. samalla menetelmällä.

- Näin saadaan *Eukleideen perusalgoritmi:*

  function syt(a, b)
  if $b = 0$ **then**
  return a
  else
  r **:=** jakolaskun $\frac{a}{b}$ jakojäännös;
  return syt(b, r)
  end if.

- Se on nopea:
$$O(\log_2 b)$$
rekursiokutsua.

  Sopii myös kalvojen 4 pitkille luvuille!

- Eukleideen algoritmi voi tuottaa lisäksi sellaiset kokonaisluvut x ja y, joilla pätee

  $$a \cdot x + b \cdot y = \text{syt}(a, b).$$

  Lukua x tarvitaan kalvojen 5.4.3 jakolaskussa halutun jakojäännöksen suhteen.

  (Lukua y tarvitaan luvun x laskemiseen.)

- Eukleideen algoritmin johdon perusteella

  $$\text{syt}(a, b) = \text{syt}(b, r)$$

  missä r on jakolaskun $\frac{a}{b}$ jakojäännös. Se voi tuottaa induktiivisesti (eli rekursiivisesti) myös luvut x' ja y' joilla

  $$= b \cdot x' + r \cdot y'.$$

  Toisaalta

  $$r = a - b \cdot s$$

  missä

  $$s = \text{kyseisen jakolaskun osamäärä.}$$

  Näiden tietojen järjestely uudelleen antaa

  $$\begin{aligned} x &= y' \\ y &= x' - s \cdot y'. \end{aligned}$$

- Ei-rekursiivisena perustapauksena on

  $$\begin{aligned} \text{syt}(a, 0) &= a \\ &= a \cdot 1 + 0 \cdot 0 \end{aligned}$$

  eli

  $$\begin{aligned} x &= 1 \\ y &= 0. \end{aligned}$$

- *Eukleideen laajennettu algoritmi* on siten

  function syt$'(a, b)$
    if $b = 0$ **then**
      return $\langle a, 1, 0 \rangle$
    else
      s,r **:=** jakolaskun $\frac{a}{b}$ osamäärä ja jakojäännös**;**
      $\langle g, x', y' \rangle$ **:=** syt$'(b, r)$**;**
      return $\langle g, y', x' - s \cdot y' \rangle$
    end if.

5.4 Modulaariaritmetiikasta

Kellotauluintuitio:

- Numerot

  $$\underbrace{0, 1, 2, 3, \ldots, n-1}_{n \text{ kappaletta}}$$

  kirjoitetaan ympyrän kehälle peräkkäin myötäpäivään.

- Lukua $n - 1$ seuraa taas 0.

- Kahden tällaisen luvun *yhteenlasku* $p + q$ on

  "Aloita luvusta p. Kulje myötäpäivään q askelta."

- Vähennyslasku kulkee *vasta*päivään.

- Silloin kaikki luvut muotoa

  $$p + i \cdot n \qquad \text{missä } i \in \mathbb{Z}$$

  ovat oleellisesti yksi ja sama luku!

Jakojäännösintuitio: Jokaisen laskutoimituksen jälkeen tulos jaetaan vielä luvulla n ja otetaan pelkkä jakojäännös.

Koneen oma kokonaislukuaritmetiikka on tällaista:

- Kierroksen koko on

  $$\begin{aligned} n &= 2^\ell \\ \ell &= \text{muistipaikan bittien lukumäärä.} \end{aligned}$$

- Negatiivisina tulostetaan ne luvut, joiden ylin eli $(\ell - 1)$. bitti on `1`.

- Siis lukualue näyttää olevan

  $$-2^{\ell-1} \ldots + 2^{\ell-1} - 1.$$

- Tavallisimmat lukualueet ovat kalvoilla 4.

Merkintä x mod n

- luetaan "x *modulo* n"
- tarkoittaa luvun x "luontevaa" edustajaa tässä *aritmetiikassa modulo* n
- eli sitä lukua $0 \leq y < n$, jolla
$$x = y + i \cdot n \qquad \text{missä } i \in \mathbb{Z}$$
- eli sitä lukua y, johon päädytään kellotaulussa, kun
  - lähdetään liikkeelle luvusta 0
  - otetaan x askelta.

Merkintä $x \equiv y \pmod{n}$ tarkoittaa, että luvuilla x ja y on sama edustaja modulo n.

Yhteen- ja vähennyslasku on siten suoraan

$$(x \pm y) \bmod n = ((x \bmod n) \pm (y \bmod n)) \bmod n.$$

5.4.1 Esimerkki: Viikonpäivät

- Mille viikonpäivälle tämä sama päivämäärä osuu ensi vuonna?
- Etäisyys siihen on

  366 päivää jos välissä on karkauspäivä 29.2.

  365 päivää jos ei.
- Voisimme toki laskea näin monta askelta eteenpäin kalenterista. . .
- Tai huomata: viikonpäivät muodostavat "kellotaulun" jossa on $n = 7$ "numeroa".
- Riittää siis laskea
$$366 \bmod 7 = 2 \qquad \text{ja} \qquad 365 \bmod 7 = 1.$$
- Vastaus on siis

  ylihuominen jos väliin jää karkauspäivä

  huominen jos ei.

5.4.2 Modulaarinen kertolasku

Kertolasku on toistettuna yhteenlaskuna

$$(x \cdot y) \bmod n = ((x \bmod n) \cdot (y \bmod n)) \bmod n.$$

- Erityisesti
$$n = \text{kantaluku}^{\ell}$$
  tarkoittaa tulon ℓ *viimeistä numeroa*.

Potenssiin korotus on toistettuna kertolaskuna

$$(x^y) \bmod n = ((x \bmod n)^y) \bmod n.$$

- Kalvojen 4.8 algoritmissa voidaan suorittaa kaikki laskutoimitukset "modulo n".

  Vihje tehtävään [8, Problem 7.6.3]!
- Jos n on lyhyt luku, ne voidaan suorittaa koneen omalla aritmetiikalla.
- Silloin on esimerkiksi nopeaa selvittää luvun 2^{100} viimeinen numero 10-järjestelmässä tuottamatta itse lukua.

5.4.3 Modulaarinen jakolasku

- Luvun a *käänteisalkio kertolaskun suhteen* a^{-1} (multiplicative inverse) määritellään yhtälön
$$a \cdot x = 1$$
  ratkaisuna.
- Silloin jakolasku voidaan määritellä
$$\frac{b}{a} = b \cdot a^{-1}.$$
- Samat määritelmät soveltuvat myös modulo n, mutta nyt yhtälöllä
$$a \cdot x \equiv 1 \pmod{n}$$
  ei välttämättä olekaan ratkaisua!

  Silloin tällä a jakaminen modulo n onkin *mahdotonta!*

- Tällä yhtälöllä

$$a \cdot x \equiv 1 \pmod{n}$$

on ratkaisu täsmälleen silloin, kun luvut a ja n ovat suhteellisia alkulukuja.

Eli täsmälleen silloin kun kalvojen 5.3 laajennettu Eukleideen algoritmi vastaa

$$\text{syt}'(a, n) = \left\langle 1, x', y' \right\rangle.$$

- Silloin tämän vastauksen ja modulaariaritmetiikan ominaisuuksista seuraa

$$a \cdot x \equiv 1 \equiv a \cdot x' + n \cdot y' \equiv a \cdot x' \pmod{n}$$

joten etsitty käänteisalkio on

$$a^{-1} = x'.$$

- Yleinen jakolaskun yhtälö

$$a \cdot x \equiv b \pmod{n}$$

ratkeaa seuraavasti:

1. Lasketaan $\langle d, x', y' \rangle = \text{syt}(a, n)$ kalvojen 5.3 laajennetulla Eukleideen algoritmilla.

2. Jos (tavallinen) jakolasku

$$b' = \frac{b}{d}$$

ei mene tasan, niin ratkaisuja ei ole lainkaan.

3. Muuten yhtälöllä on d ratkaisua

$$x = b' \cdot x' + i \cdot n' \pmod{n}$$

missä

$$n' = \frac{n}{d}$$
$$i = 0, 1, 2, 3, \ldots, d - 1$$

eli jakolaskun varsinainen tulos (mod d)+ kierros kellotaulun ympäri d numeron mittaisin askelein.

- Tapaus $a = 0$ voidaan käsitellä erikseen:
  - Jos myös $b = 0$, niin jokainen $x = 0, 1, 2, 3, \ldots, n - 1$ käy tulokseksi.
  - Jos $b \neq 0$, niin mikään niistä ei käy.

- Jos n on kalvojen 5.2 alkuluku, niin
  - $d = 1$ kaikilla $a \neq 0$
  - jakolaskun ainoa tulos on varsinainen

$$x = b \cdot x'.$$

- *Lineaarinen kahden muuttujan Diofantoksen yhtälö* on

$$a \cdot x - n \cdot y = b$$

jonka

- vakiot a, n ja b
- muuttujat x ja y

ovat kokonaislukuja. Se on sama yleinen jakolaskun yhtälö kun y saa olla mikä tahansa.

5.4.4 Kiinalainen jäännöslause

- Kiinalainen jäännöslause (Chinese Remainder Theorem, CRT) ratkaisee yhtälöparin

$$x \equiv a_1 \pmod{m_1}$$
$$x \equiv a_2 \pmod{m_2}$$

jonka modulukset m_1 ja m_2 ovat keskenään suhteellisia alkulukuja.

- Tämä ratkaisu $x \pmod{m_1 \cdot m_2}$
  - löytyy ratkaisemalla ristikkäiset yhtälöt

$$m_2 \cdot x_1 \equiv 1 \pmod{m_1}$$
$$m_1 \cdot x_2 \equiv 1 \pmod{m_2}$$

kalvojen 5.4.3 tapaan, ja asettamalla

$$x = a_1 \cdot x_1 \cdot m_2 + a_2 \cdot x_2 \cdot m_1$$

HUOMAA: Kaavassa [8, luku 7.4.2] on painovirhe!

  - on yksikäsitteinen.

- Ohjelmoijan tulkinta:
  - Luku $c \pmod{m_1 \cdot m_2}$ voidaan esittää yksikäsitteisesti *lukuparina* $\langle c \pmod{m_1}, c \pmod{m_2}\rangle$.
  - Tässä lukupariesityksessä voidaan
    - yhteen-
    - vähennys-
    - kerto-

    laskutoimitus $\odot$ tehdä paikka paikalta:

    $$\langle p_1, q_1\rangle \odot \langle p_2, q_2\rangle = \langle p_1 \odot p_2 \pmod{m_1}, q_1 \odot q_2 \pmod{m_2}\rangle.$$
  - Nämä paikkakohtaiset laskutoimitukset $\odot$ voivat puolestaan onnistua koneen omalla aritmetiikallakin.
- Laskutoimitus $\odot$ voi olla myös jakolasku kalvojen 5.4.3 tapaan:

  tulokset saadaan paikkakohtaisten tulosten karteesisena tulona.

- Kiinalainen jäännöslause yleistyy myös useammalle paikalle:
  - Aritmetiikka modulo

    $$m = m_1 \cdot m_2 \cdot m_3 \cdot \ldots \cdot m_k$$
  - onnistuu näin k luvun monikoilla
  - kunhan jokainen modulopari

    $$m_i \quad \text{ja} \quad m_j$$

    on keskenään suhteelliset alkuluvut.
- Hyödyllinen siis silloin, jos
  - täytyy laskea modulo sellainen suuri m
  - jonka alkulukukehitelmässä (kalvoilta 5.2) on erikokoisia tekijöitä
  - koska silloin pienet modulot m_i voidaan valita monesta eri vaihtoehdosta.
  - Ainoa rajoitus on, että saman tekijän *kaikki* kopiot pitää laittaa samaan pieneen moduloon m_i.

- Esimerkiksi

  $$\begin{aligned} m &= 90 \\ &= 2 \cdot 3 \cdot 3 \cdot 5 \end{aligned}$$

  antaa vaihtoehdot

  $$\begin{gathered} 2, \underbrace{3 \cdot 3 \cdot 5}_{45} \\ \underbrace{2 \cdot 3 \cdot 3}_{18}, 5 \\ 2, \underbrace{3 \cdot 3}_{9}, 5 \\ \underbrace{3 \cdot 3}_{9}, \underbrace{2 \cdot 5}_{10} \end{gathered}$$

5.4.5 Esimerkki: Hammasrattaat

- Olkoon ongelmassa 2 hammasratasta:

  suuri ratas, jossa on m_1 hammasta
  - numeroituina $0, 1, 2, 3, \ldots, m_1 - 1$
  - *myötä*päivään

  pieni ratas, jossa on $m_2 < m_1$ koloa
  - numeroituina $0, 1, 2, 3, \ldots, m_2 - 1$
  - *vasta*päivään.
- Aluksi hammas 0 on kolossa 0.
- Kun suurta ratasta käännetään 1 askel vastapäivään, niin
  1. nykyinen hammas 0 irtoaa nykyisestä kolosta 0
  2. seuraava hammas 1 painuu seuraavaan koloon 1

  ja niin edelleen.

- Kun suurta ratasta on käännetty m_2 askelta vastapäivään

  pieni ratas on pyörähtänyt täyden kierroksen takaisin alkukoloonsa 0

  suuri ratas on seuraavassa hampaassaan m_2

  ja niin edelleen.

- Kuinka monta askelta x tarvitaan jotta saadaan hammas a_1 koloon a_2?

- Vastauksen antaa lukujen 5.4.4 kiinalainen jäännoslause

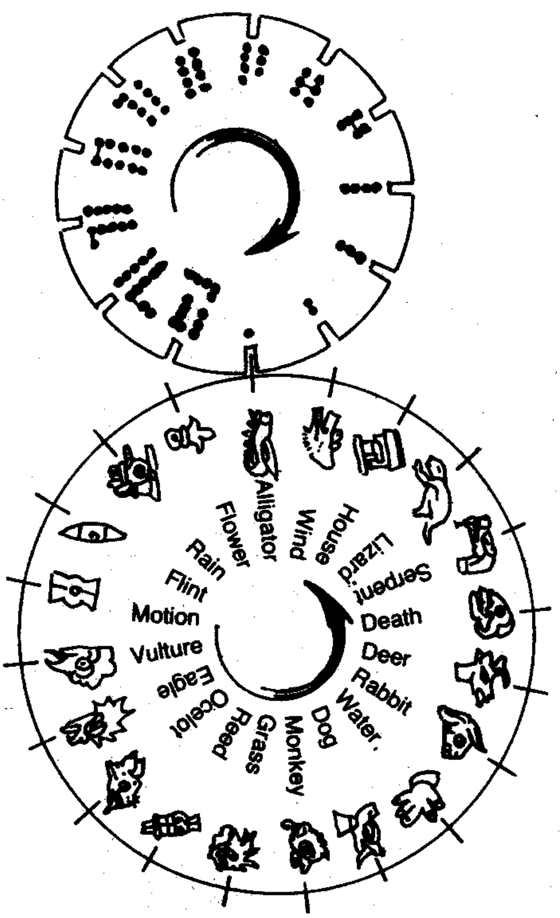
  jos m_1 ja m_2 ovat suhteellisia alkulukuja.

- Esimerkkinä seuraavan kuvan mukainen Atsteekkien pyhä kalenteri: päivät nimettiin "m n" missä

  – m on järjestysluku $1 \dots 13$

  – n on jokin 20 eri jumaluudesta

  ja päivät alkoivat "1 alligaattori", "2 tuuli", "3 talo",... [2, kuva 38].



6 Kombinatoriikasta

[8, luku 6]

- Matematiikan haara, joka laskee *kuinka monta erilaista rakennelmaa voi tehdä* kun annetaan

  – käytettävissä olevat rakennuspalikat

  – säännöt joilla niitä saa yhdistellä.

- Sen tuntemus säästää

  – itse rakennelmien kasaamisen

  – niiden laskemisen yksi kerrallaan

  vaivan.

- Tuttu esimerkki (binomikertoimet):

  – Huoneessa on n ihmistä.

  – Niistä pitää valita k ihmisen työryhmä.

  Montako erilaista työryhmää voi rakentaa?

- Sovitaan merkintä $f(n, k) =$ kysytty lukumäärä.

- Aloitetaan (tällä kertaa) rajatapauksista:

  – $f(n, n) = 1$ koska jokainen on pakko valita.

  – $f(n, 0) = 1$ koska erilaisia tyhjiä työryhmiä on vain 1 — tyhjä joukko.

  – Jos $k < 0$ tai $k > n$ tai $n < 0$, niin $f(n, k) = 0$ — sellaisia ryhmiä on mahdotonta muodostaa.

    Tästä nähdään, että mielenkiintoinen alue on $0 \leq k \leq n$.

- Voidaan muodostaa *rekurrenssi* eli yhtälö, joka määrittelee suuremman tapauksen vetoamalla pienempiin tapauksiin:
  - Otetaan huoneesta yksi ihminen x pois, ja muodostetaan huoneeseen jääneistä $n-1$ ihmisestä työryhmiä.
  - Jos x halutaan *mukaan* työryhmään, niin huoneeseen jääneistä pitää muodostaa $k-1$ hengen työryhmiä.

    Sellaisia on $f(n-1,k-1)$ erilaista — merkintäsopimuksemme nojalla.
  - Jos x halutaan *pois* työryhmästä, niin huoneeseen jääneistä pitää muodostaa k hengen työryhmiä.

    Sellaisia on $f(n-1,k)$ erilaista.
  - Saadaan rekurrenssi

    $$f(n,k) = f(n-1,k-1) + f(n-1,k)$$

    koska mikään ryhmä ei
    * jäänyt pois
    * tullut laskettua kahdesti.

- Tämä rekurrenssi *voitaisiin* ohjelmoida suoraan rekursiivisena funktiona:

  function $f(n,k)$
    if $n = 0$ **or** $n = k$ **then**
      return 1
    else
      return $f(n-1,k-1) + f(n-1,k)$
    end if.

- Se kuitenkin laskisi saman välituloksen monta kertaa:

  $$\begin{aligned} f(5,3) = & f(4,2) \\ & + f(4,3) \\ = & f(3,1) + \underline{f(3,2)} \\ & + \underline{f(3,2)} + f(3,3). \end{aligned}$$

- *Taulukoidaan* välitulokset matriisiksi

  $$F[i][j] = f(i+j,j)$$

  eli "montako tapaa on valita j ihmistä työryhmän sisä- mutta i ulkopuolelle".

  $F[i][0] = 1$
  $F[0][j] = 1$
  $F[i][j] = F[i-1][j] + F[i][j-1]$ kun $i,j > 0$.

- Matriisi F voidaan täyttää
  - rivijärjestyksessä $i = 0,1,2,3,\ldots,n+k$
  - rivin i sisällä sarakejärjestyksessä $j = 0,1,2,3,\ldots,k$

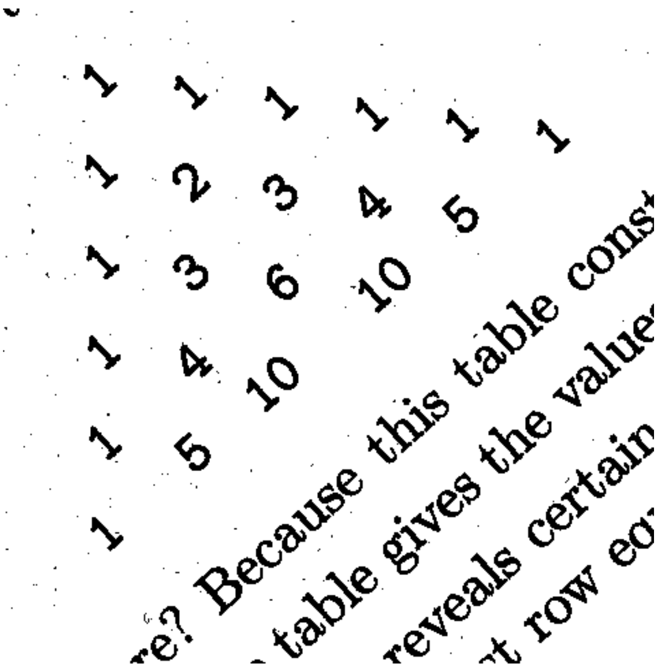
  koska paikan $F[i][j]$ $(i,j>0)$ arvo riippuu
  - edellisen rivin samasta sarakkeesta $F[i-1][j]$
  - saman rivin edellisestä sarakkeesta $F[i][j-1]$.

  function $f(n,k)$
    for j := 0 **to** k **do**
      $F[0][j]=1$
    end for;
    for i := 1 **to** $n+k$ **do**
      $F[i][0]=1$;
      for j := 1 **to** k **do**
        $F[i][j]= [i-1][j] + F[i][j-1]$
      end for
    end for;
    return $F[n+k][k]$**.**

- Näet taulukon
  - kääntämällä [8, sivu 132] 45° vastapäivään
  - leikkaamalla kolmion suorakulmioksi.
- Saimme laskettua **binomikertoimet**
  - joita merkitään tavallisesti $\binom{n}{k}$
  - ilman suljettua muotoa

    $$\binom{n}{k} = \frac{n!}{k! \cdot (k-n)!}$$

    ja sen aritmeettisia hankaluuksia
  - itse asiassa jopa laskemalla pienempi taulukko kuin kirjan yksinkertaisemmalla taulukointitavalla joka laskee kolmion. . .
- Samalla "johda rekurrenssi ja taulukoi sen arvot" -periaatteella saa ratkaistua monia kombinatorisia laskemisongelmia.
- Laskennan tehostaminen taulukoimalla sen välituloksia on keskeinen työväline myös kalvojen 10 dynaamisessa ohjelmoinnissa.



7 Peruuttavasta etsinnästä

Usein tehtävässä on:

alkutilanne josta pitää lähteä

operaatiot joilla tilanteesta saa seuraavat

loppuehto joka pitää saattaa voimaan jotta nykyinen tilanne olisi tehtävän ratkaisu.

Esimerkiksi Rubikin kuutiossa

alkutilanne on kuution nykyinen asento

operaatiot ovat kaikki "naksautukset"

loppuehto on että jokainen sivu on yksivärinen.

+ Ratkaisu(j)a voi *etsiä* jo näillä tiedoilla.

– Yleinen etsintämenetelmä on *hitaampi* kuin tehtävään erikoistunut algoritmi.

Etsintää voi ajatella (mahdollisesti äärettömän) *polkupuun kasvattamisena*:

Juurena on alkutilannesolmu s. Siitä aloitetaan.

Lapsina on isätilanteesta yhdellä operaatiolla saatavat tilanteet.

Yksi kasvatusaskel on laskea lapset jollekin vielä lapsettomalle solmulle.

Tavoitteena on saada kasvatettua puuhun solmu t, joka toteuttaa loppuehdon.

Usein loppusolmun tunnistus tehdään vasta kun se on valittu kasvatuskohdaksi.

Ratkaisuna tulostetaan löydetty operaatiojono

$$s \to \ldots \to t$$

joka voidaan lukea *takaperin*

$$s \leftarrow \ldots \leftarrow t$$

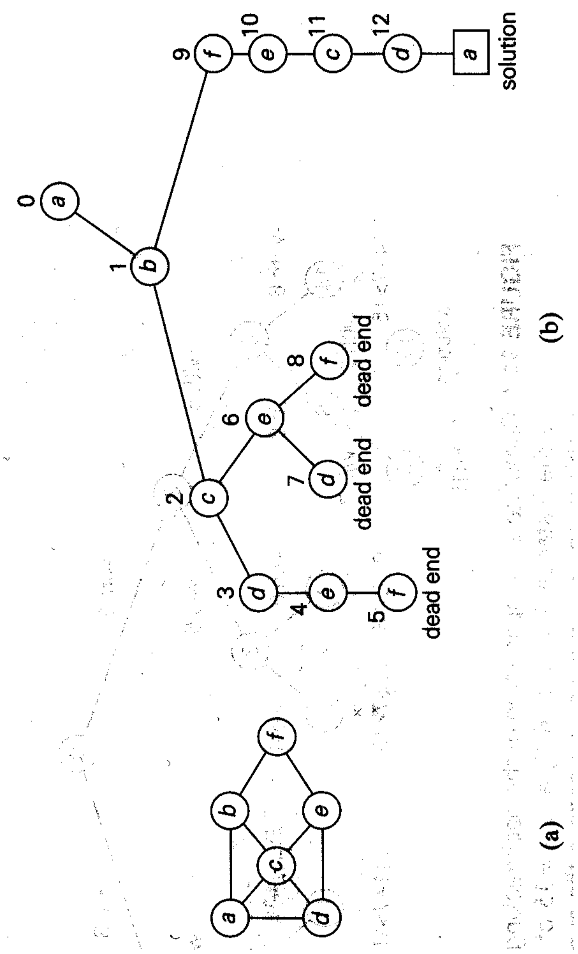
kun jokaisessa solmussa muistetaan sen isä, josta siihen tultiin.

7.1 Peruuttava etsintä

Peruuttava etsintä (Backtracking) [6, luku 11.1 ja kuva 11.3; 8, luku 8] valitsee seuraavaksi kasvatuskohdaksi viimeisimmän tehdyn lapsen:

procedure BackTrack(t: **tilanne**) **is**
  if tilanne t täyttää loppuehdon **then**
    Operaatiopolku polku(t) juuri- eli alkutilanteesta tilanteeseen t on eräs ratkaisu tehtävään
  else
    for all tilanteen t lapset $t_1, t_2, t_3, \ldots$ **do**
      BackTrack(t_i)
    end for
  end if.

- Pysähtymisen takaamiseksi olisi vältettävä äärettömät polut.
- Yksi (hidas) tapa on jättää pois ne lapsitilanteet t_i jotka jo ovat polulla juuritilanteesta isätilanteeseen t.
- Voi olla mahdotonta välttää jos erilaisia tilanteita todella on äärettömästi.



Jos halutaan ratkaisuista

jokin niin voidaan lopettaa koko rekursio ensimmäiseen loppuehdon täyttävään ratkaisuun polku(t).

jokainen niin rekisteröidään kukin löydetty ratkaisu polku(t) ja jatketaan rekursiota.

paras niin käydään läpi jokainen ja rekisteröidään löydetyistä paras.

Koska menetelmä ei takaa järjestystä jossa ratkaisut löydetään.

+ Helppo ohjelmoida.

+ Nopea tuottamaan erilaisia tilanteita t.

+ Tehokas käyttämään muistia.

– Etenee sokeasti ottamatta huomioon mahdollista tehtäväkohtaista lisäinformaatiota.

7.2 Rajoittava etsintä

Käytetään kalvojen 7.1 peruuttavaa etsintää etsimään *halvin* ratkaisu.

Tehostuu, kun *jätetään kasvattamatta ne solmut, jotka ovat jo nyt huonompia kuin paras tähän mennessä löydetty ratkaisu* p_{min}.

- Ylläpidetään p_{min} globaalina tietona.

  Alkuarvona "ratkaisu" jonka hinta on $+\infty$.

- Olkoon meni$(t, t') \geq 1$ sen operaation hinta, jolla tilanteesta t saa naapuritilanteen t'.

procedure BranchBound(t: **tilanne**, g: $\mathbb{R}$) **is**
  if hinta g on halvempi kuin ratkaisun p_{min} **then**
    if tilanne t täyttää loppuehdon **then**
      p_{min} := polku(t)
    else
      for all tilanteen t lapset t' **do**
        BranchBound(t', g + meni(t, t'))
      end for
    end if
  end if.

- Sitä tehokkaampi mitä aikaisemmin ratkaisun p_{min} hinta putoaa.

- Etsitään siis rajua pudotusta kalvojen 10.3 ahneella ajattelutavalla.

- Keksitään tätä varten jokin menee(t) = *arvio* siitä kuinka lähellä tilannetta t on lähin sellainen tilanne, joka täyttää tehtävän loppuehdon.

  Ns. *heuristinen* funktio.

- Silloin lapset t' voidaan käydä läpi arvioiden menee(t') mukaan kasvavassa järjestyksessä.

Näin saadaan *rajoittava etsintä* (Branch and Bound).

[6, luku 11.2]

Repunpakkausongelma: Saat kerätä kaupasta repullisen tavaraa ilmaiseksi...

$ub = v + (W - w) \cdot d$
$W =$ repun kantavuus
$v =$ repullisen nykyinen hinta
$w =$ repun nykyinen paino
$d =$ korkein hyllyssä olevan tavaran kilohinta.

[6, kuva 11.8]

Tähän ongelmaan palataan kalvoilla 10.3.2.

Lyhyin Hamiltonin kehä: Kierrä teitä pitkin kaikissa kaupungeissa mahdollisimman lyhyesti käymättä missään kahdesti.

$lb =$ summa

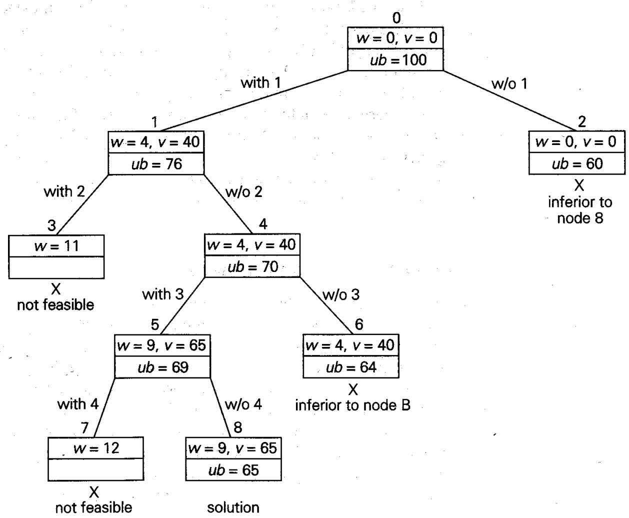
- reitille jo valittujen teiden pituudet
- reitin kummallekin päätepisteelle *puolet* lyhyimmästä tiestä (eteenpäin)
- reitin ulkopuolisille kaupungeille puolet *kahdesta* lyhyimmästä tiestä (toista sisään, toista ulos).

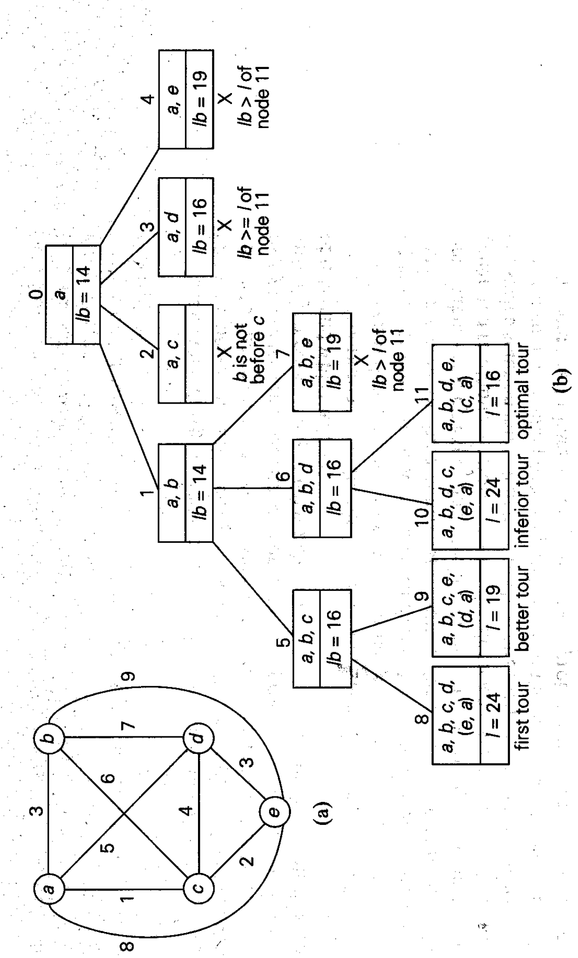
[6, kuva 11.9]

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.

At the root of the state-space tree (see Figure 11.8), no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal to 0. The value of the upper bound computed by formula (11.1) is $100. Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 and $40, respectively; the value of the upper bound is 40 + (10 − 4) ∗ 6 = $76.





- Rajoittaminen puree vielä tehokkaammin, jos verrataankin *summaa* $g +$ menee(t) ratkaisun $p_{\min}$ hintaan.
- menee(t) *olkoon optimistinen* eli menee$(t) \leq$ todellinen etäisyys tilanteesta t lähimpään lopputilanteeseen.

  Muuten tämä tehostus voi jättää kasvattamatta sen etsityn halvimman ratkaisun!
- Erityisesti siis menee$(t) = 0$ kun t täyttää loppuehdon.
- Esimerkiksi *kolmioepäyhtälö*

  $$\text{menee}(t) \leq \text{meni}(t, t') + \text{menee}(t')$$

  takaa optimismin.
- Etsintäalgoritmi voi itsekin pakottaa kolmioepäyhtälön voimaan.

  Silloin pessimistinen arvo menee(t) ei ohjaa etsintää — ei oikeaan eikä väärään suuntaan.

procedure BnB(t: **tilanne**, g: $\mathbb{R}$, h: $\mathbb{R}$) **is**
  if $\boxed{g+h}$ on halvempi kuin ratkaisun p_{min} hinta **then**
    if tilanne t täyttää loppuehdon **then**
      p_{min} := polku(t)
    else
      for all tilanteen t lapset t' kasvavassa järjestyksessä arvioiden $b = \boxed{\min(\text{menee}(t'), h - \text{meni}(t,t'))}$ suhteen **do**
        BnB($t', g + \text{meni}(t,t'), b$)
      end for
    end if
  end if.

- Etsintä on sitä tehokkaampaa mitä tarkemman arvion optimistinen menee(t) antaa.

  Kannattaa siis valita *optimisteista pessimistisin*.

- Joskus keksitäänkin *perhe*

  $$\text{menee}_1(t), \ldots, \text{menee}_k(t)$$

  optimistisia arvioijia, joista kukin ennustaa hyvin tietyn laisia tilanteita t mutta huonosti muita.

  Silloin niiden hyvät puolet voidaan yhdistää

  $$\text{menee}(t) = \max(\text{menee}_1(t), \ldots, \text{menee}_k(t)).$$

- Näin voidaan

  1. ottaa yleiskäyttöinen etsintäalgoritmi
  2. sijoittaa siihen tehtäväkohtaiset loppuehto- ja lastenkasvatusaliohjelmat
  3. lisätä erilaisia mieleen tulevia arvioijia

  ellei keksitä juuri tähän tehtävään sopivaa erikoisalgoritmia.

7.3 A^*: optimaalinen etsijä

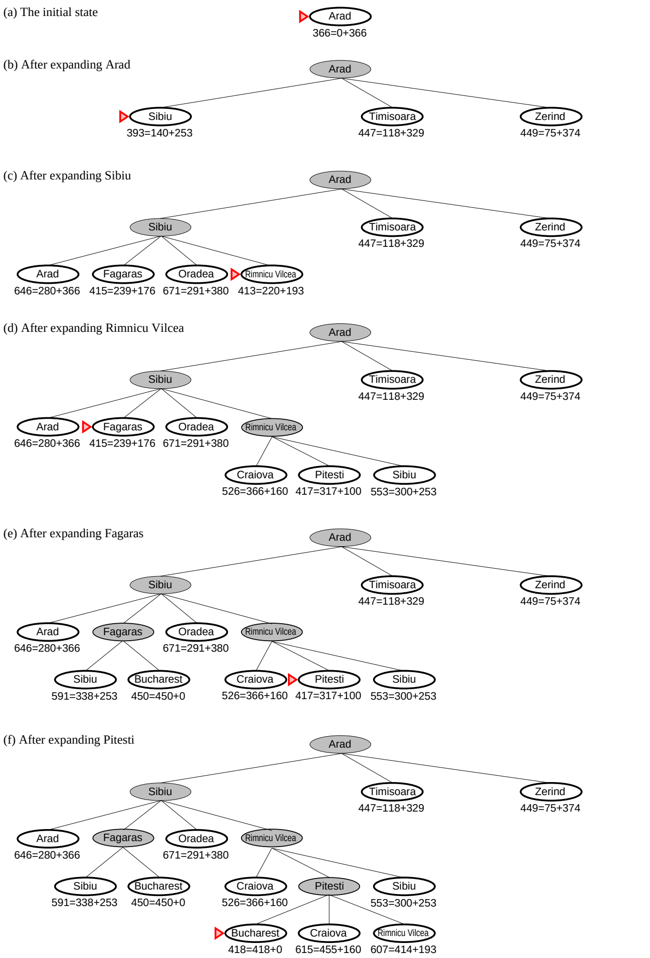
Kalvojen 7.2 BnB-etsinnässä jatkovaihtoehto valittiin *paikallisesti* nykyisen tilanteen t lapsista t'.

Jos valitaankin *globaalisti* kaikkien vielä lapsettomien solmujen joukosta, niin saadaan tekoälyalgoritmi A^* [7, luku 4.1 ja kuva 4.3].

Tämä voidaan tehdä viemällä rekursiokutsun sijasta sen parametrit $\langle t', g', h' \rangle$ kalvojen 2.3 kekoon avaimella $g' + h'$.

  Alusta keko K kolmikolla $\langle t_0, 0, \text{menee}(t_0) \rangle$ missä t_0 on etsinnän alkutilanne;
  while $K \neq \emptyset$ **do**
    Poista keosta K avaimeltaan pienin $\langle t, g, h \rangle$;
    if tilanne t toteuttaa loppuehdon **then**
      return polku(t)
    else
      for all tilanteen t lapset t' **do**
        Lisää kekoon K kolmikko $\langle t', g + a, \min(\text{menee}(t'), h - a) \rangle$ missä $a = \text{meni}(t, t')$
      end for
    end if
  end while;
  return "Tehtävällä ei ole ratkaisuja!".

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374



+ *Paras mahdollinen* näiden periaatteiden mukainen etsintäalgoritmi:

Jos toinenkin algoritmi A'

- toimii laajentamalla operaatiopolkuja

$$t_0 \xrightarrow{\mathrm{op}_1} t_1 \xrightarrow{\mathrm{op}_2} t_2 \xrightarrow{\mathrm{op}_3} \cdots \xrightarrow{\mathrm{op}_m} t_m$$

- tekee laajennuspäätökset luvuilla

$$\mathrm{menee}(t_m, t_{m+1}) + \sum_{0 \le i < m} \mathrm{meni}(t_i, t_{i+1})$$

- löytää aina parhaan polun

niin A' käsittelee (oleellisesti) ainakin yhtä suuren polkupuun kuin A^*.

– Mutkikkaampi ohjelmoida kuin kalvojen 7.2 rajoittava etsintä.

Keko K kasvaa dynaamisesti kalvojen 2.3.4 tapaan.

– Kuluttaa paljon työmuistia!

7.4 *IDA**: vähämuistinen A^*

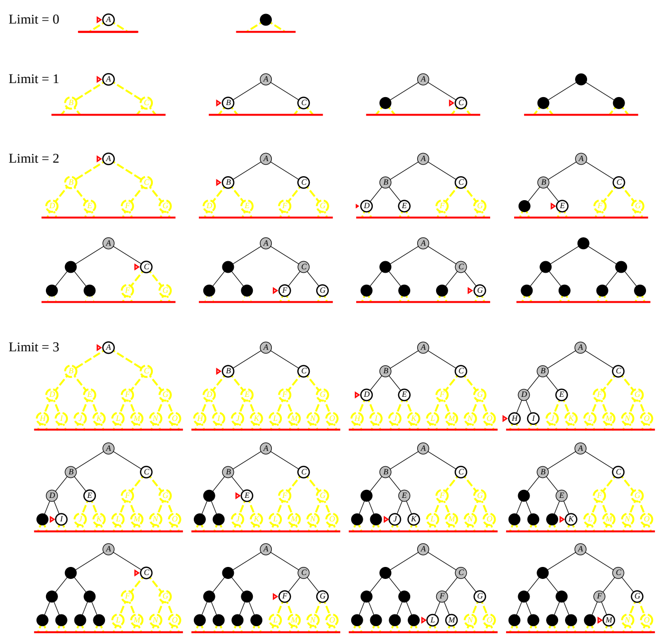
Kalvojen 7.3 algoritmista A^* on kehitetty enemmän aikaa mutta vähemmän muistia kuluttavia versioita.

Niistä yksinkertaisin on *toistavasti syventävä* A^* (Iterative Deepening A^*, *IDA**):

- Etsitään rekursiivisesti
- kunnes *ylitetään ennalta asetettu hinta-arvioraja* f_{min}

  (tai löydetään ratkaisu).
- Samalla löytyy pienin $f'_{\mathrm{min}} > f_{\mathrm{min}}$, jolla etsintä olisi vielä jatkunut.
- Etsitään *alusta alkaen uudelleen* rajaan f'_{min}.

[7, sivut 78–79 ja 101 sekä kuva 3.15]



f'_{min} := 0;
while $f'_{\text{min}} < +\infty$ **do**
  f_{min} := f'_{min}; f'_{min} := $+\infty$;
  DLS(t_0, 0)
end while;
Lopeta koko etsintä koska ratkaisua ei ole.

procedure DLS(t: **tilanne**, g: $\mathbb{N}$) **is**
  h := menee(t);
  if $g + h > f_{\text{min}}$ **then**
    f'_{min} := min($f'_{\text{min}}, g + h$)
  else if tilanne t täyttää loppuehdon **then**
    Lopeta koko etsintä ratkaisuun polku(t)
  else
    for all tilanteen t lapset t' **do**
      DLS($t', g +$ meni(t, t'))
    end for
  end if.

- Jos menee(t') on aina 0, niin iteratiivisesti syventävä ei-heuristinen perushaku.

- Pienempi yläraja η kuin $+\infty$ **while**-silmukassa katkaisee koko etsinnän alle hintaan η.

\+ Jos on ratkaisuja, niin *IDA** löytää parhaan.

\+ Muistissa (rekursiopinossa) on vain yksi polku kerrallaan.

\+ Helppo ohjelmoida.

! Kalvojen 7.3 algoritmin *A** edut ilman haittoja.

? Sitä tehokkaampi mitä vähemmän erilaisia arvoja $(g + h)$ esiintyy.

  – Voi olla ongelma liukulukuhinnoilla.

  \+ Kilpailuissa on (yleensä) kokonaisluvut.

– Tuottaa saman tilanteen t lapset t' monta eri kertaa.

  - Ongelma vain jos puussa paljon 1-lapsisia solmuja.

  - Muuten toiston lisärasite etenemistä pienempi.

8 Verkkojen läpikäynneistä

[1, liite B.4 ja kuva B.2; 8, luku 9]

- *Verkko* (graph) G koostuu 2 joukosta:

  Solmujen (node, vertex*) joukosta $V(G)$.

    Oletamme solmut numeroiduiksi $1, 2, 3, \ldots, |V(G)|$.

  Kaarten (edge, arc) joukosta $E(G) \subseteq V(G) \times V(G)$.

    – Yksi kaari $\langle p, q \rangle \in E(G)$ on siis *solmupari*.

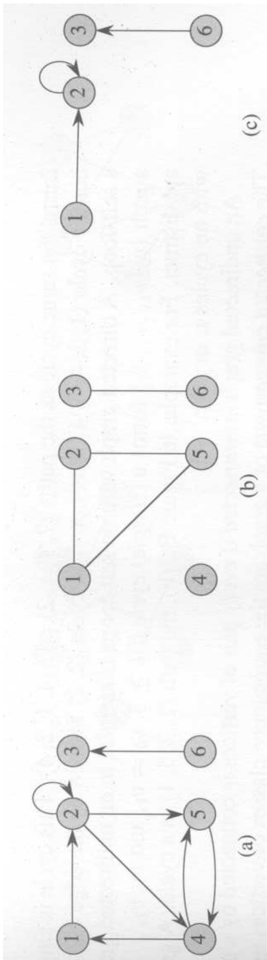
    – Merkitään kaarta $p \to q$.

- Verkkoja käytetään algoritmeissa

  ongelman mallintamisen abstrakteina suunnitteluvälineinä

  informaation tallentamisen konkreettisina tietorakenteina.

*Monikko "vertices".



Kaaren $p \to q$ solmut ovat *vierekkäisiä* (adjacent) eli *vierussolmuja*.

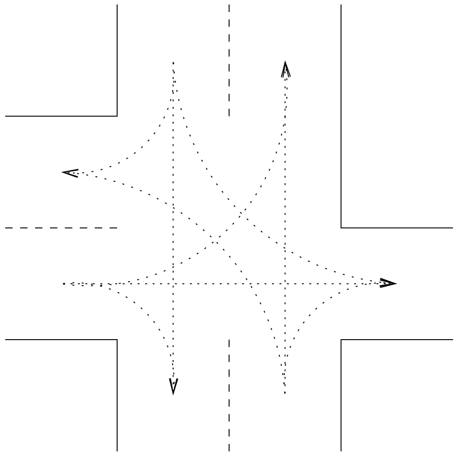
Lähtösolmu (source) p *edeltää* solmua q.

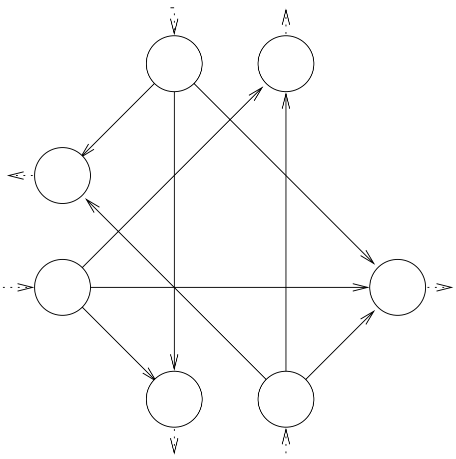
Maalisolmu (sink) q *seuraa* solmua p.

Solmun lähtöaste (outdegree, out-degree) on *siitä lähtevien* kaarten lukumäärä.

Tuloaste (indegree, in-degree) on *siihen tulevien* kaarten lukumäärä.

- Tästä perusteemasta on paljon muunnelmia eri käyttötarkoituksiin.
- Esimerkkinä kaupungin *katuverkko:*
  - Kukin kaari esittää sellaista ajokaistan pätkää, joka auton on ajettava alusta loppuun, koska välillä ei voi kääntyä.
  - Kukin solmu esittää sellaista kohtaa, jossa auto voi valita seuraavan suuntansa useasta eri ajokaistan pätkästä.





Polku eli **reitti** on kaarijono

$$p_0 \to p_1 \to p_2 \to \cdots \to p_m$$

missä jokainen solmu p_{i+1} siis seuraa solmua p_i.

Solmusta $v \in V(G)$ **saavutettavissa** (reachable, accessible) ovat ne solmut $w \in V(G)$ joihin on polku solmusta v.

Yksinkertainen (simple) polku on sellainen, jossa mikään solmu ei toistu.

Sykli eli **kehä** (cycle) on sellainen polku

- joka on muuten yksinkertainen
- mutta sen alku- ja loppusolmu ovat samat $p_1 = p_m$.

DAG (**D**irected **A**cyclic **G**raph) on suosittu lyhenne *suunnatulle syklittömälle verkolle*.

- Tällaisia verkkoja voidaan käydä läpi *systemaattisesti* eli pelkästään kaaria seuraten

  syvyyssuunnassa kalvoilla 8.4

  leveyssuunnassa kalvoilla 8.5

  välittämättä verkossa mahdollisesti olevasta lisäinformaatiosta.

- Lisäinformaatiota selitetään kalvoilla 8.1

- Nämä läpikäynnit tuottavat π-puun:
  - solmun $v \in V(G)$ kenttä $\pi[v] =$ se solmu u, josta solmuun v tultiin ensimmäisen kerran
  - kulkemalla vastaava kaari

    $$u \to v \in E(G)$$

  Se on kalvojen 7 polkupuu.

- Nämä läpikäynnit ovat *tehokkaita:*

  $$O(|V(G)| + |E(G)|) \text{ askelta.}$$

8.1 Painotetut verkot

- Kalvoilla 8 määriteltiin *painottamaton* (unveighted) verkko.
- *Painotetun* (weighted) verkon G jokaisella kaarella on oma kaari*paino* (edge *weight*) w:

$$p \xrightarrow{w} q \in E(G).$$

- Paino w on *yleensä luku* kuten
  - pituus
  - siirtokapasiteetti
  - ...

Kun painot esittävät kaarten pituuksia niin voidaan puhua solmujen välisistä *etäisyyksistä* polkuja pitkin.

Polun pituus on silloin kaaripainojen w_i summa

$$w_1 + w_2 + w_3 + \cdots + w_m.$$

8.2 Suuntaamattomat verkot

- Kalvoilla 8 määriteltiin *suunnatut* (directed) verkot:

  Verkon G kaarella

$$p \to q \in E(G)$$

  on *luonteva kulkusuunta* lähtösolmusta p maalisolmuun q.

- *Suuntaamattomassa* (undirected) verkossa kaarella ei ole suuntaa

$$\{p, q\} \in E(G)$$

  eikä sen päitä p ja q erotella toisistaan.

- Merkitään suuntaamatonta kaarta

$$p — q.$$

8.3 Verkkojen talletusrakenteita

Vieruslistat (adjacency lists):

- Taulukko $A[0 \ldots |V(G)| - 1]$ jonka paikka $A[p]$ sisältää solmun $p \in V(G)$
  - vieruslistan
  - solmukohtaiset muut tiedot (jos on).
- Solmun $p \in V(G)$ vieruslista sisältää siitä lähtevät kaaret:
  - Kaarelle $p \xrightarrow{w} q \in E(G)$ parin $\langle w, q \rangle$ (sopivana tietueena).
  - Painottamattomalla G ei kenttää w.
- Hyvä tapa kun:
  - Verkkoa G käsitellään *solmu kerrallaan* — kuten useimmat algoritmit.
  - Verkko G on *harva* (sparse), eli sisältää vain "pienen" osan kaikista mahdollisista kaarista, eli suhde

$$\frac{|E(G)|}{|V(G)|^2} \quad \text{on pieni.}$$

Vierusmatriisi (adjacency matrix):

- 2-ulotteinen matriisi $B[1 \ldots |V(G)|][1 \ldots |V(G)|]$.
- $B[p][q] =$ solmujen p ja q välinen tieto:

  Painottamattomalla verkolla G *totuusarvo*

$$p \to q \in E(G)?$$

  Painotetulla verkolla G
  - vastaava kaaripaino w, jos

$$p \xrightarrow{w} q \in E(G)$$

  - Jokin sopiva "mahdoton paino" $\aleph$, jos

$$p \xrightarrow{w} q \notin E(G).$$

    Jos esimerkiksi lukuarvoiset painot w esittävät etäisyyksiä, niin $\aleph = +\infty$ on luonteva valinta.

- Hyvä tapa kun:
  - Verkkoa G käsitellään *kaari sieltä, toinen täältä*.
  - Verkko G onkin *tiheä* (dense).

Suuntaamaton verkko G voidaan esittää *molempiin suuntiin suunnattuna*, eli kaariparina

$$p \rightleftarrows q \in E(G)$$

jokaiselle särmälle solmujen p ja q välillä.

Tapoja, jotka eivät kaksinkertaista särmiä:

***Kolmio*matriisina** (*triangular* matrix)

$$C[\underbrace{1 \ldots |V(G)|}_{\text{indeksi } i}][1 \ldots i]$$

missä solmujen p ja q välinen särmä on paikassa $C[\max(p,q)][\min(p,q)]$.

***Moni*listoina** (*multi*lists) joissa solmujen p ja q välisen särmän lista-alkiossa on *molemmat* solmut ja listojen $\max(p,q)$ ja $\min(p,q)$ osoittimet.

[1, kuvat 22.1 ja 22.2; 6, kuva 1.8; 8, luku 9]

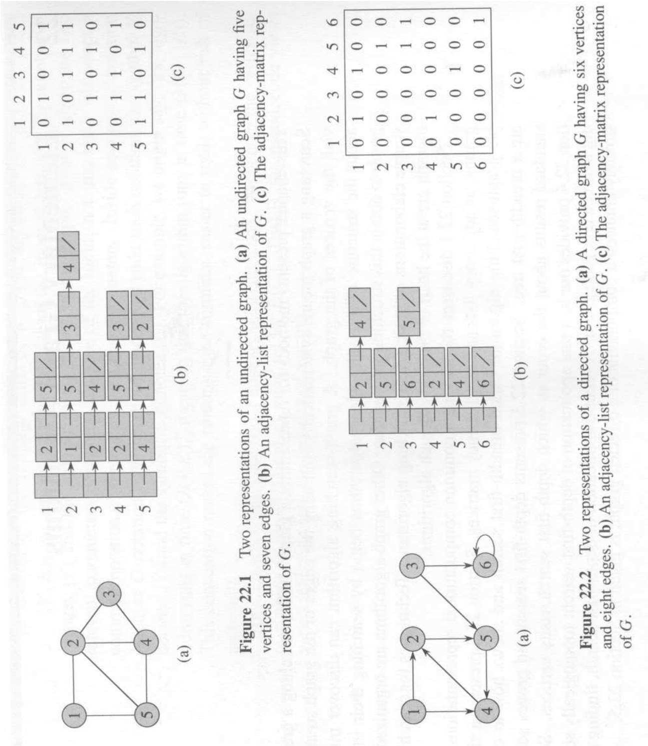
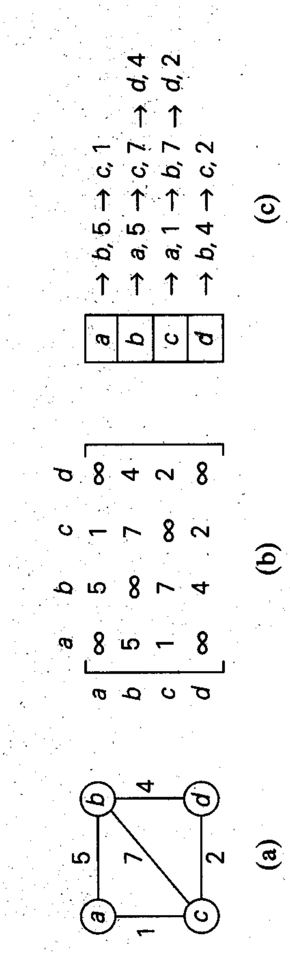


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



8.4 Läpikäynti syvyyssuunnassa

Verkon G *syvyyssuuntainen läpikäynti* (Depth-First Search, DFS) yrittää aina edetä rekursiivisesti verkon tutkimattomaan osaan ja peruuttaa vasta kun tämä on mahdotonta.

Verkon solmu on

VALKEA kunnes siihen edetään ensimmäisen kerran

MUSTA kun kaikki siitä lähtevät kaaret on käsitelty

HARMAA väliajan kun sen käsittely on kesken.

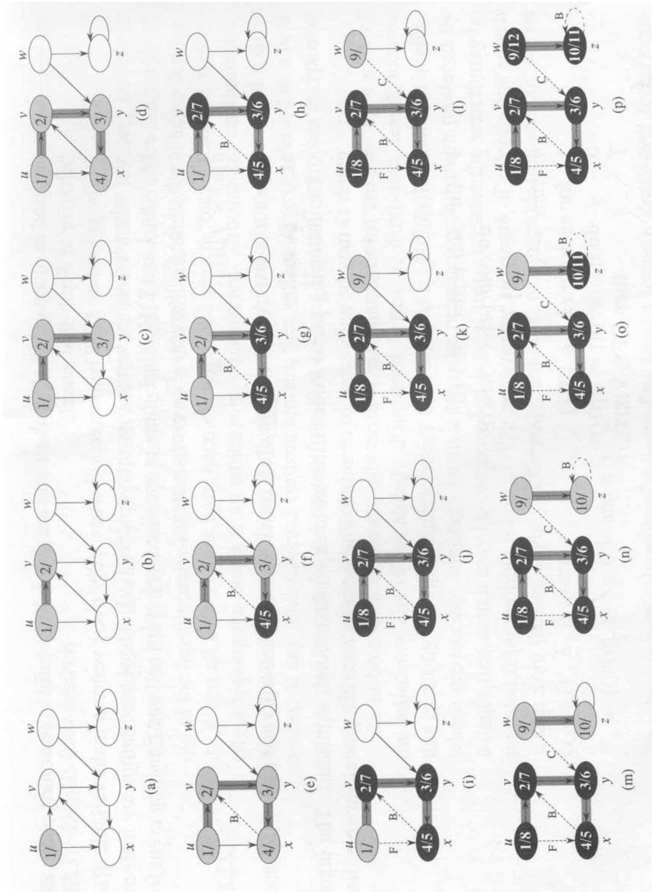
Lisäksi voidaan muistaa *milloin* väri vaihtui:

- alku$[u]$ = valkeasta harmaaksi
- loppu$[u]$ = harmaasta mustaksi.

[1, luku 22.3 sekä kuvat 22.4 ja 22.5; 6, luku 5.2]

for all $r \in V(G)$ **do**
  tummuus[r] **:=** VALKEA**;**
  $\pi[r]$:= NULL
end for;
Globaali kello **:=** 0**;**.
for all $r \in V(G)$ **do**
  if tummuus[r] $=$ VALKEA **then**
    DFS(r)
  end if
end for.

procedure DFS(u: $V(G)$) **is**
  tummuus[u] **:=** HARMAA**;**
  kello **:=** kello$+$1**;**
  alku[u] **:=** kello**;**
  for all $u \to v \in E(G)$ **do**
    if tummuus[v] $=$ VALKEA **then**
      $\pi[v]$ **:=** u**;**
      DFS(v)
    end if
  end for;
  tummuus[u] **:=** MUSTA**;**
  kello **:=** kello$+$1**;**
  loppu[u] **:=** kello**.**



Syvyyssuuntainen läpikäynti luokittelee kaaren $u \to v \in E(G)$:

Puukaari (Tree Edge): lähtösolmu u on tulosolmun v *isä* π-puussa.

Eli kaarta käsiteltäessä tummuus[v] $=$ VALKEA.

Kaari taakse (Back Edge): tulosolmu v on lähtösolmun u *(esi-)isä* π-puussa.

Erikoistapauksena $u = v$ eli 1 kaaren silmukka solmun u umpäri.

Eli kaarta käsiteltäessä tummuus[v] $=$ HARMAA.

Kaari eteen (Forward Edge): lähtösolmu u on tulosolmun v *esi-isä* π-puussa.

Eli kaarta käsiteltäessä tummuus[v] $=$ MUSTA ja alku[u] $<$ alku[v].

Kaari sivulle (Cross Edge) muuten, eli u ja v ovat eri π-(ali)puissa.

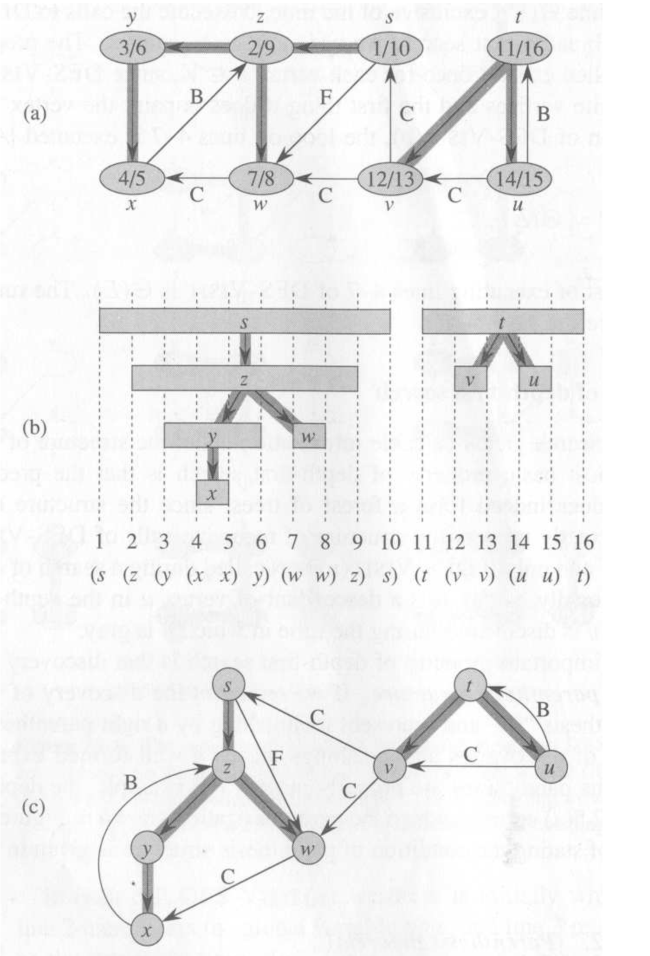
Eli kaarta käsiteltäessä tummuus[v] $=$ MUSTA ja lopuksi alku[u] $>$ alku[v].

Eri π-puut voidaan vielä erottaa toisistaan:

- Liitetään joka solmuun $u \in V(G)$ kenttä juuri[u] $=$ se pääohjelman juurisolmu r josta tähän solmuun u päästiin rekursiokutsulla DFS(r).
- Kenttä juuri[u] voidaan asettaa heti kun solmu u harmaantuu.
- **Kaari sivulle** $u \to v \in E(G)$ kulkee *samassa* π-puussa kun juuri[u] $=$ juuri[v].
- **Kaari sivulle** $u \to v \in E(G)$ *puusta toiseen järjestää ne aikajärjestykseen:*

$$\text{loppu[juuri}[v]] < \text{alku[juuri}[u]].$$

Suuntaamattomassa verkossa on vain **puukaaria** ja **kaaria taakse**.



8.4.1 Topologinen lajittelu

Suunnatun kehättömän verkon G *topologinen lajittelu* (Topological Sort) on sen solmujen listaus sellaisessa järjestyksessä $v_1, v_2, v_3, \ldots, v_{|V(G)|}$ joka *noudattaa nuolia*:

$$\text{Jos } v_i \to v_j \in E(G), \text{ niin } i < j.$$

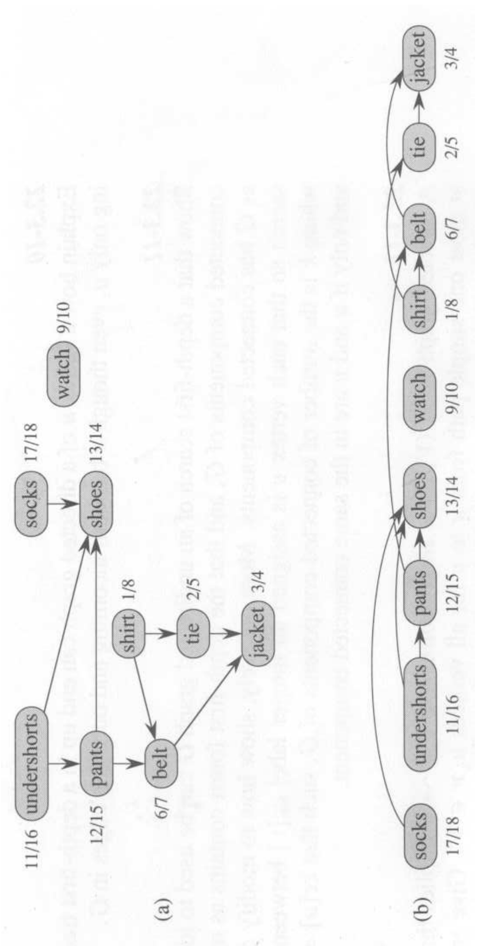
Löytyy kalvojen 8.4 syvyysläpikäynnillä kun *solmu lisätään järjestyksen alkuun vasta kun kaikki sen jälkeläiset on jo lisätty*.

1. Tee DFS(G) siten että solmu $u \in V(G)$ viedään (aluksi tyhjään) vastauspinoon P kun loppu[u] asetetaan.

2. Jos askeleen 1 aikana löytyi **kaari taakse**, niin G ei ollutkaan syklitön.

3. Muuten P on (eräs) topologinen järjestys.

[1, luku 22.4 ja kuva 22.7; 6, luku 5.3]



- Joskus voidaan määritellä verkon G solmuille $u \in V(G)$ jokin arvo $f(u)$ sen seuraajasolmujen $u \to v \in E(G)$ arvojen $f(v)$ perusteella.

- Se ei ole kehämääritelmä, jos G on kehätön.

- Esimerkiksi *pisin polku* solmusta u on
$$f(u) = \max\left\{0, w + f(v) : u \xrightarrow{w} v \in E(G)\right\}.$$

- Arvo $f(u)$ voidaan laskea kun kaikki siihen tarvittavat arvot $f(v)$ on laskettu.

- Siis juuri topologisessa järjestyksessä!

- Pisin yksinkertainen polku *kehällisessä* verkossa taas on $\mathbf{NP}$-täydellinen [4, ongelma ND23] eli vaikea tehtävä!

- Esimerkkinä *projektinhallinta*.

- Solmu on rakennusurakan taitekohta kuten "perustukset aloitetaan", "seinäelementit pystytetty",...

- Kaari tarkoittaa riippuvuutta kuten "perustusten pitää olla valmiit ennen kuin seinäelementtejä aletaan pystyttää".

- Kaaripaino perustusten aloittamisesta niiden valmistumiseen on sen kesto.

- Pisin polku on *kriittinen:* Jos jokin sillä oleva työvaihe myöhästyy aikataulustaan, niin koko urakan valmistuminen siirtyy myöhäisemmäksi.

- Pisimmän polun pituus kertoo, kauanko urakkaan täytyy varata aikaa siinäkin tapauksessa, että se etenee mahdollisimman hyvin.

8.4.2 Vahvasti yhtenäiset komponentit

- Suunnatun verkon G *vahvasti yhtenäinen komponentti* (Strongly Connected Component, SCC) on mikä tahansa mahdollisimman suuri aliverkko H joka on vahvasti yhtenäinen:

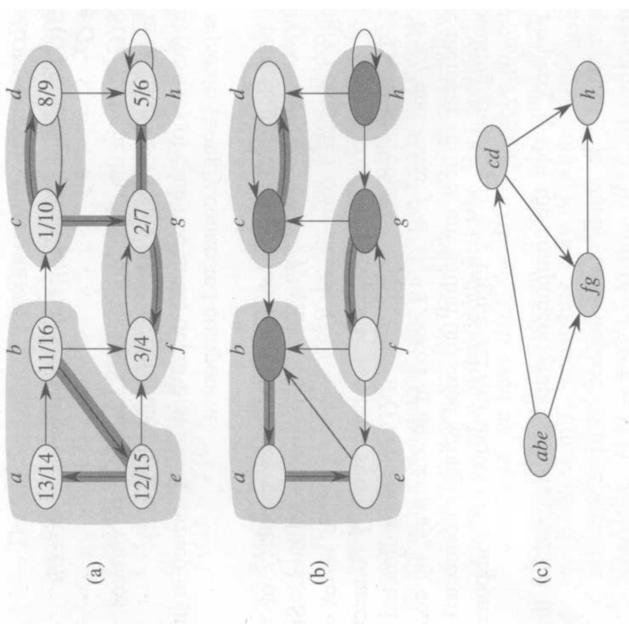
  Kahden solmun $p, q \in V(H)$ välillä on aina polku $p \to \cdots \to q \in E(H)$.

- Jos kaikki samaan komponenttiin kuuluvat solmut sulautetaan yhdeksi solmuksi, niin tuloksena on suunnattu kehätön verkko G^{SCC}.

- Nämä komponentit voidaan löytää tekemällä kalvojen 8.4 syvyyssuuntainen läpikäynti *kaksi* kertaa:

  1. ensin *eteen*päin
  2. sitten *takse*päin.

[1, luku 22.5 ja kuva 22.9]



1. Tee DFS(G) siten että solmu $u \in V(G)$ viedään (aluksi tyhjään) pinoon P kun loppu[u] asetetaan.

   Eli lajittele G topologisesti kalvoilta 8.4.1 vaikka se ei olekaan syklitön.

2. Olkoon G^T verkon G *transpoosi*

   $$E(G^T) = \{q \to p \colon p \to q \in E(G)\}$$

   eli käännetään nuolten suunnat.

   Käytännössä kannattaa luoda syöteverkosta G kaksi kopiota:

   (a) G itse jossa on saadut syötekaaret

   (b) G^T jossa on syötekaarten vastakaaret.

3. Tee DFS(G^T) siten että sen pääsilmukassa solmut r poimitaan pinosta P sen antamassa järjestyksessä.

4. Askeleen 3 π-puut ovat verkon G vahvasti yhtenäiset komponentit.

   Komponenteista voi pitää kirjaa kentillä juuri[u].

- Edellinen algoritmi
  - + on melko helppo nähdä oikeaksi
  - – vaatii syöteverkon G transpoosin G^T askeleessa 2.
- Päinvastainen algoritmi saadaan kalvojen 8.4 syvyyssuuntaisesta läpikäynnistä seuraavin muutoksin:
  - Vain alkuajat alku$[u]$ kiinnostavat

    mutta sovitaan lisäksi alku $= +\infty$ (rivi 17) silloin kun solmu u on jo luokiteltu komponenttiinsa.
  - Lisäksi kiinnostaa $m =$ pienin alku$[v]$ joka löytyy kulkemalla ensin puukaaria alas ja sitten yhtä takakaarta ylös (jos sellainen on).
  - Komponentti

    havaitaan ehdosta $m =$ alku$[u]$ (rivi 14)

    tulostetaan käyttäen (aluksi tyhjää) apupinoa S.


```
function SCC(u: V(G)) is
1:  tummuus[u] := MUSTA;
2:  kello := kello+1;
3:  alku[u] := kello;
4:  m := alku[u];
5:  PUSH(S, u);
6:  for all u → v ∈ E(G) do
7:     if tummuus[v] = VALKEA then
8:        m' := SCC(v)
9:     else
10:       m' := alku[v]
11:    end if;
12:    m := min(m, m')
13: end for;
14: if m = alku[u] then
15:    repeat
16:       t := POP(S);
17:       alku[t] := +∞;
18:       Tulosta "Solmu t kuuluu solmun u
          komponenttiin."
19:    until t = u;
20: end if;
21: return m
```


- Seuraavassa kuvassa on esimerkkiverkko, jossa
  - solmut on käsitelty aakkosjärjestyksessä
  - puukaaret on piirretty yhtenäisillä nuolilla
  - muut kaaret on piirretty pistenuolilla.
- Solmuille saadaan seuraavat arvot:

	A	B	C	D	E	F	G	H	I	J	K	L	M
e	1	2	7	5	4	3	6	12	13	8	9	10	11
m	1	2	1	3	3	3	1	12	12	6	9	6	8

- Huomaa: Solmun H arvoksi m tuli

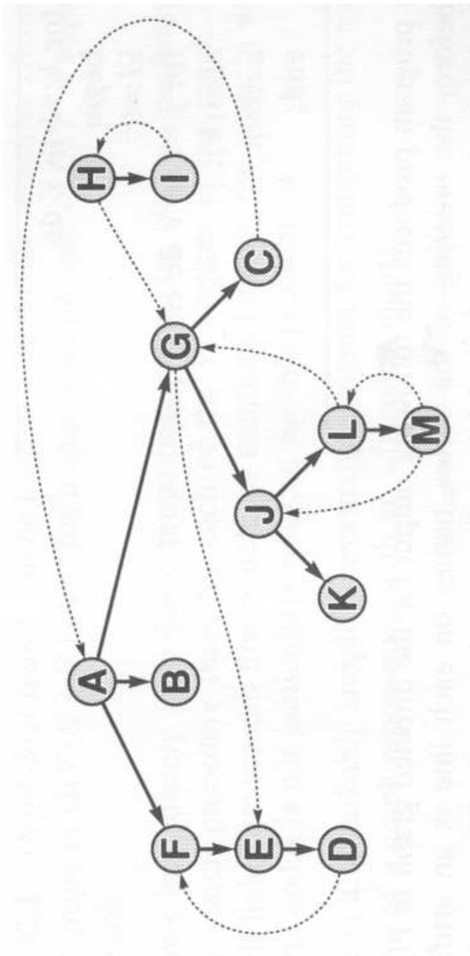
$$\text{alku}[H] = 12$$

eikä

$$\text{alku}[G] = 6$$

koska solmu G on jo luokiteltu ennen kuin solmua H käsitellään, joten sillä hetkellä onkin

$$\text{alku}[G] = +\infty.$$



8.4.3 Esimerkki: Rahankeruu

Suunnatun verkon G jokaisessa solmussa $u \in V(G)$ on saalis$[u]$ kultarahaa. Verkon kaarilla edetään nuolen suuntaan. Montako kultarahaa yhteensä voi kerätä?

1. Lasketaan G^{SCC} siten, että samalla lasketaan
$$\text{saalis}[C] = \sum_{u \in C} \text{saalis}[u]$$
jokaiselle komponentille $C \in V(G^{\mathsf{SCC}})$.

   Hoituu laskureilla saalis[juuri$[u]$].

2. Käydään komponentit $C \in V(G^{\mathsf{SCC}})$ läpi käänteisessä topologisessa järjestyksessä ja lasketaan
$$\text{tulos}[C] = \text{saalis}[C] + \max\left\{\text{tulos}[C'] : C \to C' \in E(G^{\mathsf{SCC}})\right\}$$
joka on nyt hyvin määritelty.

3. Vastataan suurin näin saatu tulos$[C]$.

8.5 Läpikäynti leveyssuunnassa

Verkon G *leveyssuuntainen läpikäynti* (Breadth-First Search) saadaan kalvojen 8.4 syvyyssuuntaisesta siten, että harmaat solmut viedäänkin (rekursio)pinon sijasta kalvojen 2.2 *jonoon*.

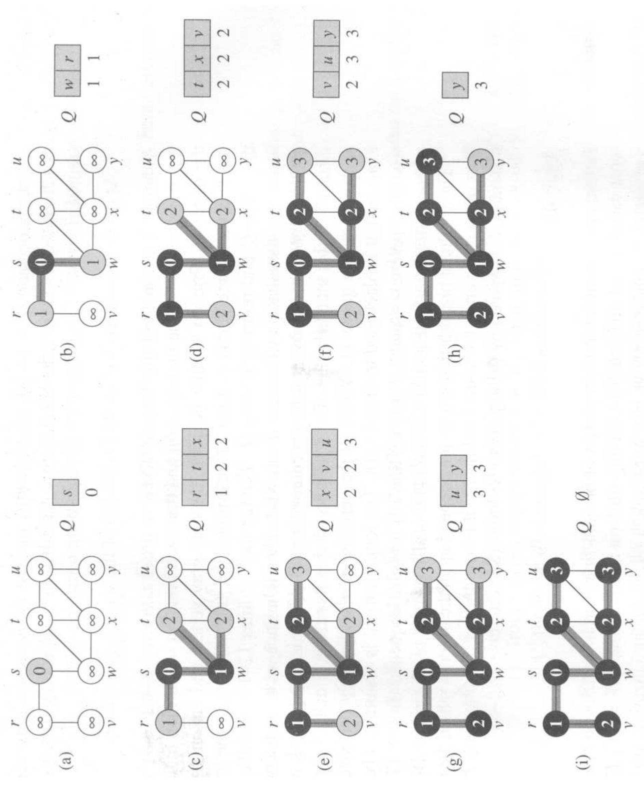
- Alku- ja loppuajoilla ei enää ole selkeää tulkintaa, jätetään ne pois.

- Polku
$$u \leftarrow \pi[u] \leftarrow \pi[\pi[u]] \leftarrow \pi[\pi[\pi[u]]] \leftarrow \cdots \leftarrow u$$
on (eräs) *vähäkaarisin* polku lähtösolmusta s solmuun u.

- Samalla voidaan laskea helposti syvyys$[u]$ = tämän polun pituus.

- Leveyssuuntaista läpikäyntiä sovelletaankin silloin, kun halutaan löytää mahdollisimman suoria reittejä lähtösolmusta s muualle verkkoon.

[1, luku 22.2 ja kuva 22.3; 6, sivut 165–168]



procedure BFS(G: **verkko**, s: **solmu**) **is**
  for all $r \in V(G) \setminus \{s\}$ **do**
    tummuus$[r]$ **:=** VALKEA**;**
    syvyys$[s]$ **:=** $+\infty$**;**
    $\pi[r]$:= NULL
  end for;
  tummuus$[s]$ **:=** HARMAA**;**
  syvyys$[s]$ **:=** 0**;**
  $\pi[s]$:= NULL;
  Alusta työjono Q solmulla s**;**
  while jonossa Q on yhä solmuja **do**
    Poista jonon Q alusta solmu u**;**
    for all $u \to v \in E(G)$ jolla
    tummuus$[v]$ = VALKEA **do**
      tummuus$[v]$ **:=** HARMAA**;**
      syvyys$[v]$ **:=** syvyys$[u] + 1$**;**
      $\pi[v]$ **:=** u**;**
      Lisää solmu v jonon Q loppuun
    end for;
    tummuus$[u]$ **:=** MUSTA
  end while.

Yksinkertaistus:	tummuus$[v]$ = VALKEA
jos ja vain jos	syvyys$[v] = +\infty$
jos ja vain jos	$v \neq s$ **and** $\pi[v]$ = NULL.

9 Verkkoalgoritmeista

[8, luku 10]

- Kun verkon G (kalvoilta 8) kaaren on *painotettu pituuksilla*, voidaan laskea erilaisia reittejä:

  lyhyimpiä kuten kalvoilla 9.1

  riittäviä kuten kalvoilla 9.3

- Kun painot ovatkin *kapasiteetteja*, niin voidaan laskea niiden yhteinen

  kokonaiskapasiteetti kuten kalvoilla 9.4.

9.1 Lyhyimmistä poluista

- Annettu:
  - (suunnattu tai suuntaamaton) verkko G
  - jonka jokaisella kaarella $u \xrightarrow{w} v \in E(G)$ on *pituus w numerona*.
- Halutaan laskea seuraavaa:
  - Annetaan vielä *lähtösolmu* $p_0 \in V(G)$ ja *kohdesolmu* $p_m \in V(G)$.
  - Laske lyhyin polku
    $$p_0 \xrightarrow{w_1} p_1 \xrightarrow{w_2} p_2 \xrightarrow{w_3} \cdots \xrightarrow{w_m} p_m$$
  - ja sen pituus
    $$\delta(p_0, p_m) = w_1 + w_2 + w_3 + \cdots + w_m.$$
- Jos verkossa G on *negatiivinen kehä*
  $$p_0 \xrightarrow{w_1} p_1 \xrightarrow{w_2} p_2 \xrightarrow{w_3} \cdots \xrightarrow{w_m} p_m \xrightarrow{w_0} p_0$$
  $$w_1 + w_2 + w_3 + \cdots + w_m + w_0 < 0$$
  niin $\delta(p_0, p_0) = -\infty$.

Dijkstran algoritmi (kalvot 9.1.1):

- Lähdetään *yhdestä* lähtösolmusta s.
- Halutaan etäisyydet $\delta(s, u)$ (ja polut) *kaikkiin muihin* solmuhin $u \in V(G)$.
- Kielletään negatiiviset *kaaret* $w_i < 0$.
- Esimerkki: Laske etäisyydet Helsingistä kaikkiin muihin Suomen kaupunkeihin.

Floydin algoritmi (kalvot 9.1.2):

- Halutaan etäisyydet $\delta(p, q)$ (ja polut) *jokaisesta* solmusta $p \in V(G)$ *jokaiseen* muuhun solmuun $p \in V(G)$.
- Kielletään negatiiviset *kehät* (kaaria saa olla).
- Esimerkki: Laske välimatkataulukko kaikkien Suomen kaupunkien välillä.

Bellmanin ja Fordin algoritmi (kalvot 9.1.3):

- Sama tavoite kuin Dijkstran algoritmissa.
- *Sallitaan* myös negatiiviset kehät (ja kaaret).

9.1.1 Dijkstran algoritmi

[1, luku 24.3 ja kuva 24.6; 6, luku 9.3]

- Jos etäisyydet ovat kokonaislukuja $w_i \geq 1$, niin lyhyimmät polut alkusolmusta s voidaan löytää seuraavasti:

  1. Korvataan jokainen alkuperäinen kaari
     $$u \xrightarrow{w} v \in E(G)$$
     kokonaisella polulla
     $$u \overbrace{\rightarrow \underbrace{\bigcirc \rightarrow \bigcirc \rightarrow \cdots \rightarrow \bigcirc}_{w-1 \text{ uutta } \textit{väli}\text{solmua}} \rightarrow}^{w \text{ uutta painotonta kaarta}} v.$$
  2. Sovelletaan kalvojen 8.5 leveyssuuntaista läpikäyntiä näin saatuun *lavennettuun* verkkoon G'.

- Tämän "ratkaisun" ongelmia:
  - Suurilla w toivottoman hidas ja muistisyöppö.
  - Entä muunlaiset painot w?

- Tämä "ratkaisu" värjää *lavennetun* verkon G' solmuja $p \in V(G')$ **musta**ksi kenttien syvyys[p] mukaan kasvavassa järjestyksessä.

- Tämä "ratkaisu" värjää *alkuperäisen* verkon G solmuja $q \in V(G) \subseteq V(G')$ **musta**ksi kenttien syvyys[q] mukaan kasvavassa järjestyksessä.

- Tämä "ratkaisu" takaa syvyys[q] $= \delta(s, q)$ sen perusteella, miten alkuperäinen verkko G lavennettiin verkoksi G'.

- Alkuperäiset solmut q värjätään **musta**ksi samassa järjestyksessä
  - jos seuraavaksi värjätään niistä se, jonka matka[q] = syvyys[q] on *pienin* mahdollinen
  - ilman tarpeettomia välisolmuja $V(G') \setminus V(G)$
  - myös muunlaisille kaaripainoille ≥ 0.

- Vaihtoehtoinen tapa hahmottaa Dijkstran algoritmi:
  - Solmut ovat kaupunkeja, kaaret teitä naapurikaupunkien välillä.
  - Kuningas haluaa lähettää pääkaupungistaan s viestin maansa kaikkiin kolkkiin.
  - Viestinviejät kulkevat
    * kaupungista toiseen teitä pitkin
    * keskenään samalla nopeudella
      $$c \text{ km/h}.$$
  - Kuningas lähettää yhtä aikaa pääkaupungista s viestinviejän sen jokaiseen naapurikaupunkiin.
  - Kuningas päivää näihin viesteihin niiden yhteisen lähettämishetken t.

- Dijkstran algoritmin hahmotus, jatkoa...
  - Kun viestinviejä x saapuu kaupunkiin u, niin hän kysyy sen asukkailta "Oletteko jo kuulleet kuninkaan viestin?"

    Ei: silmänräpäyksessä x
    * kuuluttaa viestin kaikille asukkaille
    * värvää heistä uudet viestinviejät

      yhden jokaiselle tästä kaupungista vievälle tielle
    * lähettää heidät matkaan.

    Kyllä (eli joku toinen viestinviejä y on ehtinyt tänne ensin): x ei tee mitään.

    Nyt x saa levätä, työ on hänen osaltaan ohi.
  - Kaupungin u etäisyys pääkaupungista s on silloin tietenkin
    $$c \cdot (t_u - t)$$
    missä t_u on se hetki, jolloin sen asukkaat kuulivat kuninkaan viestin ensimmäisen kerran.

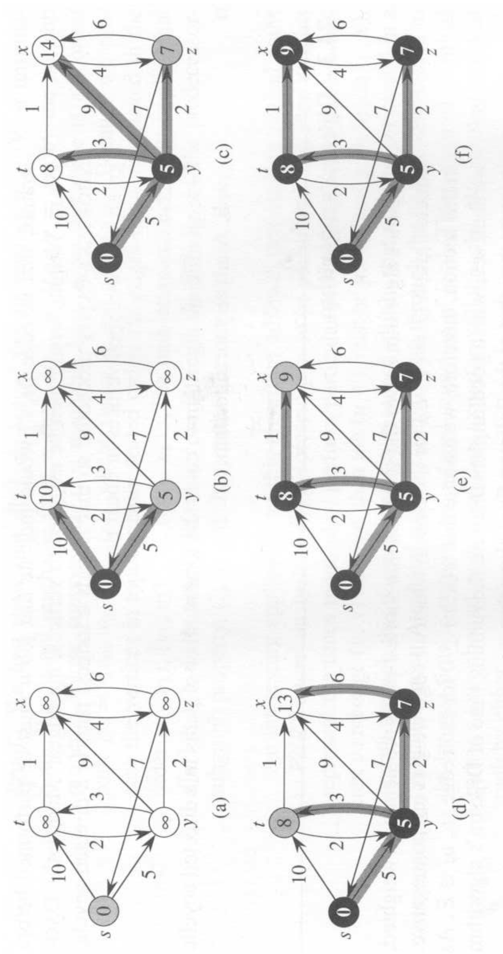
- Nyt Dijkstran algoritmin voi hahmottaa tämän viestinviennin simulaationa:
  - Simulaation kiinnostavat ajanhetket ovat nämä eri kellonajat t_u.
  - Kun ensimmäinen sanansaattaja x saapuu kaupunkiin u, niin voidaan laskea, milloin hänen värväämänsä sanansaattaja z saavuttaa oman kohdekaupunkinsa v:
    $$t_u + \frac{w(u,v)}{c}.$$
  - Näillä laskelmilla voidaan pitää yllä ja parantaa arvioita $d_v =$ milloin kuninkaan viesti saavuttaa kaupungin v.
  - Simulaatio etenee valitsemalla aina seuraavaksi pienimmän d_v, missä kaupunki v ei vielä ole kuullut kuninkaan viestiä, ja siirtymällä kellonaikaan $t_v = d_v$.
  - Laskutoimitusten helpottamiseksi voimme vielä valita

    nopeudeksi $c = 1$ ja päiväykseksi $t = 0$.

procedure Dijkstra$_1$(G: **verkko**, s: **solmu**)
  for all $r \in V(G) \setminus \{s\}$ **do**
    tummuus[r] := VALKEA;
    matka[r] := $+\infty$;
    $\pi[r]$:= NULL
  end for;
  tummuus[s] := HARMAA;
  matka[s] := 0;
  $\pi[s]$:= NULL;
  while harmaita solmuja on yhä jäljellä **do**
    Valitse sellainen harmaa solmu u jonka kenttä matka[u] on mahdollisimman pieni;
    for all $u \xrightarrow{w} v \in E(G)$ **do**
      c := matka[u] $+ w$;
      if tummuus[v] = VALKEA **then**
        tummuus[v] := HARMAA;
        matka[v] := c;
        $\pi[v]$:= u
      else if $c <$ matka[v] **then**
        matka[v] := c;
        $\pi[v]$:= u
      end if
    end for;
    tummuus[u] := MUSTA
  end while.

- Koska jokainen etäisyys $w \geq 0$, niin vastaus matka[u] ei enää muutu, kun solmu u on tullut **musta**ksi.
- Solmut u tulevat yhä **musta**ksi etäisyyden suhteen kasvavassa järjestyksessä.
- Valinta on helppo toteuttaa käymällä läpi kaikki (**harmaa**t) solmut.

  Silloin koko algoritmi vie aikaa
  $$O(|V(G)| \cdot |E(G)|) \text{ askelta.}$$
- Valintaa voi nopeuttaa:
  - Pidetään **harmaa**t solmut v kalvojen 2.3 (minimi)*keossa* kentän matka[v] suhteen.
  - Koko algoritmi vie enää aikaa
    $$O(\log_2(|V(G)|) \cdot |E(G)|) \text{ askelta.}$$
  - Koska **harmaa**n solmun v avainkenttä matka[v] voi *pienentyä*, tarvitaan kalvojen 2.3.3 *kahvoja*.



- Voimme pitää keossa myös valkeat solmut.

- Silloin keosta nousee solmu jolla matka$[u] = +\infty$, kun on käsitelty kaikki ne solmut, joihin alkusolmusta s pääsee.

procedure Dijkstra$_2$(G: **verkko**, s: **solmu**)
  for all $u \in V(G) \setminus \{s\}$ **do**
    matka$[u]$ **:=** $+\infty$;
    $\pi[u]$ **:=** NULL
  end for;
  matka$[s]$ **:=** 0;
  $\pi[s]$ **:=** NULL;
  Tee minimikeko kaikista solmuista $V(G)$ kentän matka suhteen;
  while keko ei ole tyhjä **do**
    Poista keosta sen pienin alkio u;
    for all $u \xrightarrow{w} v \in E(G)$ **do**
      c **:=** matka$[u] + w$;
      if $c <$ matka$[v]$ **then**
        matka$[v]$ **:=** c;
        $\pi[v]$ **:=** u;
        siftup(kahva$[v]$) kalvoilta 2.3.3
      end if
    end for
  end while.

- Vertaa kalvojen 7.3 algoritmiin A*:
  - Algoritmissa A* kaaripainot koostuivat heuristisesta ja todellisesta osasta

    kun taas Dijkstrassa vain todellisesta.
  - Algoritmissa A* eri reitit samaan solmuun tulkittiin eri ratkaisuiksi

    kun taas Dijkstrassa muistetaan joka solmulle vain paras reitti.
  - Algoritmi A* ei käsittele verkon solmua u ennen kuin se on luonut jonkin polun

    $$s \rightsquigarrow u$$

    mutta myös Dijkstra voidaan ohjelmoida siten.

- Tämä muotoilu tarvitsee kahvalliset keot.
  - Usein (esimerkiksi STL-kirjastossa) keot ovatkin kahvattomia.
  - Dijkstran algoritmi voidaan ohjelmoida kahvatta, jos keossa sallitaankin *samasta solmusta monta kopiota* eri avainarvoilla.

- Kekoon talletetaan nyt kolmikoita

  $$\langle m, v, p \rangle$$

  jotka edustavat alkuperäisen algoritmin operaatioita "on löydetty polku

  $$s \to \bigcirc \to \bigcirc \to \ldots \to \bigcirc \to p \to v$$

  jonka pituus on kekoavain m".

- Nyt matka$[u]$ asetetaan äärelliseksi vain kerran.

procedure Dijkstra$_3$(G: **verkko**, s: **solmu**)
  for all $u \in V(G)$ **do**
    matka$[u]$ **:=** $+\infty$
  end for;
  Tee minimikeko jossa aluksi on vain $\langle 0, s, \text{NULL} \rangle$;
  while keko ei ole tyhjä **do**
    Poista keosta sen pienin $\langle m, u, p \rangle$;
    if matka$[u] = +\infty$ **then**
      matka$[u]$ **:=** m;
      $\pi[u]$ **:=** p
      for all $u \xrightarrow{w} v \in E(G)$ **do**
        Vie kekoon $\langle m + w, v, u \rangle$
      end for
    end if
  end while.

9.1.2 Floydin algoritmi

[1, luku 25.2; 6, luku 8.2 ja kuva 8.7].

- Oletetaan solmut $V(G)$ luvuiksi $1, 2, 3, \ldots, n$ joilla voi indeksoida.

- Yritetään
  1. kirjoittaa
  2. ohjelmoida

  yhtälö kysytyille luvuille $\delta(i, j)$.

- Koska verkossa G ei ole negatiivisia kehiä, niin lyhyimmällä polulla

  $$i \to \underbrace{\bigcirc \to \bigcirc \to \bigcirc \to \cdots \to \bigcirc}_{\text{sisäsolmut}} \to j$$

  ei tarvitse toistaa mitään solmua.

- Sallitaan kuitenkin kehä eli $i = j$.

- Sellainen polku tai kehä on *yksinkertainen*.

- Siis lyhyin polku voidaan jakaa

$$\overbrace{i \to \bigcirc \to \cdots \to \bigcirc \to}^{\text{alkuosaan}} k \underbrace{\to \bigcirc \to \cdots \to \bigcirc \to j}_{\text{loppuosaan}} \qquad (2)$$

  missä

  - solmu k on välisolmuista *suurin*
  - alku- ja loppuosien välisolmut ovat $< k$
  - alku- ja loppuosat ovat lyhyimpiä polkuja.

- Voidaan siis määritellä induktiivisesti luvut $W(i, j, k)$ = lyhyimmän sellaisen polun pituus, joka

  - kulkee solmusta i solmuun j
  - käyttämättä sisäsolmuja

$$k + 1, k + 2, k + 3, \ldots, n.$$

- Silloin

$$\delta(i, j) = W(i, j, n).$$

- Saadaan induktiivinen määritelmä

$$\begin{aligned} W(i, i, 0) &= 0 \\ W(i, j, 0) &= \min\left\{w \colon i \xrightarrow{w} j \in E(G)\right\} \\ W(i, j, k) &= \min(W(i, j, k-1), \\ &\qquad W(i, k, k-1) \\ &\qquad\quad + W(k, j, k-1)) \end{aligned}$$

  missä

$$\min \emptyset = +\infty.$$

- Helppo ohjelmoida 3-ulotteisella matriisilla $W[i][j][k]$:

  alustus tasolle $k = 0$: $W[i][j][0] =$

  diagonaalilla 0

  muualla verkon G vierusmatriisi(sta)

  laskenta tasoille $1 \le k \le n$:

```
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      W[i][j][k] := min(W[i][j][k −
      1], W[i][k][k − 1] + W[k][j][k − 1])
    end for
  end for
end for.
```


- Laskenta vie

$$O(n^3) \qquad \text{askelta.}$$

- Seuraava taso $k + 1$ voidaan kirjoittaa edellisen tason k *päälle:*

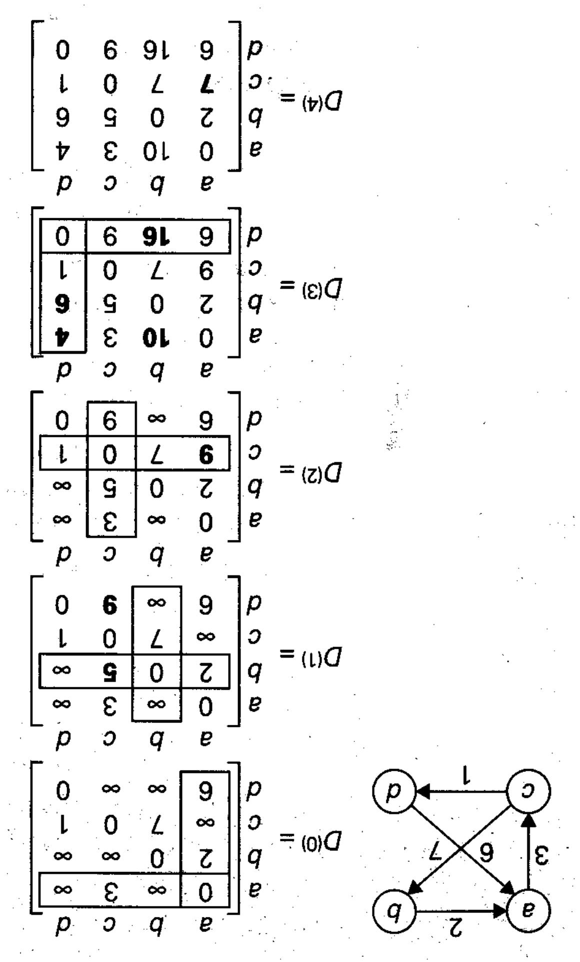
```
W := verkon G vierusmatriisi;
for i := 1 to n do
  W[i][i] := 0
end for;
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      W[i][j] := min(W[i][j], W[i][k] +
      W[k][j])
    end for
  end for
end for.
```

- Silloin muistiksi riittääkin vain 2-ulotteinen matriisi $W[i][j]$ eli

$$O(n^2) \qquad \text{muistipaikkaa}$$

  joka tarvittaisiin jo pelkille tuloksille

$$\delta(i, j) = W[i][j].$$



- Verkon vierus*matriisi*esitys kalvoilta 8.3 soveltuu hyvin juuri tälle algoritmille.

- Algoritmi voi havaita kelvottoman syötteen:

  Negatiivinen kehä yrittää päivittää diagonaalia $W[i][i]$ negatiiviseksi.

- Jos laskennan jälkeen pitää vastata kysymyksiin

  "Mikä on lyhyin polku solmusta i solmuun j?"

  niin matriisia $W[i][j]$ on täydennettävä lisätiedoilla.

- Eräs tapa on tallettaa samalla toiseen matriisiin $K[i][j] =$ muuttujan $k \geq 0$ arvo sillä hetkellä kun paikan $W[i][j]$ sisältö muuttuu.

- Silloin vastauspolku voidaan tulostaa *jäljittämällä rekursiivisesti arvon $W[i][j]$ perustelu.*

if $i = j$ **then**
  Tulosta solmu i
else
  perustelu(i, j)
end if.

procedure perustelu(i: **solmu**, j: **solmu**)
  if $K[i][j] = 0$ **then**
    Tulosta kaari $i \xrightarrow{W[i][j]} j$;
  else
    perustelu$(i, K[i][j])$;
    perustelu$(K[i][j], j)$
  end if.

9.1.3 Bellmanin ja Fordin algoritmi

- Seuraava kuva [5, kuva 6.21] osoittaa, ettei kalvojen 9.1.1 Dijkstran algoritmi toimi, jos kaarilla on *negatiivisia painoja:*

  (a) Algoritmi valitsee väärän polun $s \rightsquigarrow v$.

  (b) Jos negatiivinen paino -3 poistetaan lisäämällä $+3$ jokaiselle kaarelle, niin lyhyin polku $s \rightsquigarrow t$ vaihtuu.

- Silloin soveltuukin Bellmanin ja Fordin algoritmi [1, luku 24.1; 5, luku 6.8].

- Määritellään solmun v etäisyydeksi

$$\text{matka}[v] = -\infty$$

jos sinne on alkusolmusta s polku

$$s \to \bigcirc \to \bigcirc \to \cdots \to \underbrace{u \xrightarrow{w_1} \bigcirc \xrightarrow{w_2} \bigcirc \xrightarrow{w_3} \cdots \xrightarrow{w_k} u}_{\text{kehä}} \to \bigcirc \to \bigcirc \to \cdots \to v \quad (3)$$

jolla on *negatiivinen kehä* eli

$$w_1 + w_2 + w_3 + \cdots + w_k < 0.$$

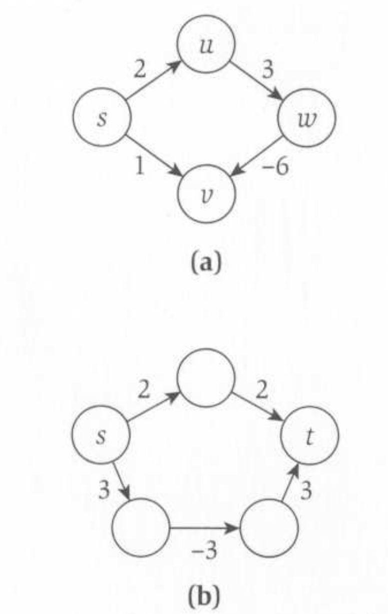


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest s-t path.

- Algoritmin perusajatus:
  - Jos etäisyys matka[v] on äärellinen, niin vastaava lyhyin polku on yksinkertainen.

    (Yksinkertaiset polut ja kehät määriteltiin kalvoilla 9.1.2.)
  - Yksinkertaisella polulla on

    $\leq |V| - 1$ kaarta.
  - Päivitetään siis etäisyydet matka[u] yhteensä

    $|V|$ kertaa
  - Jos viimeisellä päivityskerralla matka[u]...

    pysyy samana niin ollaan jo löydetty todellinen äärellinen etäisyys ja vastaava polku

    $$u \leftarrow p[u] \leftarrow p[p[u]] \leftarrow \cdots \leftarrow s.$$

    pienenee yhä niin u on negatiivisella kehällä (3).

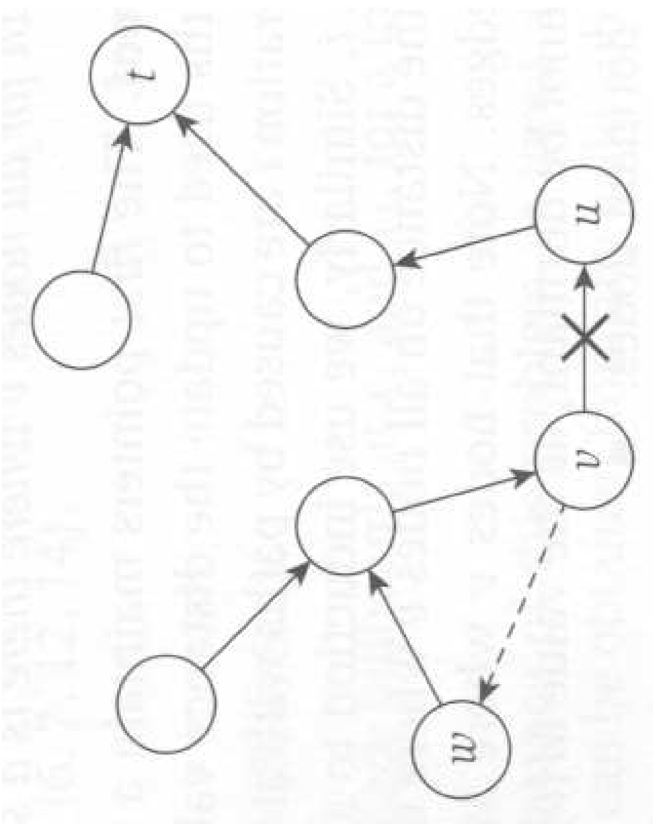
    Silloin pitääkin lopuksi korjata matka[v]**:=** $-\infty$ kaikille solmuille joihin on polku $u \rightsquigarrow v$.

Varsinainen algoritmi on:

1: **for** jokainen solmu $p \in V(G)$ **do**
2:    matka[p] **:=** $+\infty$**;**
3:    $\pi[p]$ **:=** NULL
4: **end for;**
5: matka[s] **:=** 0**;**
6: **for** $|V|\,(G) - 1$ kertaa **do**
7:    **for** jokainen kaari $p \xrightarrow{w} q \in E(G)$ **do**
8:       **if** matka[q] $>$ matka[p] $+ w$ **then**
9:          matka[q] **:=** matka[p] $+ w$**;**
10:          $\pi[q]$ **:=** u
11:       **end if**
12:    **end for**
13: **end for**

- Riveillä 8–11 käytetään samaa laskusääntöä kuin Dijkstran algoritmissa kalvoilla 9.1.1.
- Samoin pidetään yllä osoitinketjuja $\pi[q]$.
- Nyt ketju viekin takaisin alkusolmuun s vain kun matka[q] on äärellinen.
- Jos matka[q] $= -\infty$, niin se viekin johonkin negatiiviseen kehään (3).

  [5, kuva 6.26 (b)]



Jälkikäsittelyvaihe tekee vielä yhden silmukkakierroksen:

for jokainen kaari $p \xrightarrow{w} q \in E(G)$ **do**
   if matka[q] $>$ matka[p] $+ w$ **then**
      matka[q] **:=** matka[p] $+ w$**;**
      korjaa(q)
   end if
end for

Sitä ei tarvita, jos tiedetään etukäteen, ettei syöteverkossa G ole negatiivisia kehiä (3).

Vaiheen aikana löydetään negatiiviset kehät

$$q \rightsquigarrow q$$

ja päivitetään kaikki siitä saavutettavat solmut x.

Päivitys voidaan tehdä vaikkapa kalvojen 8.4 syvyyssuuntaisella läpikäynnillä:

korjaa(x)
   if matka[x] $\neq -\infty$ **then**
      matka[x] **:=** $-\infty$**;**
      for jokainen kaari $x \to y \in E(G)$ **do**
         korjaa(y)
      end for
   end if

- Aikaa kuluu

$$\underbrace{|V(G)|}_{\text{kierroksia}} \cdot \underbrace{|E(G)|}_{\text{kierros}}$$

askelta.

  – Siis sama kun Dijkstran algoritmissa kalvoilla 9.1.1, jos ei käytä kekoa.

  – Dijkstra onkin nopeampi juuri siksi, että se voi keolla valita nopeasti sellaisen solmun, jonka kenttä matka saa nyt lopullisen arvonsa.

  – Bellman-Ford joutuu sen sijaan päivittämään kaikkia solmuja.

- Seuraavassa kuvassa [1, kuva 24.4] on esimerkki, jossa

  – kenttien matka arvot on merkitty solmuihin

  – π-kaaret on tummennettu

  – ei ole negatiivisia kehiä, joten jälkikäsittelyvaihetta ei tarvita.

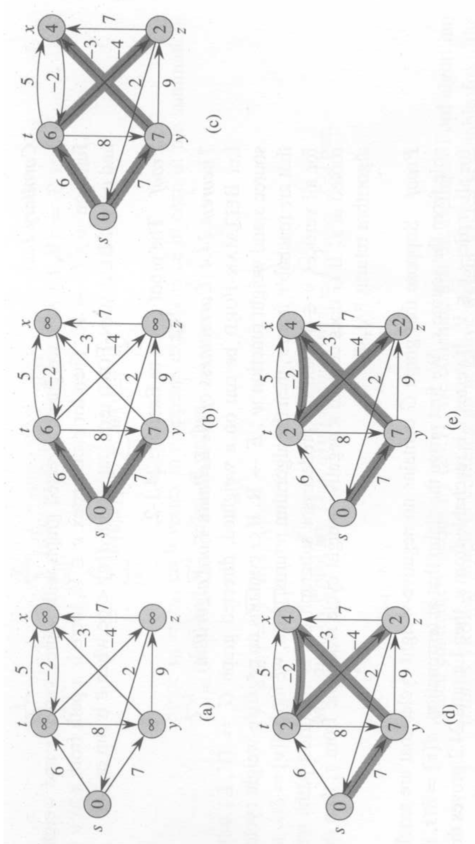


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

9.2 Transitiivinen sulkeuma

[1, sivut 632–634; 6, sivut 280–284 ja kuva 8.2]

- Suunnatun painottamattoman verkon G *transitiivinen sulkeuma* G' (transitive closure) *oikoo polut:*

  Jos verkossa G on epäsuora polku

$$u \to \bigcirc \to \bigcirc \to \bigcirc \to \cdots \to \bigcirc \to v$$

  niin verkossa G' on suora kaari

$$u \to v.$$

- *Warshall*in algoritmi saadaan kalvojen 9.1.2 Floydin algoritmista:

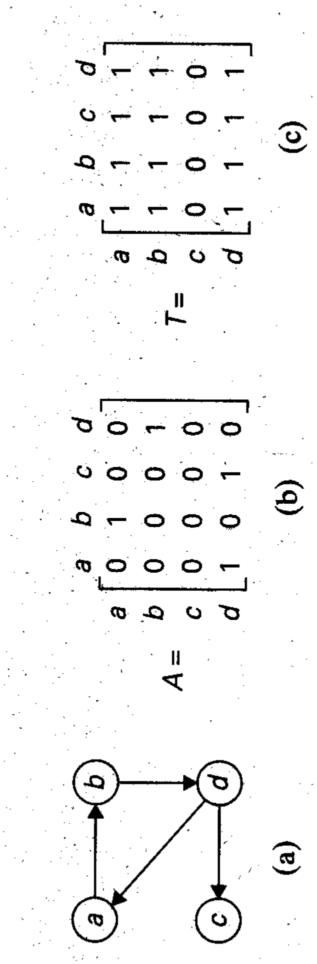
  – Lasketaankin lukujen sijaan *totuusarvoilla*.

  – $W'[i][j] =$ "olisiko $W[i][j] < +\infty$?".

  – 'min' on 'tai'.

  – '+' on 'ja'.

W' **:=** verkon G vierusmatriisi**;**
for i **:=** 1 **to** n **do**
  $W'[i][i]$ **:=** TRUE**;**
end for;
for k **:=** 1 **to** n **do**
  for i **:=** 1 **to** n **do**
    for j **:=** 1 **to** n **do**
      $W'[i][j]$ **:=** $W'[i][j]$ **or (**$W'[i][k]$ **and** $W'[k][j]$**)**
    end for
  end for
end for.



9.2.1 Esimerkki: yhtiökokoukset

- Sanotaan, että yhtiö A *hallitsee* yhtiötä B, jos

  joko yhtiö A omistaa itse $> 50\%$ yhtiön B osakkeista

  tai yhtiö A hallitsee muita yhtiöitä $C_1, C_2, C_3, \ldots, C_m$, jotka yhdessä yhtiön A itsensä kanssa omistavat yhteensä $> 50\%$ yhtiön B osakkeista.

  Silloinhan yhtiö A voi äänestää yhtiön B yhtiökokouksessa läpi minkä tahansa haluamansa päätöksen.

Syötteenä annetaan verotiedot

"yhtiö A omistaa $p\%$ yhtiön B osakkeista".

Tuloksena halutaan päätellä niistä tiedot

"yhtiö A hallitsee yhtiötä B".

Syötteenä saadaan matriisi osuus$[A, B] =$ yhtiön A omistusosuus yhtiön B osakkeista.

Tuloksena tuotetaan matriisi hallitsee$[A, B] =$ hallitseeko yhtiö A yhtiötä B vaiko ei.

Alustuksena on hallitsee$[i, j] = \mathbf{false}$ kaikilla indeksipareilla i, j.

Lasketaan säännön

$$\underbrace{\text{osuus}[i,k] + \sum_{\text{hallitsee}[i,j]} \text{osuus}[j,k]}_{\text{valta}[i,k]=\text{yhtiön } i \text{ äänivalta yhtiössä } k} > 50\%$$

transitiivista sulkeumaa.

Aputietorakenteena pidetään yllä em. matriisia valta$[A, B]$

- kunnes hallitsee$[A, B]$
- alustuksena valta $=$ osuus.

- Laskenta on luontevaa tehdä *inkrementaalisesti:* lisäämällä

  – transitiivisesti suljettuun verkkoon

  – annettu uusi kaari $i \to j$

  – sekä kaikki lisäkaaret, jotta verkosta saadaan taas transitiivisesti suljettu.

  Intuitio: pidetään se yhtiökokous, jossa yhtiö i ottaa hallintaansa yhtiön j.

- Yhtä nopeaa kuin kalvojen 9.2 algoritmi:

  – Jokainen N^2 matriisipaikasta hallitsee$[i, j]$ vaihtaa arvoaan korkeintaan kerran missä $N =$ yhtiöiden lukumäärä.

  – Arvon vaihtavassa rekursiokutsussa suoritetaan $O(N)$ operaatiota $+$ syntyneet rekursiokutsut.

  – Rekursiokutsut, joissa arvo on jo vaihtunut, ovat vakioaikaisia.

  – Siis kaikissa rekursiokutsuissa tehdään yhteensä $O(N^3)$ operaatiota.

```
Alusta globaalit matriisit hallitsee ja valta
syötteestä osuus;
for jokainen syöte jolla osuus[i,j] > 50% do
   valloita(i,j)
end for.

procedure valloita(i,j)
  if not hallitsee[i,j] then
     hallitsee[i,j] := true;
     for jokainen yritys k do
        if hallitsee[j,k] then
           valloita(i,k)
        else
           valta[i,k] := valta[i,k] + osuus[j,k];
           if valta[i,k] > 50% then
              valloita(i,k)
           end if
        end if
     end for;
     for jokainen yritys h joka hallitsee[h,i] do
        valloita(h,j)
     end for
  end if.
```


9.3 Virittävistä puista

[1, luku 23 ja kuva 23.5; 6, luvut 9.1–9.2]

- Suuntaamaton verkko G on *yhtenäinen* jos jokaisesta solmusta pääsee jokaiseen muuhun solmuun.

  (Vastaavaa käsitettä suunnatuille verkoille käsiteltiin kalvoilla 8.4.2.)

- Sen *virittävä puu* (spanning tree, ST) on pienin kaarijoukko $T \subseteq E(G)$ joka säilyttää yhtenäisyyden.

- Jos G on painotettu, niin valitaan vielä sellainen T, johon kuuluvien kaaripainojen summa

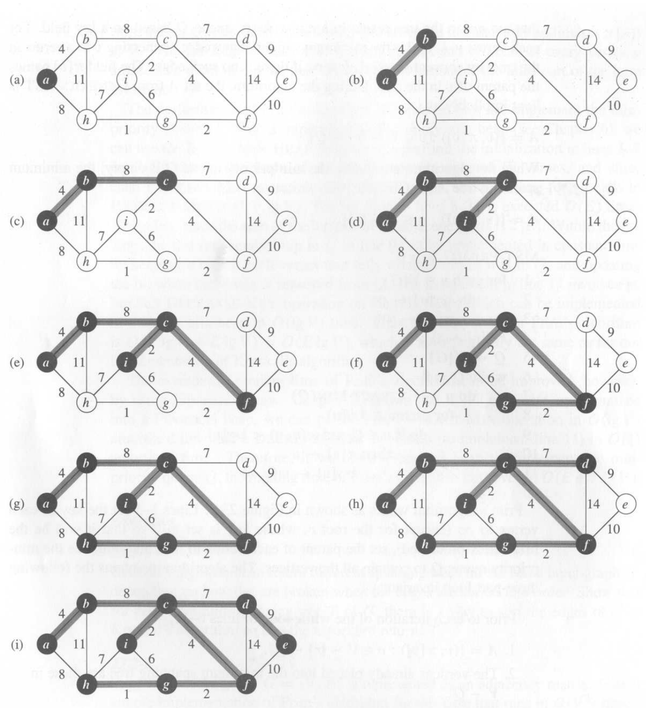
$$\sum_{u \xrightarrow{w} v \in T} w$$

  on *pienin* mahdollinen (minimal ST, MST).

- Suunnatuilla painotetuilla verkoilla vastaava käsite olisi *arborescence* ja algoritmi mutkikkaampi [5, luku 4.9].

- Esimerkki:
  - Suomen rautatieverkko on yhtenäinen:

    Jokaiselta asemalta pääsee jotakin kautta jokaiselle muulle asemalle.
  - Jokaisella rataosuudella on ylläpitokustannus.
  - Mitkä rataosuudet VR voi lakkauttaa, ja silti pitää rataverkkonsa yhtenäisenä?
  - Mitkä rataosuudet kannattaa lakkauttaa, jotta säästöt olisivat suurimmat?
- Eräs tapa laskea pienin virittävä puu:
  - Aloitetaan puun rakentaminen mielivaltaisesta lähtösolmusta s.
  - Liitetään puuhun aina *lyhyin* kaari
    - ∗ puussa jo olevasta solmusta
    - ∗ vielä puun ulkopuolella olevan solmuun.
  - Lopetetaan, kun kaikki solmut on saatu liitettyä puuhun.



- Tämä on *Prim*in algoritmi.
- Kalvojen 9.1.1 Dijkstran algoritmi, jossa
  - kuljetusta matkasta *unohdetaan kaikki muut kuin viimeinen kaari*
  - lisätään kaari vain nykyisen *puun ulkopuoliseen* solmuun v
  - 2-arvoisella kentällä tummuus[u] muistetaan, onko solmu u kasvavan puun T

    VALKEA: ulkopuolella

    MUSTA: sisäpuolella
  - ulkopuolella olevan solmun u kenttä matka[u] antaa lyhyimmän kaaren, jolla u pääsisi sisäpuolelle.
- Algoritmiin Dijkstra$_2$ tarvittavat muutokset on $\boxed{\text{laatikoitu}}$.
- Jos verkko G ei ollutkaan yhtenäinen, niin lähtösolmusta s saavuttamattomat solmut jäävät valkeiksi.

procedure Prim(G: **verkko**, s: **solmu**)
  for all $u \in V(G) \setminus \{s\}$ **do**
    $\boxed{\text{tummuus}[u] \text{ := VALKEA;}}$
    matka[u] := $+\infty$;
    $\pi[u]$:= NULL
  end for;
  $\boxed{\text{tummuus}[s] \text{ := VALKEA;}}$
  matka[s] := 0;
  $\pi[s]$:= NULL;
  Tee minimikeko kaikista solmuista $V(G)$ kentän matka suhteen;
  while keko ei ole tyhjä **do**
    Poista keosta sen pienin alkio u;
    $\boxed{\text{tummuus}[u] \text{ := MUSTA;}}$
    for all $u \xrightarrow{w} v \in E(G)$ **do**
      $\boxed{c \text{ := } w;}$
      if $\boxed{\text{tummuus}[v] \neq \text{MUSTA } \textbf{and}}$
      $c <$ matka[v] **then**
        matka[v] := c;
        $\pi[v]$:= u;
        siftup(kahva[v]) kalvoilta 2.3.3
      end if
    end for
  end while.

9.3.1 Esimerkki: ryvästys

- Syötteenä annetaan tason pisteet

$$\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \langle x_3, y_3\rangle, \ldots, \langle x_n, y_n\rangle.$$

- Ne pitää jakaa kahteen eri luokkaan A ja B siten, että nämä luokat ovat mahdollisimman kaukana toisistaan.
- Tarkemmin sanoen: siten, että lyhyinkin etäisyys luokan A pisteestä luokan B pisteeseen on mahdollisimman suuri.
- Tämä on eräs tapaus *ryvästyksestä* (clustering) [5, luku 4.7].

Algoritmi:

1. Muodosta suuntaamaton verkko G, jonka

   solmuina ovat annetut pisteet

   kaarina $p \overset{d_{pq}}{—} q$ ovat kaikki pisteparit $p \neq q$

   painoina d_{pq} ovat pisteiden p ja q välinen etäisyys.
2. Järjestä nämä kaaret painojensa mukaan kasvavaan järjestykseen.
3. Sijoita aluksi jokainen solmu yksinään omaan luokkaansa.
4. Toista seuraavaa, kunnes luokkia on jäljellä vain 2:

   (a) Ota järjestyksessä seuraava kaari $p \overset{d_{pq}}{—} q$.

   (b) Jos sen päätesolmut p ja q ovat vielä eri luokissa, niin yhdistä niiden luokat yhdeksi luokaksi.

Perustelu: Jokaisella silmukkakierroksella 4 huolehditaan, ettei ainakaan näiden pisteiden p ja q välinen etäisyys d_{pq} ole se pienin, joka erottaa luokat A ja B toisistaan.

Kruskalin MST-algoritmi [5, luku 4.5] saadaan seuraavin muutoksin:

- Askeleessa 1 käytetäänkin syötteenä annettua verkkoa G.
- Silmukkaa 4 toistetaankin, kunnes luokkia onkin jäljellä enää 1.

Toteutuksessa tarvitaan tietorakenne, joka

- sallii askeleen 4b tarvitsemat operaatiot
  - "Mikä on tämän solmun luokka?"
  - "Yhdistä näiden solmujen luokat!"
- toimii niin nopeasti, ettei silmukka 4 ole hitaampi kuin järjestämisaskel 2.

Sellainen on ns. *Union-Find* [1, luvut 21.2-21.3 ;5, luku 4.6] kalvoilla 9.3.2.

9.3.2 Union-Find-tietorakenne

- On pidettävä yllä *alkioiden ekvivalenssiluokkia*

  eli jokainen alkio kuuluu koko ajan tasan yhteen luokkaan

  seuraavien operaatioiden suhteen:

  Luo uusi alkio ja sille oma luokka.

  Etsi alkiota vastaava luokka.

  Yhdistä annettujen kahden alkion luokat yhdeksi yhteiseksi luokaksi.

- Rakennetta tarvitaan kalvojen 9.3.1 Kruskalin algoritmissa

  mutta se on hyödyllinen myös muualla.

- Perusidea: Jokaisella luokalla on *nimi*, joka on jokin sen alkioista.

- Liitetään siis jokaiseen alkioon a *nimikenttä* $a.\mathsf{name} =$ osoitin sen luokan nimialkioon, johon a kuuluu.

- Ylläpidetään näitä nimikenttiä siten, että

  joko alkio a on samalla itse oman luokkansa nimialkio

  $$a.\mathsf{name} = a \tag{4}$$

  joka on samalla nimikentän alkuarvo (mikä muukaan?)

  tai nimikenttäketju

  $$a.\mathsf{name}.\mathsf{name}.\mathsf{name}.\cdots.\mathsf{name} \tag{5}$$

  vie lopulta alkion a nimialkioon.

- Tämä ei vielä takaa tehokkuutta:

  Miten pidetään nimikenttäketjut (5) lyhyinä?

- Lisätään siis vielä *pituuskenttä* $a.\mathsf{rank} =$ pisimmän alkioon a tulevan nimikenttäketjun (5) pituus.

- Pituuskentän alkuarvo olkoon 0 (tai muu vakio).

- Kun alkio lakkaa olemasta nimialkio (4), niin sen pituuskenttääkään ei enää tarvita.

- Nyt voidaan määritellä tehokas luokkien yhdistämisoperaatio, joka saa syötteenään kahden eri luokan nimialkiot:

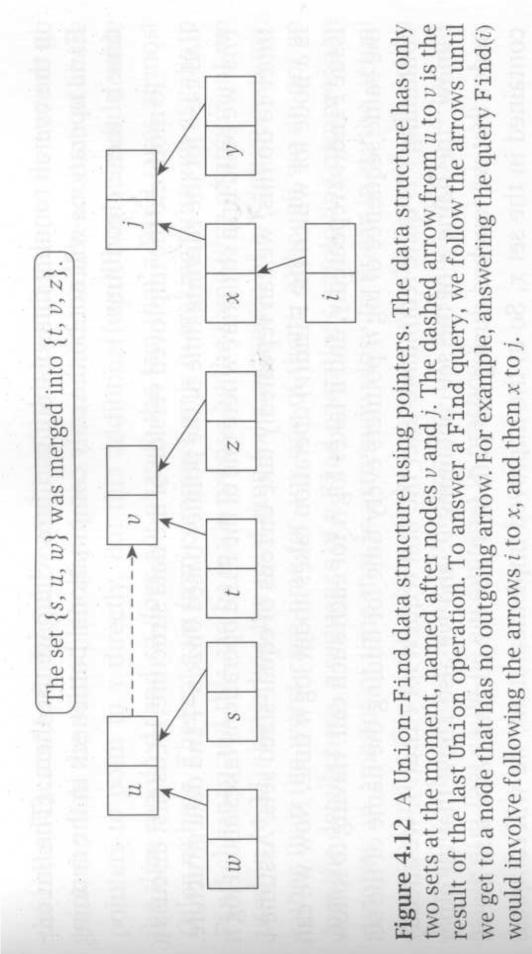
  procedure Union(a, b)

  ```
  if a.rank < b.rank then
    a.name := b
  else if a.rank > b.rank then
    b.name := a
  else
    b.name := a
    a.rank := a.rank+1
  end if.
  ```

  Toisin sanoen, kasvatetaan lyhyempiä nimikenttäketjuista (5).

  [5, kuva 4.12]

- Jo tämä takaa logaritmiset ketjut.

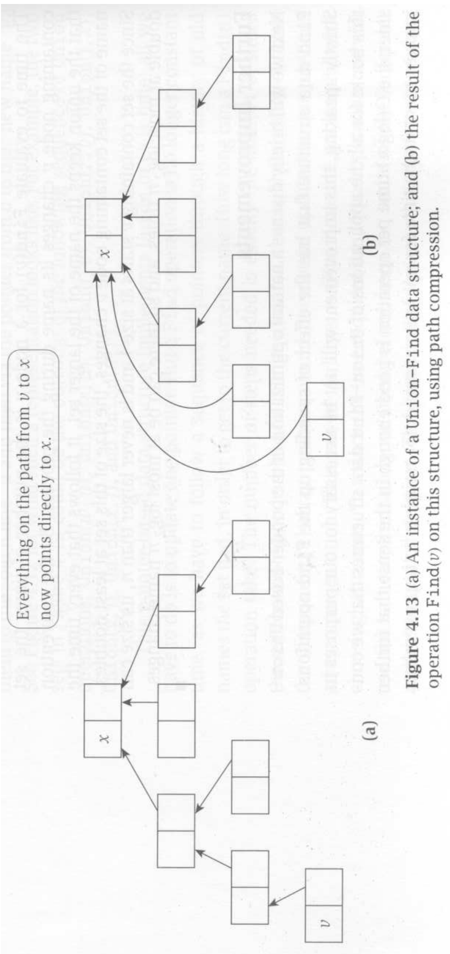


- Myös nimialkion hakuoperaatiota voidaan tehostaa *tiivistämällä polku* (path compression) jota kuljettiin:

  procedure Find(a)
    if a.name $= a$ **then**
      return a
    else
      a.name **:=** Find(a.name)**;**
      return a.name
    end if.

  [5, kuva 4.13]

- Tämä lisäparannus tekee Kruskalin algoritmin silmukan 4 operaatioista 4b yhdessä *käytännössä vakioaikaisia:*
  - Yksittäinen Find-kutsu voi viedä logaritmisen ajan
  - mutta sivutuotteenaan se tiivistää hyvin monen myöhemmän kutsun polkua
  - joten kuljetun nimikenttäketjun pituus onkin keskimäärin ≤ 4 [1, luku 21.4].



9.3.3 Prim vai Kruskal?

- Prim on käytännössä näistä kahdesta algoritmista nopeampi

  mutta ei niin paljon, että sillä olisi kisakäytössä merkitystä.

- Primin algoritmi on helppo muistaa Dijkstran muunnoksena.

- Primin algoritmin toteutus vaatii keon kalvoilta 2.3

  joko kahvoilla kalvoilta 2.3.3 — mutta niitä ei valmiskirjastoissa aina ole

  tai kahvoitta kuten kalvoilla 9.1.1.

- Kruskalin algoritmin toteutus taas vaatii

  sekä järjestämisalgoritmin — jonka saa valmiskirjastosta

  että Union-Find-tietorakenteen kalvoilta 9.3.2 — mutta sitä ei valmiskirjastoissa aina ole.

9.4 Parituksista ja virtauksista

Paritusongelmassa

- verkon solmut jakautuvat 2 eri luokkaan
- verkon kaaret kulkevat vain luokkien välillä
- kaarista pitää valita mahdollisimman

  vahvat kalvoilla 9.4.1

  monta kalvoilla 9.4.2

  kun solmu saa kuulua vain yhteen valittuun kaaren.

Maksimivirtausongelmassa kalvoilla 9.4.3

- verkon kaarilla on painot jotka kuvaavat siirtokapasiteetteja
- pitää laskea koko siirtoverkoston maksimikapasiteetti.

Kalvoilla 9.4.4 käsitellään paritusta virtauksena.

9.4.1 Vakaa paritus

Vakaiden parien (Stable Marriage) muodostamisongelmassa annetaan lähtötietoina seuraavaa:

- Miehet $1, 2, 3, \ldots, M$.
- Naiset $1, 2, 3, \ldots, N$.
- Jokaiselle miehelle i järjestetty lista niistä naisista $n_{i,1}, n_{i,2}, n_{i,3}, \ldots, n_{i,m_i}$ joita hän voisi kosia.
- Jokaiselle naiselle j järjestetty lista niistä miehistä $m_{j,1}, m_{j,2}, m_{j,3}, \ldots, m_{i,n_j}$ joille hän voisi myöntyä.
- Siis riittää tarkastella vain pareja jotka ovat toistensa listoilla.

Halutaan muodostaa sellaiset kihlaparit, että tulos on vakaa.

- Nainen/mies y *kiehtoo* miestä/naista x jos
  - x ei ole tällä hetkellä kihloissa kenenkään kanssa ja y on henkilön x listalla
  - x on tällä hetkellä kihloissa jonkun z kanssa mutta y on henkilön x listalla aikaisemmin kuin z.
- Pyydetään kihlaamaan annettuja naisia ja miehiä toisiinsa siten, ettei jäljelle jää enää yhtään miestä ja naista (kihloissa tai ei) jotka kiehtoisivat toisiaan — aiheuttaen epävakautta.
- Valitaan epäsymmetriset käyttäytymismallit:
  - Miehet kosivat, naiset eivät.
  - Naiset purkavat kihlauksiaan, miehet eivät.

  Silloin

  miehet saavat *parhaimman* mahdollisen puolison

  naiset tyytyvät *huonoimpaan* mahdolliseen.

Alussa kukaan ei ole vielä kihloissa;
while on mies i joka ei ole vielä kihloissa ja jonka listalla on vielä naisia **do**
  Otetaan miehen i listalta pois sen ensimmäinen nainen j;
  if mies i on naisen j listalla **then**
    if nainen i ei ole vielä kihloissa **then**
      Mies i ja nainen j kihlautuvat keskenään
    else if naisen j kihlattu k on naisen j listalla vasta miehen i jälkeen **then**
      Nainen j purkaa ensin kihlauksensa miehen k kanssa jonka jälkeen mies i ja nainen j kihlautuvat
    end if
  end if
end while.

9.4.2 Maksimaalinen paritus

Muutetaan kalvojen 9.4.1 ongelmaa:

- *Ei enää järjestystä* minua kiinnostavien partnereiden välillä — listasta joukoksi.
- Yritetäänkin tehdä *mahdollisimman monta* kihlaparia.
- Yritetään muodostaa *auttava polku* (augmenting path)

$$m_1, n_1, m_2, n_2, m_3, n_3, \ldots, m_k, n_k \qquad (6)$$

  missä

  1. ensimmäinen m_1 ja viimeinen n_k eivät ole vielä kihloissa kenenkään kanssa
  2. edellinen mies m_i ja seuraava nainen n_i kiinnostavat toisiaan
  3. mutta edellinen nainen n_i onkin jo kihloissa seuraavan miehen m_{i+1} kanssa.

- Käännetään auttava polku (6) *päin vastoin*:

  Korvataan vanhat kihlaukset 3 uusilla kihlauksilla 2.

- Kihlausten kokonaislukumäärä *kasvaa yhdellä*.

  Toistetaan siis

  1. auttavan polun muodostamista
  2. sen kääntämistä nurin

  kunnes auttavaa polkua ei enää ole.

- Lopputuloksessa on mahdollisimman monta kihlaparia:

  Jos olisi toinen tapa, jossa olisi vielä enemmän pareja, niin

  – katsotaan missä suhteissa tulos ja tapa eroavat toisistaan

  – nähdään eroista tulosta auttava polku.

- Menetelmä tuottaa *maksimaalisen parituksen 2-jakoisissa verkoissa* (Maximal Matching in Bipartite Graphs) [1, kuva 26.7]:

  – Solmut on jaettu

    * miesten luokkaan vasemmalla
    * naisten luokkaan oikealla.

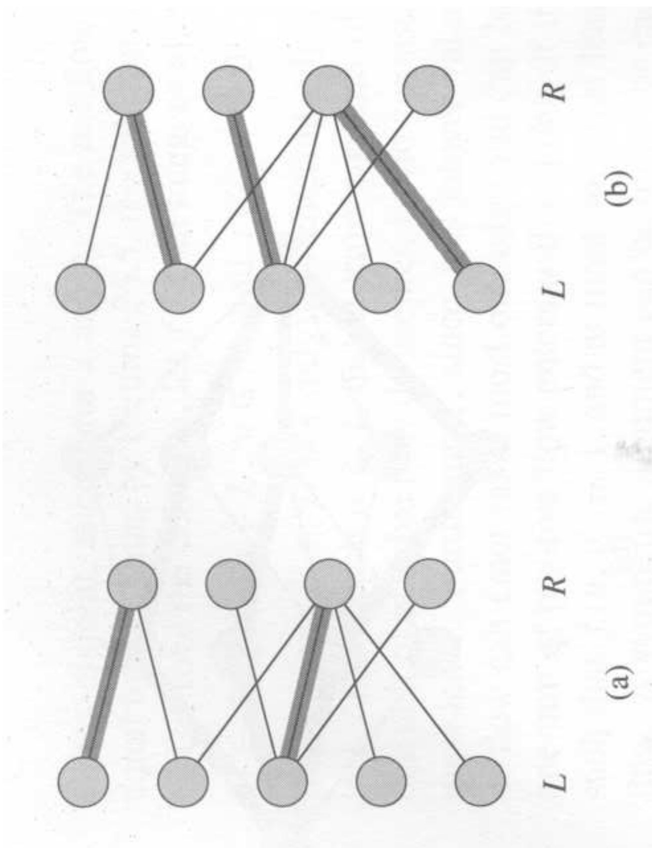
  – Suuntaamattomat kaaret kulkevat luokasta toiseen ja kertovat (molemminpuoliset) kiinnostukset.

  – Auttava polku (6) vuorottelee

    * vasemmalta oikealle vaaleaa
    * oikealta vasemmalle tummaa

    kaarta pitkin.

- Tämä ongelma löytyy (osana) monista sellaisista tehtävistä, joissa pyydetään sovittamaan yhteen kahdenlaisia asioita mahdollisimman kattavasti.



- Menetelmän optimoitu toteutus on *Hopcroftin ja Karpin* algoritmi [1, ongelma 26-7].

- Optimoinnissa

  lasketaan ensin mahdollisimman *monta eri henkilöistä koostuvaa polkua*

  käännetään kaikki nämä polut samalla kerralla.

- Silloin yksi toistokierros voi kasvattaa kihlaparien lukumäärää useammalla kuin yhdellä.

- Menetelmän kuvailussa käytetään apuverkkoja.

  Todellisessa toteutuksessa niitä ei tarvitse muodostaa erillisinä tietorakenteina.

1. Muodosta suunnattu apuverkko G_1 jossa

   solmuina ovat syötteen henkilöt

   miehestä lähtee kaari jokaiseen muuhun häntä kiinnostavaan naiseen kuin hänen omaan kihlattuunsa

   naisesta lähtee kaari vain hänen omaan kihlattuunsa.

   Apuverkon G_1

   - kehättömät polut
   - lähtien naimattomasta miehestä

     (eli oikean laidan solmusta johon ei tule kaaria)
   - päättyen naimattomaan naiseen

     (eli vasemman laidan solmuun josta ei lähde kaarta).

   ovat täsmälleen *kaikki* mahdolliset auttavat polut (6) määritelmän nojalla.

2. Suorita apuverkolle G_1 sellainen kalvojen 8.5 leveyssuuntainen läpikäynti

   - jonka työjono Q alustetaankin kaikilla vapailla miehillä
   - joka lopetetaan heti, kun työjonon Q alkuun ilmestyy vapaa nainen.

   Jos Q tyhjeni, niin koko algoritmin toistaminen voidaan lopettaa: enää ei synny uusia auttavia polkuja (6).

3. Muodosta apuverkko G_3 poistamalla apuverkosta G_1 kaikki ne kaaret $u \to v$ joilla

   $$\text{syvyys}[v] \neq \text{syvyys}[u] + 1.$$

   - Eli kaikki ne kaaret $u \to v$, joita pitkin askeleen 2 leveyssuuntainen läpikäynti *ei* olisi voinut edetä.
   - Leveyssuuntainen läpikäynti takaa, että apuverkon G_3 askeleen 1 mukaiset polut ovat täsmälleen *lyhyimmät* auttavat polut (6).

4. Suorita seuraava laskenta:

   Alusta tummuus[u] := VALKEA jokaiselle solmulle u;
   for jokainen työjonossa Q oleva vapaa nainen v **do**
     if uusipolku(v) **then**
       Rekursiokutsun muodostama polku

       $v \leftarrow \pi[v] \leftarrow \pi[\pi[v]] \leftarrow \cdots \leftarrow$ vapaa mies

       on eräs uusista henkilöistä koostuva auttava polku (6)
     end if
   end for.

   Rekursiivinen aliohjelma uusipolku(v)

   - yrittää muodostaa solmulle v sellaisen π-polun, joka voitaisiin saada apuverkon G_3 syvyyssuuntaisella läpikäynnillä kalvojen 8.4 mukaisesti
   - etenee siis apuverkon G_3 kaaria *takaperin*
   - jättää kokeillut solmut **musta**ksi, jotta niitä ei enää kokeiltaisi toiste
   - takaa siten, etteivät löydetyt polut jaa solmuja.

function uusipolku(q: **solmu**) : **boolean**
  if tummuus[q] = VALKEA **then**
    tummuus[q] **:=** MUSTA**;**
    if q on vapaa mies **then**
      return true
    else
      for $p \to q \in E(G_3)$ **do**
        if uusipolku(p) **then**
          $\pi[q]$ **:=** p**;**
          return true
        end if
      end for
    end if
  end if;
  return false.

9.4.3 Maksimivirtaus

Maksimivirtaus (eli -vuo) (Maximal Flow) [1, luku 26] mallintaa verkkona (esimerkiksi) vesijohtoputkistoa:

- Otetaan suunnattu verkko G, jonka kaarilla $u \xrightarrow{c} v \in E(G)$ on painot $c > 0$.
- "Putken yläpäästä u voi virrata alapäähän v korkeintaan c litraa sekunnissa."

  Verkossa G sallitaan myös kehät.
- Verkossa G on

  lähtösolmu (source) $s \in V(G)$ johon ei tule kaaria ja josta lähtee kaikki putkistossa virtaava vesi ("vesilaitos")

  maalisolmu (sink) $t \in V(G)$ josta ei lähde kaaria ja johon päätyy kaikki putkistossa virtaava vesi ("meri").
- Kaikissa muissa solmuissa $u \in V(G)$ tuleva ja lähtevä virtaus ovat aina yhtä suuret.

Virtaus (flow) $f\colon V(G)^2 \mapsto \mathbb{R}$ toteuttakoon:

Kapasiteettirajoitus (Capacity Constraint)
$f(u,v) \leq c(u,v)$ missä

$$c(u,v) = \begin{cases} c & \text{jos } u \xrightarrow{c} v \in E(G) \\ 0 & \text{muuten} \end{cases}$$

on suora kapasiteetti yläsolmusta u alasolmuun v.

Kallistussymmetria (Skew Symmetry)
$f(u,v) = -f(v,u)$ kaikilla $u,v \in V(G)$.

Negatiivinen virtaus on mitattu kaaren suuntaa *vastaan*.

Todellinen virtaus on positiivista.

Kumous (Cancellation): vain toisella kaarista $u \xrightarrow{c} v, v \xrightarrow{d} u \in E(G)$ voi olla todellista virtausta.

Virtauksen säilyminen (Flow Conservation)

$$\sum_{v \in V(G)} f(u,v) = 0$$

kaikissa välisolmuissa $u \in V(G) \setminus \{s,t\}$.

- Tehtävänä on löytää sellainen virtaus f jonka *arvo* (Value) eli putkistoon lähtevä vesimäärä

  $$|f| = \sum_{v \in V(G)} f(s,v)$$

  on mahdollisimman suuri.
- Esitetään *Fordin ja Fulkersonin* perusmenetelmä (the Ford-Fulkerson method) [1, luku 26.2 ja kuva 26.5].

  Se yleistää kalvojen 9.4.2 auttavan polun (6) ideaa:

  - Solmuilla ei enää ole sukupuolta.
  - Kihlausta vastaa virtauksen lisäys eli polku kulkee kaaren suuntaan.
  - Kihlauksen purkua vastaa virtauksen vähennys eli polku kulkee kaarta vastaan.

Aluksi virtausta ei ole, joten alusta $f[u,v], f[v,u] := 0$ kaikille $u \xrightarrow{c} v \in E(G)$;

repeat

Muodosta *jäännösverkko* (Residual Network) G_f joka kertoo, miten virtausta f voi muuttaa:

- Jos $f[u,v] < c$ jollakin $u \xrightarrow{c} v \in E(G)$ (eli tälle kaarelle voi lisätä virtausta) niin lisää myötäkaari $u \xrightarrow{c-f[u,v]} v$.
- Jos $f[u,v] > 0$ jollakin $u \xrightarrow{c} v \in E(G)$ (eli tältä kaarelta voi vähentää virtausta) niin lisää vastakaari $v \xrightarrow{f[u,v]} u$.
- Jos näin syntyisi kaksi vierekkäistä kaarta $u \xrightarrow{p} v$ ja $u \xrightarrow{q} v$, niin summaa ne yhdeksi kaareksi $u \xrightarrow{p+q} v$.

if verkossa G_f on yksinkertainen polku p solmusta s solmuun t **then**

Olkoon d polun p *jäännöskapasiteetti* (Residual Capacity) eli pienin kaaripaino;

for all kaari $u \to v$ polulla p **do**

if $f[v,u] > 0$ eli kaari osoittaa virtausta *vastaan* **then**

$f[v,u] := f[v,u] - d$ eli vähennä virtausta jäännöskapasiteetin verran;

$f[u,v] := -f[v,u]$

else

$f[u,v] := f[u,v] + d$;

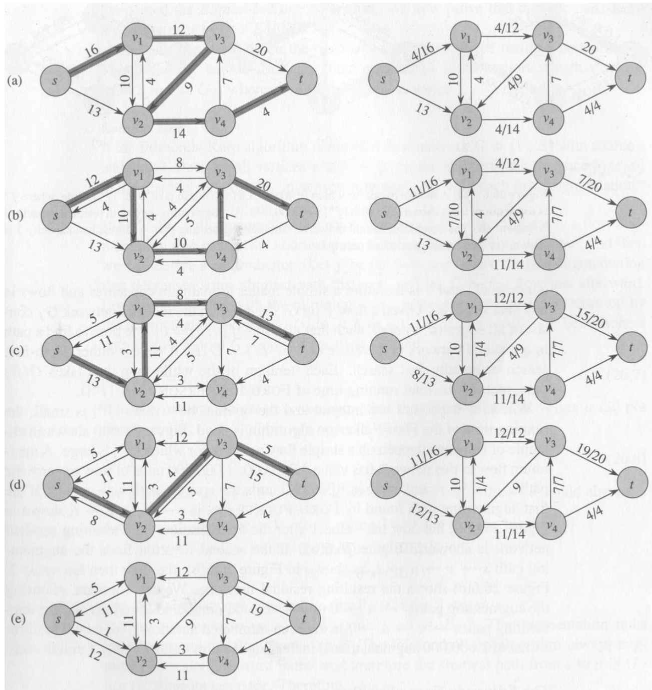
$f[v,u] := -f[u,v]$

end if

end for

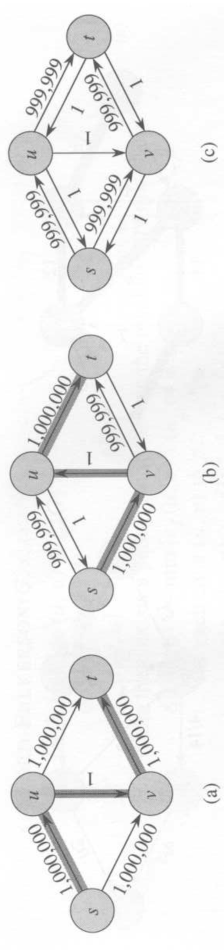
end if

until sellaista polkua p ei ole.



- Kokonaislukukapasiteeteilla (väli)tuloksetkin ovat kokonaislukuja.
- Kokonaisluvuilla algoritmi vie
$$O(|E(G)| \cdot |f^*|) \tag{7}$$
askelta, missä $|f^*|$ on tuo paras arvo.

  [1, kuva 26.6]
- *Edmondsin ja Karpin* algoritmi (the Edmonds-Karp algorithm)
  - valitsee *mahdollisimman vähäkaarisen* polun p
  - koska sellaisella lienee suuri jäännöskapasiteetti d.
- Tällainen p löytyy kalvojen 8.5 leveysläpikäynnillä:
  - Aloitetaan lähtösolmusta s.
  - Lopetetaan kun jonon Q alkuun ilmestyy maalisolmu t, ja luetaan sen π-puupolku (takaperin).



- Läpikäyntejä käytetään auttavien polkujen etsintään samaan tapaan kuin kalvojen 9.4.2 Hopcroftin ja Karpin algoritmin askeleissa 3 ja 4.

- Edmondsin ja Karpin algoritmi vie

$$O(|V(G)| \cdot |E(G)|^2) \qquad \text{askelta.} \qquad (8)$$

  – Riippuu vain verkon G koosta

  – eikä kapasiteeteista kuten (7).

- Polkua p voi etsiä myös kalvojen 8.4 *syvyys*läpikäynnillä:

  – helpompi ohjelmoida

  – usein riittävän nopea.

9.4.4 Esimerkki: Maksimaalinen paritus virtausongelmana

Moni ongelma voidaan palauttaa kalvojen 9.4.3 maksimivuon laskentaan.

Esimerkiksi kalvojen 9.4.2 ongelma voidaan esittää putkistona [1, kuva 26.8]:

- Lisätään *supermies* s ja putki siitä joka perusmieheen, ja *supernainen* t vastaavasti.

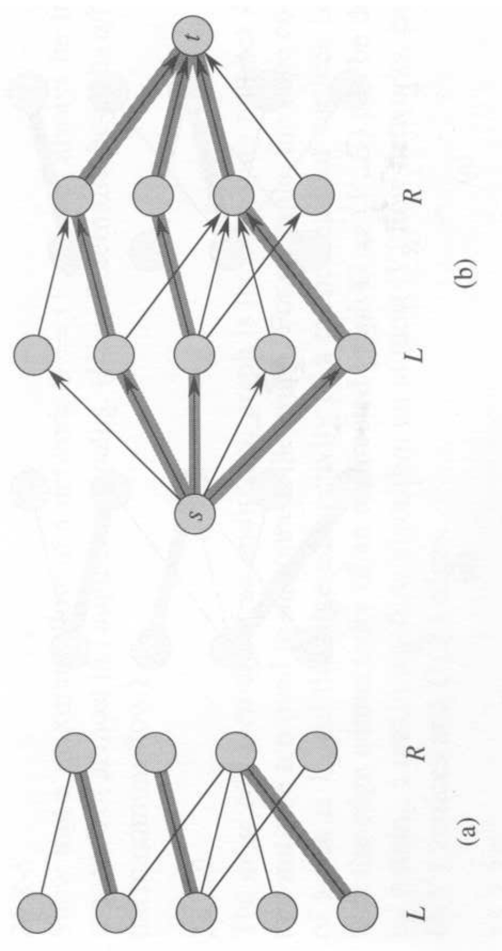
- Muut putket kulkevat miehestä häntä kiinnostaviin naisiin.

- Joka putkella on kapasiteettina 1.

- Vaaditaan vielä että virtaukset ovat kokonaislukuja.

  Fordin ja Fulkersonin menetelmä takaa tämän.

  Erityisesti sen erikoistapaus Edmondsin ja Karpin algoritmi takaa tämän.



- Nyt parituksen maksimikoko on maksimivuon arvo

$$|f^*| \leq \min(M, N)$$

  missä M on miesten ja N naisten lukumäärä.

- Palautuksella saatu algoritmi vie siis yhtälön (7) nojalla

$$O(L \cdot \min(M, N)) \qquad \text{askelta}$$

  missä L on syötteenä annettujen kiinnostusten lukumäärä.

- Kalvojen 9.4.2 erikoistunut Hopcroftin ja Karpin algoritmi veisi vain

$$O\left(L \cdot \sqrt{\min(M, N)}\right) \qquad \text{askelta.} \qquad (9)$$

- Jälkimmäinen algoritmi lieneekin keksitty edellistä optimoimalla juuri tähän tehtävään. . .

9.4.5 Minimileikkaus

- *Maksimivirtaus-minimileikkauslause* (Max-Flow Min-Cut Theorem) [1, Theorem 26.7]
  - perustelee miksi kalvojen 9.4.3 auttavat polut johtavat maksimivirtaukseen
  - osoittaa miten maksimivirtauksella voi löytää edullisimman tavan halkaista verkko kahteen palaseen.
- Verkon G (solmujen) leikkaus 2 palaseen S ja T on sellainen, että

  lähtösolmu kuuluu *alku*palaan, eli $s \in S$

  maalisolmu kuuluu *loppu*palaan, eli $t \in T = V(G) \setminus S$.
- Sen kapasiteetti on sen yli palasta S palaan T kulkevien kaarten painojen summa
$$c'(S,T) = \sum_{\substack{u \xrightarrow{w} v \in E(G) \\ u \in S, v \in T}} w.$$

- Virtauksen f aiheuttama *nettovirtaus* leikkauksen yli on se virtaus joka kulkee alkupalasta S loppupalaan T
$$n_f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v).$$

Apuväite: Kaikille verkon G

- virtauksille f
- leikkauksille S, T

pätee
$$|f| \leq c'(S,T)$$
eli virtaus ei voi ylittää leikkauksen kapasiteettia.

Perustelu: Poistetaan verkosta G kaaret alkupalasta S loppupalaan T. Silloin lähtösolmusta s ei enää pääse maalisolmuun t.

Jos siis halutaan mikä tahansa virtaus f lähtösolmusta $s \in S$ maalisolmuun $t \in T$, niin siihen täytyy käyttää ainakin osaa näistä poistetuista kaarista.

Pääväite: Seuraavat 3 ehtoa ovat yhtäpitävät:

1. f on maksimivirtaus verkossa G
2. sitä vastaavassa jäännösverkossa G_f ei ole auttavia polkuja
3. verkolla G on sellainen leikkaus, että
$$c'(S_f, T_f) = |f|.$$

Erityisesti: Maksimivirtauksen arvo = minimileikkauksen kapasiteetti.

Apuväitteen nojalla.

Perustelu $3 \Rightarrow 1$**:** Apuväitteen mukaan ei voi olla vielä suurempaa virtausta kuin $c'(S_f, T_f)$.

Perustelu $1 \Rightarrow 2$**:** Jos jäännösverkossa G_f on yhä auttava polku p, niin sitä voidaan käyttää yhä parantamaan virtausta f, joten f ei vielä ollutkaan maksimaalinen.

Perustelu $2 \Rightarrow 3$**:** Valitaan alkupalaksi S_f täsmälleen ne solmut, joihin pääsee lähtösolmusta s jäännösverkon G_f kaaria pitkin.

Silloin maalisolmu $t \in T_f = V(G) \setminus S_f$, kuten pitikin, koska jäännösverkossa G_f ei ole polkua solmusta s solmuun t.

Jos kaari $u \xrightarrow{w} v \in E(G)$ vie loppupalasta T_f alkupalaan S_f, niin valinnan nojalla sillä ei saa olla yhtään virtausta: $f(u,v) = 0$. Muutenhan solmusta $v \in S_f$ pääsisi jäännösverkon G_f vastakaarta pitkin solmuun $u \in T_f$.

Siis kaikki virtaus f kulkee alkupalasta S_f loppupalaan T_f eikä palaa, eli $|f| = n_f(S_f, T_f)$.

Jos kaari $u \xrightarrow{w} v \in E(G)$ vie alkupalasta S_f loppupalaan T_f, niin valinnan nojalla sen virtauksen pitää toimia täysillä, eli $f(u,v) = w$. Muutenhan solmusta $u \in S_f$ pääsisi jäännösverkon G_f myötäkaarta pitkin solmuun $v \in T_f$.

Siis $n_f(S_f, T_f) = c'(S_f, T_f)$ (eli leikkauskin toimii täysillä).

- Yhdistetään apu- ja päälause:

  Suurimman mahdollisen virtauksen arvo = pienimmän mahdollisen leikkauksen kapasiteetti.

- Perustelussa 2 ⇒ 3 tehtiin leikkaus
  - S_f = ne solmut, jotka viimeinen (epäonnistunut) auttavan polun etsintä tavoitti lähtiessään alkusolmusta s
  - T_f = ne solmut, jotka jäivät tavoittamatta

  joka on (eräs) pienin mahdollinen.

- Jos siis pienentää alkupalasta S_f loppupalaan T_f kulkevan kaaren kaaripainoa w jollakin arvolla δ, niin maksimivirtauksen f arvo
  - putoaa, koska leikkauksen S_f, T_f kapasiteetti on entistäkin pienempi
  - saman δ verran, koska f voidaan päivittää vastaavasti.

9.4.6 Esimerkki: Tieverkon katkaisu

Ongelma: Halutaan katkaista kaikki rautatieyhteydet kaupunkien s ja t välillä

- räjäyttämällä rataosuuksia poikki
- mahdollisimman vähillä räjäytyksillä.

(Vertaa kalvojen 9.3 vastakkaiseen ongelmaan.

Itse asiassa, virtauslaskenta keksittiin USAssa kylmän sodan vuosina juuri tähän tehtävään. . .)

Ratkaisu: Lasketaan maksimivirtaus.

1. Annetaan rataverkon jokaiselle rataosuudelle (kokonaisluku)kapasiteetti 1.
2. Muodostetaan kalvojen 9.4.5 minimileikkaus S_f, T_f.
3. Räjäytetään leikkauksen yli kulkevat rataosuudet.

Perustelu:

- Räjäytykset erottavat kaupungit s ja t toisistaan vain jos räjäytetään
  - ainakin kaikki ne rataosuudet eli kaaret, jotka kulkevat yli jostakin rataverkon leikkauksesta S, T
  - ja mahdollisesti niiden lisäksi jotakin turhaa.
- Koska kaikki kapasiteetit ovat 1, on kapasiteetti $c'(S, T)$ = leikkauksen S, T yli kulkevien kaarten lukumäärä.
- Kaikille muille leikkauksille S, T pätee

  $$c'(S, T) \geq c'(S_f, T_f)$$

  koska S_f, T_f on minimaalinen leikkaus.

Laajennus: Verkon G *kaariyhtenäisyyden aste* (edge connectivity) on pienin lukumäärä k kaaria, joiden poisto jakaa verkon G irrallisiin osiin.

[1, harjoitustehtävä 26.2-9; 8, luku 10.1.2]

- Erona edelliseen on se, että nyt *kysytäänkin sellaisia* s *ja* t joilla $c'(S_f, T_f)$ on pienin.
- Voidaan kiinnittää mielivaltainen s ja käydä läpi kaikki muut vaihtoehdot $t \in V(G) \setminus \{s\}$.
- Saadaan

  $$O(|V(G)|^2 \cdot |E(G)|^2)$$

  algoritmi.
- Kysymys "Onko $k = 1$?" ratkeaa nopeammin ja helpommin:
  - Kokeillaan jokaisella kaarella vuorollaan, olisiko G yhä yhtenäinen ilman sitä.
  - Kukin kokeilu voidaan tehdä kalvojen 8.4 syvyyssuuntaisella läpikäynnillä.

9.4.7 Vähimmäisvirtaukset

- Lisätään kalvojen 9.4.3 maksimivirtausongelmaan myös *alarajat:*

  Kaarelle $u \xrightarrow{l,c} v \in E(G)$ vaaditaan nyt virtaus $l \leq f(u,v) \leq c$.

  "Tässä putkessa pitää virrata *vähintään* l (ja korkeintaan c) litraa sekunnissa."

- Samat menetelmät soveltuvat *jos tunnetaan jokin alkuvirtaus* f_{alku} *joka täyttää nämä vaatimukset*.
  - Aletaankin parantamaan virtausta f_{alku}.
  - Ennen alarajoja aloitus $f_{\text{alku}} \equiv 0$ kelpasi.

- Alkuvirtauksen f_{alku} laskentaan palataan kalvoilla 9.4.10.

Ongelma: Kahden solmun välinen kehä

$$u \xrightarrow{l,c} v, v \xrightarrow{l',c'} u \in E(G)$$

ei kumoudu jos min$(l, l') > 0$!

Väärä ratkaisu: Entä jos toinen kahden solmun kehän kaarista saa tarpeeksi virtausta niin toisella sallittaisiinkin virtaus = 0?

- Silloin voitaisiin optimoida virtausta *suuntaamattomassa* verkossa G.
- Tämä taas on $\mathbf{NP}$-täydellistä [4, ongelma ND37], eli (luultavasti) vaikeaa (kokonaisluvuilla)!

Oikea ratkaisu: *Kielletään* tällaisten kehien esiintyminen syöteverkossa.

- Alustus on nyt $f[u,v]$:= $f_{\text{alku}}(u,v)$; $f[v,u]$:= $-f[u,v]$ kaikille $u \xrightarrow{l,c} v \in E(G)$.

- Muutetaan jäännösverkon G_f muodostamissääntöjä siten, että alarajan l alittaminen kielletään:
  - Jos $f[u,v] < c$ jollakin $u \xrightarrow{l,c} v \in E(G)$ niin lisää myötäkaari $u \xrightarrow{c-f[u,v]} v$.
  - Jos $f[u,v] > l$ jollakin $u \xrightarrow{l,c} v \in E(G)$ niin lisää vastakaari $v \xrightarrow{f[u,v]-l} u$.
  - Enää ei voi syntyä kahta kaarta $u \xrightarrow{c} v$ ja $u \xrightarrow{d} v$ jotka pitäisi yhdistää:
    - Toinen yhdistettävistä on aina myötä- ja toinen vastakaari.
    - Siis yhdistystarve syntyy vain kahden solmun kehistä.
    - Mutta sovimme, ettei niitä sallita.

- Sitten voidaan soveltaa Fordin ja Fulkersonin menetelmää. [3, luku I.§8]

9.4.8 Minimivirtaus

- Kalvoilla 9.4.7 lisättiin kalvojen 9.4.3 maksimivirtausongelmaan alarajat l. Silloin tulee järkeväksi kysyä kääntäen:

  Millainen *pienin* virtaus riittää alarajojen täyttämiseen (ylärajoissa pysyen)?

- Ratkeaa *pienentämällä mahdollisimman paljon* saatua alkuvirtausta f_{alku}.

  Virtausta saa pienentää vain poistamalla siitä sellainen osa, joka itsekin on virtaus.

  Poistettavan osan maksimointi on siis maksimivirtauksen laskentaa.

- Joten viritellään taas Fordin ja Fulkersonin menetelmää.

  [3, luku I.§8]

- Muutetaan jäännösverkon G_f muodostamissääntöjä antamaan se virtaus, joka voidaan poistaa. Nyt

  myötäkaarilla esitetään mahdollisuus poistaa

  vastakaarilla lisätä

  virtausta, eli päin vastoin kuin ennen.

  – Jos $f[u,v] < c$ jollakin $u \xrightarrow{l,c} v \in E(G)$ niin lisää *vasta*kaari $v \xrightarrow{c-f[u,v]} u$.

  – Jos $f[u,v] > l$ jollakin $u \xrightarrow{l,c} v \in E(G)$ niin lisää *myötä*kaari $u \xrightarrow{f[u,v]-l} v$.

- Silloin jäännöskapasiteetti d kertoo löydetyn luvallisen vähennyksen, joten virtauksen päivitys onkin vastakkaismerkkinen.

9.4.9 Esimerkki: Hiihtokeskus

- Tahkovuoren laskettelukeskuksessa kaikki laskettelurinteet lähtevät vuoren huipulta ja päättyvät Syvärinlahden jäälle.

- Rinteessä on risteyksiä, johon tultuaan laskettelija voi valita mitä reittiä lähtee seuraamaan. Samaan risteykseen voi myös tulla eri reittejä.

- Henkilökunnan pitää tarkastaa joka aamu kaikki rinteet laskemalla ne läpi ennen asiakkaiden tuloa.

- Koska hiihtohissi on hidas, halutaan kuljettaa huipulle kerralla niin paljon henkilökuntaa, ettei toista kuljetusta enää tarvita.

- Toisaalta turhaa henkilökuntaa ei haluta palkata.

- Paljonko henkilökuntaa tarvitaan?

- Mallinnus virtausongelmana:

  Lähtösolmuna s on huippu.

  Maalisolmuna t on jää.

  Muina solmuina ovat risteykset.

  Kaarina ovat reitit edellisestä risteyksestä seuraavaan.
  Kaari on suunnattu ylemmästä risteyksestä alempaan.

  Alaraja kaarella on 1 — ainakin yhden henkilökunnasta on laskettava se.

  Yläraja kaarella on $+\infty$ — niin monta kuin tarpeen.

  Minimivirtaus kertoo henkilökunnan minimitarpeen.

- Vaiheet:

  1. Ensin palkataan henkilökuntaa niin paljon, että homma varmasti hoituu.
  2. Sitten annetaan potkut liikaväelle.

- Vaihe 1 voidaan tehdä kalvojen 9.4.3 Fordin ja Fulkersonin menetelmän seuraavalla muunnoksella:

  Aluksi f_{alku} on 0-virtaus.

  Jäännösverkko koostuu kaarista $u \xrightarrow{1} v \in G_{f_{\text{alku}}}$ missä $u \xrightarrow{1,+\infty} v \in E(G)$.

  – Aina on tilaa vielä yhdelle.

  – Ylimääräiset karsitaan vasta vaiheessa 2.

  Polun p pitää sisältää jokin kaari

  $$u \xrightarrow{1} v \in G_{f_{\text{alku}}} \text{ jolla } f_{\text{alku}}[u,v] = 0.$$

  Eli valitaan rinne p jolla on tarkastamaton osuus, ja palkataan sille uusi työntekijä.

- Kalvoilla 9.4.10 esitetään toinen tapa, joka perustuu ongelman ja syöteverkon muuntamiseen sopivasti.

- Vaihe 2 voidaan tehdä kalvojen 9.4.8 minimivirtausmenetelmällä:

  Myötäkaaret ovat $u \xrightarrow{f[u,v]-1} v$ kaikilla niillä kaarilla $u \xrightarrow{1,+\infty} v$ joilla $f[u,v] > 1$.

  Vastakaaret $v \xrightarrow{+\infty - f[u,v]} u$ ovat aina mahdollisia.

  Saadaan virtaus f_{tarpeen} jolla

  – reittiä edellisestä risteyksestä u seuraavaan v laskee $f_{\text{tarpeen}}[u,v] \geq 1$ työntekijää

  – henkilökunnan kokonaistarve $\left|f_{\text{tarpeen}}\right|$ on mahdollisimman pieni.

9.4.10 Kierrot

- Joskus virtausongelmissa on *rajoitteita myös solmuissa*.

- Esimerkiksi

  tieverkko joka yhdistää

  tehtaita jotka valmistavat tuotetta

  kauppoihin jotka myyvät tuotetta

  annetulla tahdilla.

- Ei enää maksimoidakaan tuotteen kokonaisvirtaa

  – koska ei annettu maksimointisuuntaa lähtösolmusta s maalisolmuun t

  – vaan pelkästään varmistetaan, että tarjonta kattaa kysynnän.

- Tilannetta kutsutaan tuotteen *kierroksi* (circulation) [5, luku 7.7].

- Liitetään siis jokaiseen solmuun $v \in V(G)$ luku d_v, joka on sen *kysyntä* (demand):

  – Kun $d_v > 0$, niin kauppasolmu v *kuluttaa* tuotetta tahtiin d_v yksikköä.

  – Kun $d_v < 0$, niin tehdassolmu v *tuottaa* tuotetta tahtiin $-d_v$ yksikköä.

  – Kun $d_v = 0$, niin solmu v on samanlainen tuotevirtojen risteys ("tukkukaupan välivarasto") kuin alkuperäisessä virtausongelmassakin.

- Kysynnät d_v täyttävä kierto palautuu alkuperäiseen virtausongelmaan kalvojen 9.4.4 tekniikalla:

  Supertehtaasta s on kaari

  $$s \xrightarrow{-d_v} v$$

  jokaiseen tehdassolmuun v.

  Supermarkettiin t on kaari

  $$v \xrightarrow{d_v} t$$

  jokaisesta kauppasolmusta v.

  [5, kuva 7.13]

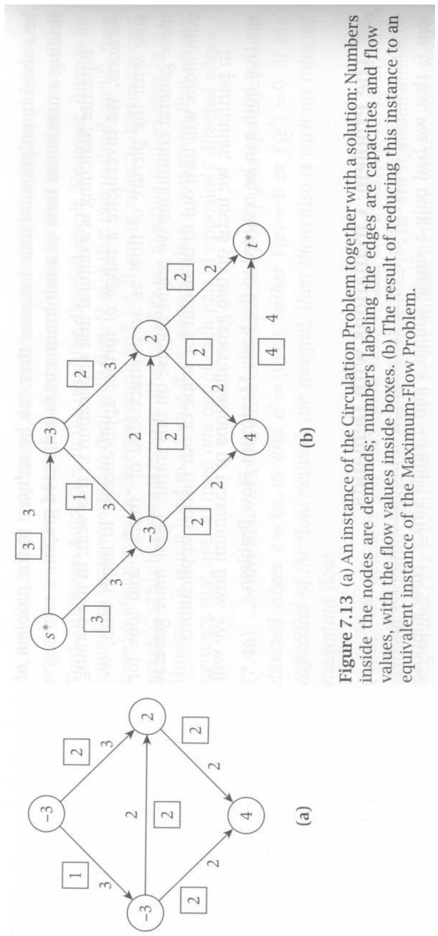


Figure 7.13 (a) An instance of the Circulation Problem together with a solution: Numbers inside the nodes are demands; numbers labeling the edges are capacities and flow values, with the flow values inside boxes. (b) The result of reducing this instance to an equivalent instance of the Maximum-Flow Problem.

- Kierroissa on helppo käsitellä kalvojen 9.4.7 verkkoja, joissa kaarilla
$$u \xrightarrow{l,c} v \tag{10}$$
on myös alaraja l:

1. Korvataan jokainen sellainen kaari tavallisella kaarella
$$u \xrightarrow{c-l} v.$$
2. Korvataan jokaisen solmun v kysyntä d_v kysynnällä $d_v - L_v$, missä
$$L_v = \underbrace{\left(\sum_{u \xrightarrow{l,c} v} l\right)}_{\text{sisääntulevat rajat}} - \underbrace{\left(\sum_{v \xrightarrow{l,c} u} l\right)}_{\text{ulostulevat rajat}}$$
on se lisätuotanto, joka tarvitaan kattamaan solmuun v liittyvien alarajojen l aiheuttama lisäkysyntä.

   [5, kuva 7.15.]
3. Olkoon f' tämän tavallisen ongelman ratkaisu.
4. Kun kullekin kaarelle (10) laitetaan takaisin virtaus $f'[u,v] + l$, niin saadaan *jokin* alarajat täyttävä virtaus f_{alku}.

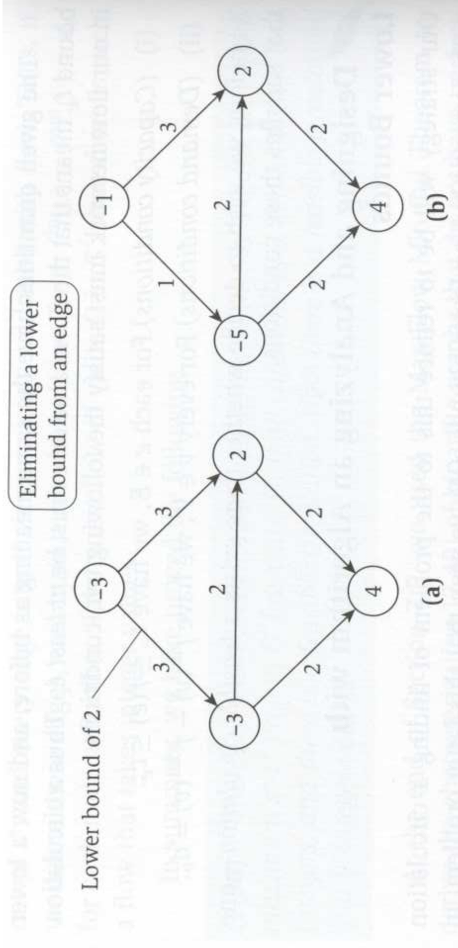


Figure 7.15 (a) An instance of the Circulation Problem with lower bounds: Numbers inside the nodes are demands, and numbers labeling the edges are capacities. We also assign a lower bound of 2 to one of the edges. (b) The result of reducing this instance to an equivalent instance of the Circulation Problem without lower bounds.

- Jos halutaankin vielä *minimaalinen* alarajat täyttävä virtaus, niin minimoidaan näin saatu f_{alku} kuten kalvoilla 9.4.8.

- Luonnollinen laajennus olisi, että eri tehtaat tuottaisivatkin eri tuotteita.

  Se kuitenkin tulee **NP**-täydelliseksi, eli (luultavasti) vaikeaksi (kokonaisluvuilla) [4, ongelma ND38]!

10 Dynaamisesta ohjelmoinnista

[8, luku 11; 1, luku 15]

Tarkastellaan seuraavia yleisiä algoritminsuunnittelumenetelmiä:

hajoita ja hallitse -periaatetta kalvoilla 10.1

Muunnelmia:

- *vähennä* (decrease) ja hallitse [6, luku 5]
- *muunna* (transform) ja hallitse [6, luku 6].

dynaamista ohjelmointia kalvoilla 10.2

ahneita algoritmeja kalvoilla 10.3.

10.1 Hajoita ja hallitse

- *Hajoita ja hallitse* (Divide and Conquer) [1, luku 2.3.1; 6, luku 4] on hyvin yleinen ja usein menestyksekäs algoritminkehitysperiaate:

  1. **Pura** syötteenä saatu suuri ongelman tapaus S aidosti pienempiin saman ongelman *erillisiin* tapauksiin $S_1, \ldots, S_m$.
  2. **Ratkaise** jokainen pienempi osa S_i *rekursiivisesti erikseen*, jotta saat jokaiselle osaratkaisun R_i.

     Kun S_i on tarpeeksi pieni, R_i selviääkin suoraan ilman rekursiota.
  3. **Yhdistä** lopuksi nämä saadut osaratkaisut $R_1, \ldots, R_m$ *yhdeksi kokonais*ratkaisuksi R alkuperäiselle S.

- Aikaisempia esimerkkejä:

  pikajärjestäminen kalvoilta 3.2

  potenssiinkorotus kalvoilta 4.8.

10.1.1 Binäärihaku

- *Binäärihaku* (Binary Search) [1, tehtävä 2.3-5; 6, luku 4.3] vastaa nopeasti seuraavaan kysymykseen:

  "Tässä on järjestetty taulukko A, eli aina $A[i] \leq A[i+1]$, ja etsittävä alkio b. Esiintyykö alkio b taulukossa A?"

- Jaetaan taulukko A
  - sisäalueeseen $S_1 = A[p \ldots r]$ jossa b *voi* olla (eli jota on käsiteltävä edelleen)
  - alueen S_1 ulkopuoliseen alueeseen S_2 jossa b *ei voi* olla (eli jota ei tarvitse enää käsitellä).

- Tässä ongelmassa vastaus saadaan suoraan alueelta S_1 ilman yhdistämistä.

  Binäärihaku onkin *puolita* ja hallitse (Decrease-by-half) -algoritmi [6, luku 5.5].

- Koko ongelmassa

  $p =$ taulukon A ensimmäinen indeksi
  $r =$ taulukon A viimeinen indeksi.

- Rajatapauksena S_1 on tyhjä eli $p > r$: vastataan "ei".

- Muuten S_1 on epätyhjä eli $p \leq r$: valitaan jokin $p \leq q \leq r$.
  - Jos $A[q] = b$, niin vastataan "kyllä".
  - Jos $A[q] < b$, niin uusi $S_1 = A[q+1 \ldots r]$.
  - Jos $A[q] > b$, niin uusi $S_1 = A[p \ldots q-1]$.

- Kannattaa valita r keskeltä aluetta S_1: uusi S_1 on varmasti alle puolet vanhasta.

- Saadaan

  $O(\log_2(\text{syötetaulukon } A \text{ pituus}))$

  askeleessa, eli *hyvin* nopeasti, toimiva algoritmi. . .

- jota on hankala saada toimivaksi kiireessä!


```
function BinarySearch(p, r):boolean is
  if p > r then
    return false
  else
    q := ⌊(p+r)/2⌋;
    if A[q] = b then
      return true
    else if A[q] < b then
      return BinarySearch(q + 1, r)
    else
      return BinarySearch(p, q − 1)
    end if
  end if.
```

Tai iteratiivisesti:

```
  while p ≤ r do
    q := ⌊(p+r)/2⌋;
    if A[q] = b then
      return true
    else if A[q] < b then
      p := q + 1
    else
      r := q − 1
    end if
  end while;
  return false
```

10.1.2 Pikavalinta

- Olkoon syötteinä
  - järjestämätön taulukko $A[1 \ldots n]$
  - järjestysnumero $1 \leq i \leq n$

  ja pitää löytää taulukon A i*. pienin alkio*.

- Tapaus $i = 1$ on kaikkein pienimmän alkion etsintä, joka sujuu helposti
  $$O(n)$$
  askeleessa käymällä A läpi.

- Ongelmalle saadaan algoritmi kalvojen 3.2 pikajärjestämisestä [1, luku 9.2]:
  - Se jakoi aineiston valitsemansa jakoalkion eri puolille kalvoilla 3.2.1.
  - Riittää jatkaa vain sillä puolella, jolla i. luku sijaitsee, kuten kalvojen 10.1.1 binäärihaussa.

- Saadaan funktio joka etsii alueen $A[p \ldots r]$ i. pienimmän luvun:

```
function QuickSelect(p, r, i) is
  if p = r then
    return A[p]
  else
    q := partition(p, r);
    k := q − p + 1;
    if i = k then
      return A[q]
    else if i < k then
      return QuickSelect(p, q − 1, i)
    else
      return QuickSelect(q + 1, r, i − k)
    end if
  end if.
```

  Ensimmäinen kutsu on QuickSelect$(1, n, i)$.

- Toimii *oletusarvoisesti* $O(n)$ askeleessa. . .
- mutta toistuvat kehnot partitioinnit voivat johtaa $O(n^2)$ askeleeseen, tosin harvoin.
- Mutkikkaampi partitiointi antaa *varmasti* $O(n)$ askeleen algoritmin [1, luku 9.3].

10.2 Optimointiongelman hajoittaminen ja hallinta

- Olkoon tehtävä *optimointi*ongelma:
  - jonkin suureen saaminen ”*oikeaksi*”, eli tavallisesti mahdollisimman
    - suureksi (maksimointitehtävä)
    - pieneksi (minimointitehtävä)

    sopivin ”*säädöin*”
  - vastaavien säätöjen ilmoittaminen.
- Esimerkki:
  - Autokorjaamolla on työruuhka.
  - Jokainen työ vie oman aikansa.
  - Jokaisella asiakkaalla on korjaamon laskuun vuokra-auto kunnes asiakkaan oma auto on valmis.
  - Missä järjestyksessä ruuhkaa kannattaa ryhtyä purkamaan?

- Yleistetään kalvojen 10.1 hajoita ja hallitse -periaate tällaisiin ongelmiin.
- Keskitytään kustannusten minimointiin.
- On *erilaisia tapoja* purkaa S osaongelmiin:

  1. tapa
     - antaa osaongelmat $S_1^1, \ldots, S_{m_1}^1$
     - joille saadaan osaratkaisut $R_1^1, \ldots, R_{m_1}^1$
     - joiden yhdistäminen ratkaisuksi R^1 maksaa kustannuksen c_1.
  2. tapa
     - antaa osaongelmat $S_1^2, \ldots, S_{m_2}^2$
     - joille saadaan osaratkaisut $R_1^2, \ldots, R_{m_2}^2$
     - joiden yhdistäminen ratkaisuksi R^2 maksaa kustannuksen c_2.
  3. tapa. . .

- Oletetaan lisäksi

  optimaaliset alirakenteet: Jokainen osaratkaisu R_i^t on vuorostaan vastaavan osaongelman S_i^t *optimaalinen* ratkaisu.

  – Jonkin S_i^t ratkaiseminen huonosti ei saa parantaa etsittyä kokonaisratkaisua.

  – "Jokainen S_i^t pitää voida ratkaista tällä samalla minimointialiohjelmalla."

  itsenäiset osaongelmat: Osaongelmat voidaan ratkaista toisistaan riippumatta.

  – Yhden osaratkaisun R_i^t valinta yhdelle osaongelmalle S_i^t ei estä minkään toisen osaratkaisun R_j^t valintaa toiselle osaongelmalle S_j^t.

  – "Minimointialiohjelman kutsut pitää voida suorittaa missä järjestyksessä tahansa."

- Näillä oletuksilla voidaan kirjoittaa minimointialiohjelma:

  function minimoi(S: **ongelma**): **hinta**
    if S on helppo **then**
      return sen suoran ratkaisun kustannus
    else
      for jokaiselle tavalle i **do**
        Jaa S osaongelmiin $S_1^i, \ldots, S_{m_i}^i$ tavalla i;
        for j := 1 **to** m_i **do**
          Laske osaongelman S_j^i osaratkaisun R_j^i kustannus
          d_j := minimoi(S_j^i)
        end for;
        Laske tavan i kustannus
        $d_i := \underbrace{c_i + d_1 + \cdots + d_{m_i}}_{(\star)}$
      end for;
      return pienin näin lasketuista kustannuksista d_i
    end if.

- Minimoitava kustannuslauseke ($\star$) voi olla osahintojen d_j^i mutkikkaampikin funktio.

- Jos tehtävästä löytyy myös

  päällekkäiset osaongelmat: Kun koko ongelmaa ratkotaan, niin sama osaosaosa...osaongelma tulee vastaan monta kertaa.

  – Tehtävässä on

    vähän erilaisia osaongelmia

    paljon erilaisia tapoja päätyä niihin.

  – "Minimointialiohjelma kutsuu itseään monta kertaa *samalla* parametrin S arvolla."

  niin silloin minimoijaa voi *tehostaa taulukoimalla* välituloksia kalvojen 6 tapaan.

- Aletaan luoda taulukkoa laskettu[S] =

  tyhjä jos osaongelmaa S ei vielä ole ratkaistu

  saatu ratkaisu jos kutsu minimoi(S) on jo tehty

  ja katsotaan aina ensin tästä taulukosta.

function minimoi′(S: **ongelma**): **hinta**
  if paikka laskettu[S] on vielä tyhjä **then**
    if S on helppo **then**
      e := sen suoran ratkaisun kustannus
    else
      for jokaiselle tavalle i **do**
        for j := 1 **to** m_i **do**
          d_j := minimoi′(S_j^i)
        end for;
        $d_i := c_i + d_1 + \cdots + d_{m_i}$
      end for;
      e := pienin näin lasketuista kustannuksista d_i
    end if;
    Täytä paikka laskettu[S] := e
  end if;
  return paikan laskettu[S] sisältö.

- Tämä on *dynaamisen ohjelmoinnin* (Dynamic Programming) ydinajatus.

  [1, luku 15; 6, luku 8]

- Tämä "ohjelmointi"
  - on peräisin *operaatiotutkimuksesta* (Operations Research)
  - tarkoittaa jonkin laitteen, prosessin,... säätimien asettamista siten, että sen tuottama tulos on haluttu.

  (Samaa sukua on *lineaarinen* (linear) ohjelmointi [1, luku 29; 6, sivut 236–238] jossa tulos ja säätimet ovat "suoraviivaisia".)

- Funktio minimoi′ etenee *ylhäältä alas muistifunktiolla* laskettu[S] (top-down memoization) [1, sivut 347–349; 6, sivut 297–299].

- Klassinen tapa edetä on *alhaalta ylös* (bottom up):
  - Rekursio täyttää taulukon laskettu[S] *pienemmistä* S *suurempiin* päin.
  - Siis rekursio voidaan korvata *iteraatiolla osaongelman* S *koon suhteen*.

10.2.1 Matriisitulo

- Jos syötematriisissa
  - A on p riviä ja q saraketta
  - B on q riviä ja r saraketta

  niin niiden tulo $A \cdot B$ on matriisi, jossa
  - on p riviä ja r saraketta
  - yhden paikan sisältö voidaan laskea q askeleella
  - koko sisältö voidaan siis laskea

    $$p \cdot q \cdot r$$

    askeleella.

- Matrisiitulo on *liitännäinen:*

  $$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

  Saamme siis valita itse kumman matriisituloista haluamme laskea ensin.

- Laskujärjestyksen valinta vaikuttaa askelmäärään:

matriisi	ri	sa	askeleet
A	1	2	0 (syöte)
B	2	3	0
C	3	4	0
$A \cdot B$	1	3	$0 + 0 + 1 \cdot 2 \cdot 3 = 6$
$B \cdot C$	2	4	$0 + 0 + 2 \cdot 3 \cdot 4 = 24$
$(A \cdot B) \cdot C$	1	4	$6 + 0 + 1 \cdot 3 \cdot 4 = \mathbf{18}$
$A \cdot (B \cdot C)$	1	4	$0 + 24 + 1 \cdot 2 \cdot 4 = \mathbf{32}$

- *Miten valitset edullisimman laskujärjestyksen?*

- Syötteenä saadaan pitkä matriisitulo

  $$M_1 \cdot M_2 \cdot M_3 \cdot \ldots \cdot M_n$$

  jossa
  - jokaisen matriisin M_i rivi- ja sarakelukumäärät tunnetaan
  - aina edellisen M_i sarakelukumäärä = seuraavan M_{i+1} rivilukumäärä.

- Ongelma **osittuu pienempiin tapauksiin** valitsemalla *viimeiseksi* laskettava matriisitulo, eli jakokohta $1 \leq k \leq n-1$ jolla lasketaan

  $$\underbrace{M_1 \cdot M_2 \cdot \ldots \cdot M_k}_{\text{alkupuoli}} \cdot \underbrace{M_{k+1} \cdot M_{k+2} \cdot \ldots \cdot M_n}_{\text{loppupuoli}}$$

  erikseen.

  $$\begin{aligned}\text{Askelmäärä} = &\text{alkupuolen askelmäärä}\\ +&\text{loppupuolen askelmäärä}\\ +&\text{viimeisen tulon askelmäärä}\end{aligned}$$

  missä

  $$\begin{aligned}\text{viimeinen} = &M_1\text{:n rivi-}\\ \cdot &M_k\text{:n sarake- eli } M_{k+1}\text{:n rivi-}\\ \cdot &M_n\text{:n sarakelukumäärä.}\end{aligned}$$

- Askelmäärän kaavasta nähdään:

  Alkupuoli kannattaa ratkaista edullisimmalla mahdollisella tavalla.

  Samoin loppupuoli.

  Siis ne ovat **optimaaliset alirakenteet**.

- Alkupuoli ja loppupuoli voidaan ratkaista toisistaan riippumatta:

  Alkupuolelle valittu edullisin ratkaisu ei estä valitsemasta loppupuolelle edullisinta ratkaisua.

  Siis nämä **osaongelmat ovat itsenäisiä**.

- Kun ratkaistaan alkupuolta, niin kokeillaan alijakoja

$$
\begin{aligned}
(M_1) \cdot (M_2 \cdot M_3 \cdot \ldots \cdot M_k) \\
(M_1 \cdot M_2) \cdot (M_3 \cdot M_4 \cdot \ldots \cdot M_k) \\
(M_1 \cdot M_2 \cdot M_3) \cdot (M_4 \cdot M_5 \cdot \ldots \cdot M_k) \\
\vdots \\
(M_1 \cdot M_2 \cdot \ldots \cdot M_{k-1}) \cdot (M_k)
\end{aligned}
$$

  joiden alkupuolia kokeillaan myös koko ongelmassa.

  Siis nämä **osaongelmat ovat päällekkäisiä**.

- Taulukoidaan päällekkäiset ratkaisut:

  $L[i][j] =$ pienin askelmäärä tulolle

$$M_i \cdot M_{i+1} \cdot M_{i+2} \cdot \ldots \cdot M_{i+j}$$

  eli "j matriisituloa matriisista M_i alkaen".

- Alustus: $L[i][0] = 0$ jokaisella $1 \leq i \leq n$.

- Suuremmilla indekseillä $j = 1, 2, 3, \ldots, n-1$:

  $L[i][j] =$ pienin askelmääristä

$$
\begin{aligned}
& L[i][k] \\
& + L[i+k+1][j-k-1] \\
& + M_i\text{:n rivi-} \\
& \quad \cdot M_k\text{:n sarake-} \\
& \quad \cdot M_j\text{:n sarakelukumäärä}
\end{aligned}
$$

  missä vaihtoehdot ovat $0 \leq k \leq j-1$.

- Lopputulos on $L[1][n-1]$.

- Jos halutaan myös itse laskujärjestys, niin liitetään kuhunkin $L[i][j]$ sen minimoinut k.

10.2.2 Floydin algoritmi

Itse asiassa kehitimme Floydin algoritmin kalvoilla 9.1.2 dynaamisella ohjelmoinnilla:

Erilaiset purkutavat: Eri vaihtoehdot korkeimmalla sisäsolmulle $1 \leq k \leq n$ lyhyimmällä polulla (2).

Optimaaliset alirakenteet: Myös alku- ja loppuosat ovat lyhyimpiä polkuja.

Muutenhan polkua (2) voisi lyhentää.

Itsenäiset osaongelmat: Kun valitaan lyhyintä loppuosaa, niin ei tarvitse tietää, millainen lyhyin alkuosa valittiin.

Päällekkäiset osaongelmat:

$$
\begin{aligned}
W(i,j,k+2) &= \min(W(i,j,k+1), \\
&\qquad W(i,k+2,k+1) + \\
&\qquad W(k+2,j,k+1)) \\
W(i,j,k+1) &= \min(W(i,j,k), \\
&\qquad \boxed{W(i,k+1,k)} + \\
&\qquad W(k+1,j,k)) \\
W(i,k+2,k+1) &= \min(W(i,k+2,k), \\
&\qquad \boxed{W(i,k+1,k)} + \\
&\qquad W(k+1,k,k))
\end{aligned}
$$

- Kysymyksiin "mikä on lyhyin polku solmusta i solmuun j?" vastattiin

  - tulostamalla *perustelu* sille, miten sen pituus $W[i][j]$ saatiin laskettua

  - käyttämällä apuna lisätietomatriisia $K[i][j]$.

- Sama idea pätee kaikessa dynaamisessa ohjelmoinnissa:

  - Optimiarvoa vastaava toimintatapa saadaan *jäljittämällä taaksepäin, miten optimiarvo saatiin* eri tapoja valitsemalla.

  - Jäljittämistä varten voidaan tarvita aputietorakenteita.

    Tässä lisätietomatriisia $K[i][j]$.

10.2.3 Editointietäisyys

[1, ongelma 15-3; 8, luvut 11.2–11.4]

- Kahden merkkijonon
  $$s = p_1 p_2 p_3 \dots p_m$$
  $$t = q_1 q_2 q_3 \dots q_n$$
  välinen *editointietäisyys* (edit distance) on pienin määrä *muokkausaskeleita*, joilla merkkijono s saadaan muokattua merkkijonoksi t.

- Muokkausaskeleet ovat *yhden merkin*

  vaihto toiseksi

  `shot` ↦ `spot`

  lisäys

  `ago` ↦ `agog`

  poisto

  `hour` ↦ `our`.

- Editointietäisyydellä mitataan esimerkiksi DNA-sekvenssien samankaltaisuutta.

- Havainto:
  - Muokkauskohtaa voidaan kuljettaa jonossa s koko ajan *eteenpäin* vasemmalta oikealle.
  - Muokkauskohdan peruuttaminen taaksepäin ei johda etenemistä parempaan ratkaisuun.

- Havainto takaa

  optimaaliset alirakenteet

  itsenäiset osaongelmat.

- Lasketaan siis matriisi $D[i][j] =$ alkuosien
  $$s = p_1 p_2 p_3 \dots p_i$$
  $$t = q_1 q_2 q_3 \dots q_j$$
  välinen editointietäisyys.

- Reunaehdot tyhjille alkuosille:
  $D[0][j] = j$ **lisäys**tä
  $D[i][0] = i$ **poisto**a.

- Entä matriisin sisäpaikat $D[i+1][j+1]$?

- Kun muokkauskohta on merkin p_{i+1} päällä, niin

  se voidaan säilyttää samana mutta vain jos $p_{i+1} = q_{j+1}$. Silloin koko hinta on $D[i][j]$ muokkausaskelta.

  se voidaan korvata merkillä q_{j+1}. Silloin koko hinta onkin $D[i][j] + 1$ muokkausaskelta.

  se voidaan poistaa. Silloin koko hinta on $D[i][j+1] + 1$ muokkausaskelta.

  sen perään voidaan lisätä merkki q_{j+1}. Silloin koko hinta on $D[i+1][j] + 1$ muokkausaskelta.

  Nämä ovat **erilaiset purkutavat**.

- Valitaan pienin:
  $$D[i+1][j+1] = \min(D[i][j] + e,\; D[i][j+1] + 1,\; D[i+1][j] + 1)$$
  missä
  $$e = \begin{cases} 0 & \text{jos } p_{i+1} = q_{i+1} \\ 1 & \text{muuten.} \end{cases}$$

- Tarvitaan vielä matriisin D täyttö*suunta:*
  - Paikkaan $D[i+1][j+1]$ tarvitaan
    1. $D[i][j] =$ edellisen rivin i edellinen sarake j
    2. $D[i][j+1] =$ edellisen rivin i sama sarake $j+1$
    3. $D[i+1][j] =$ saman rivin $i+1$ edellinen sarake j.
  - Täytetään siis
    * rivi kerrallaan
    * rivin sisällä vasemmalta oikealle.

- Minimivastaus ilmestyy viimeiseen taulukkopaikkaan $D[m][n]$.

- Minimivastauksen saavuttavien muokkausoperaatioiden jono voidaan tulostaa
  - tästä viimeisestä taulukkopaikasta $D[m][n]$ lähtien
  - rekursiokutsulla muokkaus(m, n)
  - joka jäljittää taaksepäin paikoissa $D[i][j]$ olevien osavastausten (jonkin) syntyhistorian
  - näiden osavastausten perusteella
  - ilman erillista aputietorakennetta.

- Jos halutaan erillinen aputietorakenne, niin sellainen on $C[i][j] =$ "saatiinko paikan $D[i][j]$ arvo täyttösuunnasta 1, 2 vai 3?"

  Näiden suuntien avulla voitaisiin tehdä sama rekursio. [8, taulukot sivulla 249]

procedure muokkaus(i, j: **indeksi**)
  if $i = 0$ **then**
    if $j > 0$ **then**
      muokkaus($i, j - 1$);
      Tulosta "lisää merkki q_j"
    end if
  else if $j = 0$ **then**
    if $i > 0$ **then**
      muokkaus($i - 1, j$);
      Tulosta "poista merkki p_i"
    end if
  else if $D[i][j] = D[i-1][j-1]$ **then**
    muokkaus($i - 1, j - 1$);
    Tulosta "säilytä merkki p_i"
  else if $D[i][j] = D[i-1][j-1] + 1$ **then**
    muokkaus($i - 1, j - 1$);
    Tulosta "korvaa p_i merkillä q_j"
  else if $D[i][j] = D[i-1][j]$ **then**
    muokkaus($i - 1, j$);
    Tulosta "poista merkki p_i"
  else
    muokkaus($i, j - 1$);
    Tulosta "lisää merkki q_j"
  end if.

		y	o	u	-	s	h	o	u	l	d	-	n	o	t
:	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	0	0	0	0	0	0	1	1	1	1	1	1	1	1
o:	2	0	0	0	0	0	0	0	1	1	1	1	1	0	1
u:	2	0	2	0	1	1	1	1	0	1	1	1	1	1	1
-:	2	0	2	2	0	1	1	1	1	0	0	0	1	1	1
s:	2	0	2	2	2	0	1	1	1	1	0	0	0	0	0
h:	2	0	2	2	2	2	0	1	1	1	1	1	1	0	0
a:	2	0	2	2	2	2	2	0	0	0	0	0	0	0	0
l:	2	0	2	2	2	2	2	0	0	0	1	1	1	1	1
t:	2	0	2	2	2	2	2	0	0	0	0	0	0	0	0
-:	2	0	2	2	0	2	2	0	0	0	0	0	1	1	1
n:	2	0	2	2	2	2	2	0	0	0	0	2	0	1	1
o:	2	0	0	2	2	2	2	0	0	0	0	2	2	0	1
t:	2	0	2	2	2	2	2	2	0	0	0	2	2	2	0

		y	o	u	-	s	h	o	u	l	d	-	n	o	t
:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	2	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	3	2	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	4	3	2	3	4	5	6	6	7	7	8	9	10
s:	6	6	5	4	3	2	3	4	5	6	7	8	8	9	10
h:	7	7	6	5	4	3	2	3	4	5	6	7	8	9	10
a:	8	8	7	6	5	4	3	3	4	5	6	7	8	9	10
l:	9	9	8	7	6	5	4	4	4	4	5	6	7	8	9
t:	10	10	9	8	7	6	5	5	5	5	5	6	7	8	8
-:	11	11	10	9	8	7	6	6	6	6	6	5	6	7	8
n:	12	12	11	10	9	8	7	7	7	7	7	6	5	6	7
o:	13	13	12	11	10	9	8	7	8	8	8	7	6	5	6
t:	14	14	13	12	11	10	9	8	8	9	9	8	7	6	5

10.2.4 Pisin nouseva alijono

Joskus dynaamisessa ohjelmoinnissa onkin laskettava jokaiselle *osaongelmalle kokoelma eri tavoin hyviä osaratkaisuja* koska etukäteen ei vielä tiedetä mitä niistä tarvitaan myöhemmin parhaaseen kokonaisratkaisuun.

Tarkastellaan esimerkkinä *pisimmän aidosti nousevan alijonon* (longest monotonically increasing subsequence) [1, tehtävät 15.4-5&6] ongelmaa:

Annetaan lukujono

$$p_1, p_2, p_3, \ldots, p_n.$$

Kysytään mahdollisimman pitkää indeksijonoa

$$1 \leq i_1 < i_2 < i_3 < \cdots < i_m \leq n$$

jolla aina

$$p_{i_{j-1}} < p_{i_j}.$$

Idea: Lasketaan jokaiselle alkuosalle

$$p_1, p_2, p_3, \ldots, p_k$$

sen pisin indeksijono

$$\mathcal{I} = i_{k_1}, i_{k_2}, i_{k_3}, \ldots, i_{k_\ell}.$$

Ongelma: Seuraavan alkuosan $k+1$ paras indeksijono ei ehkä olekaan edellisen alkuosan pisimmän jonon $\mathcal{I}$ jatke.

Seuraava luku voi näet olla liian pieni jatkamaan jonoa $\mathcal{I}$, eli $p_{i_{k_\ell}} \geq p_{k+1}$.

Mutta silti tarpeeksi suuri jatkamaan *toiseksi pisintä* indeksijonoa

$$\mathcal{I}' = i'_{k'_1}, i'_{k'_2}, i'_{k'_3}, \ldots, i'_{k'_{\ell-1}},$$

eli $p_{i'_{k'_{\ell-1}}} < p_{k+1}$.

Silloin jono $\mathcal{I}', (k+1)$ on parempi kuin $\mathcal{I}$:

- Molempien pituus on sama ℓ.
- Jonoa $\mathcal{I}', (k+1)$ voi jatkaa *ainakin* samoilla vaihtoehdoilla kuin jonoa $\mathcal{I}$ — ehkä jopa paremmillakin!

Ratkaisu: Jotta saataisiin haluttu paras

pisin indeksijono niin tarvitaan paras

toiseksi pisin jono johon tarvitaan paras

kolmanneksi pisin...

joten lasketaan ne kaikki.

Siis: Lasketaan jokaiselle alkuosalle

- m_k = pisin indeksijonon pituus
- taulukko $I_k[1 \ldots m_k]$ jossa
  - $I_k[\ell]$ = indeksi johon päättyy jokin pituutta ℓ oleva indeksijono, ja
  - mikään muu saman pituinen indeksijono ei voi päättyä aidosti pienempään arvoon kuin $p_{I_k[\ell]}$.

Silloin koko tehtävän vastaus on m_n.

Alustus: $m_1 = 1$ ja $I_1[m_1] = 1$.

Eteneminen: Olkoot m_k ja I_k lasketut.

Onko olemassa ℓ jolla

$$p_{I_k[\ell-1]} < p_{k+1} < p_{I_k[\ell]} \tag{11}$$

rajatapauksina

$$p_{I_k[0]} = -\infty$$
$$p_{I_k[m_k+1]} = +\infty?$$

Jos *on* niin

$$m_{k+1} = \text{jos } \ell = m_k + 1 \text{ niin } \ell \text{ muuten } m_k$$
$$I_{k+1} = I_k \text{ paitsi että } I_{k+1}[\ell] = k+1.$$

Jos *ei* niin

$$m_{k+1} = m_k$$
$$I_{k+1} = I_k.$$

Muistia kuluu $O(n)$ — taulukko I_{k+1} voidaan tehdä päivittämällä taulukkoa I_k.

Askeleita kuluu silloin $O(n \cdot m_n)$.

Lisähavainto: Päivitysehdon (11) nojalla

$$p_{I_k[1]} < p_{I_k[2]} < p_{I_k[3]} < \cdots < p_{I_k[m_k]}$$

joten päivityskohtaa ℓ voidaan etsiä kalvojen 10.1.1 binäärihaulla.

Askeleita kuluu enää $O(n \cdot \log_2 m_n)$.

Itse indeksijonot voidaan jäljittää liittämällä jokaiseen p_{k+1} lisäkenttä:

link(p_{k+1}) = taulukkoalkion $I[\ell - 1]$ arvo sillä hetkellä kun indeksi $k+1$ vietiin taulukkoalkioonsa $I[\ell]$ (jos sellaista on).

10.2.5 Pisin ei-laskeva alijono

- Entä jos kalvojen 10.2.4 tehtävässä kysytäänkin sellaista indeksijonoa
$$1 \leq i_1 < i_2 < i_3 < \cdots < i_m \leq n$$
jolla aina
$$p_{i_{j-1}} \leq p_{i_j}$$
eli joka *ei laske?*

- Muistetaan kunkin syöteluvun järjestysnumero
$$\langle p_1, 1\rangle, \langle p_2, 2\rangle, \langle p_3, 3\rangle, \ldots, \langle p_n, n\rangle$$
ja määritellään näille pareille järjestys $\langle p, i\rangle < \langle q, j\rangle$ seuraavasti:
  - joko $p < q$
  - tai $p = q$ mutta $i < j$.

- Siis myöhemmin tuleva alkio tulkitaan

  suuremmaksi kuin edelliset samankokoiset

  pienemmäksi kuin sitä suuremmat alkiot.

10.2.6 Esimerkki: Isompi parempi?

- Tehtävässä [8, Problem 11.6.1]

  annetaan elefanttien mittoja, jokaisesta

  paino ja

  älykkyysosamäärä (ÄO)

  satunnaisessa järjestyksessä

  pyydetään sellainen mahdollisimman pitkä lista elefantteja, jossa

  paino kasvaa

  ÄO vähenee

  aidosti.

- Ratkeaa yhdistämällä
  1. syötteen esilajittelu kalvoilta 3.3 säännöllä "kevyempi ensin, samanpainoisista viisaampi"
  2. kalvojen 10.2.4 algoritmiin laskevan ÄO:n suhteen.

10.3 Ahneista algoritmeista

- Joskus käy seuraavasti:
  1. Lähdetään ratkaisemaan kisatehtävää dynaamisella ohjelmoinnilla.
  2. Huomataan, että jokin tietty tapa g purkaa S on *aina paras*.

- Silloin voidaankin
  - valita aina automaattisesti g
  - jättää muut tavat kokonaan laskematta

  ja saadaan tehokkaampi algoritmi.

- Tällaisia algoritmit ovat *ahneita* (greedy):
  - ne valitsevat aina vaihtoehdon g edes harkitsematta muita
  - koska vaihtoehto g *näyttää* niin houkuttelevalta. . .

  [1, luku 16; 6, luku 9]

- Esimerkiksi Dijkstran algoritmi (kalvoilta 9.1.1) on ahne, koska se
  - kasvattaa joka silmukkakierroksella π-puuta sillä solmulla u, jonka kenttä matka[u] on pienin, eli houkuttelevin
  - ei tutki muita tapoja kasvattaa π-puuta
  - luottaa siihen, että
    - *paikallisesti* parhaan valinnan tekeminen
    - johtaa lopulta *kokonaisuutena* parhaaseen mahdolliseen tulokseen.

- Ahneen algoritmin kehittäminen "nollasta" on *kisatilanteessa vaarallista:*
  - Oletko *varma*, että ahneesti saa saman tuloksen kuin systemaattisesti kaikki vaihtoehdot tutkivalla dynaamisella ohjelmoinnilla?
  - On *mutkikkaampaa* analysoida kisatehtävä ahneuteen asti kuin dynaamiseen ohjelmointiin.

10.3.1 Jatkuva repunpakkausongelma

Tarkastellaan seuraavaa *repunpakkausongelmaa* (knapsack) [1, sivut 382–383; 6, sivut 392–395]:

- Olet murtautunut *kangas*kauppaan.
- Reppuusi mahtuu korkeintaan W m kangasta.
- Tarjolla on kangaspakat $i = 1, 2, 3, \ldots, n$ joilla
  - pakassa i on $w_i > 0$ m
  - kankaasta i saa luukuttajalta $v_i > 0$ €/m.

  Pakoista *voi leikata* kangasta saksilla haluamansa määrän.
- Miten maksimoit reppusi arvon?

- Tietenkin *ahneesti:*
  1. Ala täyttää reppuasi metrihinnaltaan kalleimmalla kankaalla.
  2. Jos se loppuu, ja repussa on vielä tilaa, niin jatka seuraavaksi kalleimmalla kankaalla.
  3. Ja niin edelleen, kunnes reppusi on täysi (tai kauppa tyhjä).
- Riittää siis lajitella pakat metrihinnan mukaan
  $$O(n \cdot \log_2 n)$$
  askeleessa

  (vaikkapa kalvojen 3.2 pikajärjestämisellä).

10.3.2 Diskreetti repunpakkausongelma

Repunpakkausongelman yleinen tapaus [1, sivut 382–383; 6, luku 8.4] on kuitenkin diskreetti:

- Oletkin murtovarkaissa *elektroniikka*liikkeessä.
- Reppusi kestää korkeintaan W kg saalista.
- Tarjolla on laitteet $i = 1, 2, 3, \ldots, n$ joilla
  - laite i painaa $w_i > 0$ kg
  - laitteesta i saa luukuttajalta $v_i > 0$ €.

  Kukin laite täytyy joko *ottaa tai jättää kokonaan*, niitä ei saa purkaa osiin.
- Miten nyt maksimoit reppusi arvon?

- Repunpakkausongelma on $\mathbf{NP}$-täydellinen [4, ongelma MP9]:

  Aidosti polynomista ($\approx$ tehokasta) algoritmia ei siis kannata etsiä.
- Kalvojen 10.3.1 jatkuva versio pysyi polynomisena:

  Repun sai aina täyteen mahdollisimman kalliita kankaita.
- Tämä diskreetti versio on vaikeampi:

  Kannattaako ottaa

  vajaampi reppu kalliimpia

  täydempi reppu halvempia

  laitteita?
- Eri pakkausvaihtoehtoja on 2^n kpl.
- Keskeneräisestä pakkausvaihtoehdosta on vaikea sanoa etukäteen, saako siitä hyvän vai huonon lopputuloksen.

- Kalvoilla 7.2 samaa ongelmaa ratkottiin rajoittavalla etsinnällä.
  - Siellä käytetty keskeneräisen vaihtoehdon arvio muistuttaakin kalvojen 10.3.1 jatkuvaa tapausta:
    * Lasketaan ikään kuin kallein jäljellä oleva tavara olisikin kangasta, ja sitä olisi riittävästi.
    * Esimerkki heuristisen mitan johtamisesta alkuperäisen tehtävän *rajoituksia lieventämällä*.
  - Etsintä onkin usein hyvä tapa ratkaista vaikea ongelma:
    * Vastaus *saattaa* löytyä nopeastikin, jos olemme onnekkaita!
    * Hyvä heuristinen mitta kasvattaa mahdollisuuksiamme.
- Toinen mahdollisuus olisi yrittää käydä läpi kaikki vaihtoehdot *systemaattisesti mutta tehokkaasti*.

  Yritetään seuraavaksi sitä.

- Edetään kalvojen 10 dynaamisella ohjelmoinnilla — vältetään samanlaisten vaihtoehtojen toistuva käsittely.
- Johdetaan palautuskaava

  hinta$(i,u) =$ paras hinta joka voidaan saada kun valitaan laitteista $1,2,3,\ldots,i$ korkeintaan u kg painoinen yhdistelmä

  parametrin i suhteen:
  - hinta$(0,u) = 0$ — ei ole mistä valita.
  - Jos $i > 0$ niin laitteen i voit

    ottaa reppuun, mutta vain jos se mahtuu eli $w_i \geq u$, ja silloin luukuttajan tarjous on hinta$(i-1, u-w_i) + v_i$

    jättää pois, jolloin tarjous onkin hinta$(i-1,u)$.

    Valitse niistä korkeampi vaihtoehto.
- Silloin paras saalis on hinta(n, W).

- Voisimme taulukoida hinta$[i,u]$ mutta
  - painot pitäisi ensin skaalata kokonaisluvuiksi
  - naapureissa aina

    $$\text{hinta}[i,u-1] \leq \text{hinta}[i,u]$$

    ja usei(mmite)n

    $$\text{hinta}[i,u-1] = \text{hinta}[i,u]$$

    joten kannattaa muistaa vain muutoskohdat u joissa

    $$\text{hinta}[i,u-1] < \text{hinta}[i,u].$$
- Aputietorakenteiksi riittää siis

  $$D_i = \{\langle p_1, q_1\rangle, \langle p_2, q_2\rangle, \langle p_3, q_3\rangle, \ldots, \langle p_m, q_m\rangle\}$$

  missä
  - pari $\langle p_j, q_j\rangle$ tarkoittaa "painolla p_j kg saadaan hinta q_j €"
  - on järjestys $p_j < p_{j+1} \leq W$ ja $q_j < q_{j+1}$. — karsitaan huonoja vaihtoehtoja.

- Aputietorakenteeksi riittää järjestetty lista.

Lista D sisältäköön aluksi vain parin $\langle 0,0\rangle$**;**
for all laitteet $1 \leq i \leq n$ **do**
  Tee listasta D muokattu kopio D' jossa
  – on mukana parit $\langle p_j + w_i, q_j + v_i\rangle$
  – mutta ei häntää jossa $p_j + w_i > W$
  – järjestys säilyy**;**
  Lomita listat D ja D' siten että verrattaessa pareja $\langle p,q\rangle \in D$ ja $\langle p',q'\rangle \in D'$:
  – Jos $p < p'$ niin viedään listan D pari tuloslistan loppuun.
  – Jos $p > p'$ niin viedäänkin listan D' pari.
  – Jos $p = p'$ niin viedäänkin pari $\langle p, \max(q,q')\rangle$.
  – Edetään listoissa D ja D' ohi niiden parien joiden hintakenttä on korkeintaan yhtä suuri kuin tuloslistaan viedyssä parissa.
  Lomituksen tulos olkoon seuraava D
end for;
return (epätyhjän) listan D viimeisen parin hintakenttä**.**

11 Ruudukoista

[8, luku 12]

- Usein kisatehtävissä käsitellään *ruudukkoja* (grid), jolla on säännöllisenä toistuva rakenne.

- Ruudukko voi koostua esimerkiksi

  neliöistä kuten shakkilauta

  kuusikulmioista kuten hunajakenno

  kolmioista kuten kalvoilla 11.3...

- Tehtävässä voidaan olla kiinnostuneita ruutujen

  sisäalueista kuten pelilaudoilla

  reunaviivoista ja risteyksistä kuten kaupungissa, jossa on ruutukaava.

- Usein kisatehtävän voi *mallintaa verkkona* G jonka solmut ja kaaret antaa ruudukko:

  pelilaudalla verkon

  solmuina ovat ruudut joissa nappula voi olla

  kaarina ovat mahdolliset siirrot ruudusta toiseen.

  ruutukaavassa verkon

  solmuina ovat katujen risteykset

  kaarina kadut.

- Silloin verkon G voi *toteuttaa helpommin*
  - sijoittamalla solmut *taulukkoon*, jonka indeksointitapa noudattaa ristikon rakennetta
  - korvaamalla kaaret niitä vastaavilla *indeksien laskutoimituksilla*.

11.1 Esimerkki: Vankilapako

- Tarkastellaan seuraavaa vankilapako-ongelmaa:
  - Syötteenä saadaan vankilan pohjapiirros matriisina $B[0 \ldots 2 \cdot m][0 \ldots 2 \cdot m]$.
  - Jokainen paikka $B[i][j]$ on joko tyhjää *käytävä*ä tai *muuri*a $\bullet$.

●			●	●
●		●		●
●	●	○	●	
●			●	●
	●	●		●

- Sinä $\circ$ olet aluksi vankilan keskipaikassa $B[m][m]$ (joka on käytävää).

- Tavoitteenasi on päästä vankilasta sen jonkin reunan yli vapauteen.

- Vankilassa voi liikkua seuraavasti:
  - Nykyisestä paikasta voi siirtyä mihin tahansa naapuripaikkaan, ei kulmittain.
  - Käytävillä voi hiiviskellä, mutta niillä *ei kannata maleksia* turhaan.
  - Muuriin täytyy ensin *kaivautua* ennen kuin siihen voi kulkea.

- Hyvä pakosuunnitelma on sellainen, jossa on mahdollisimman vähän

  1. kaivamista
  2. käytävillä hiiviskelyä

  tässä järjestyksessä.

  Siis pakosuunnitelma $\mathcal{A}$ on aidosti parempi kuin $\mathcal{B}$ jos

  1. joko $\mathcal{A}$ sisältää vähemmän kaivamista kuin $\mathcal{B}$
  2. tai ne sisältävät yhtä paljon kaivamista, mutta $\mathcal{A}$ sisältää vähemmän käytävillä hiiviskelyä kuin $\mathcal{B}$.

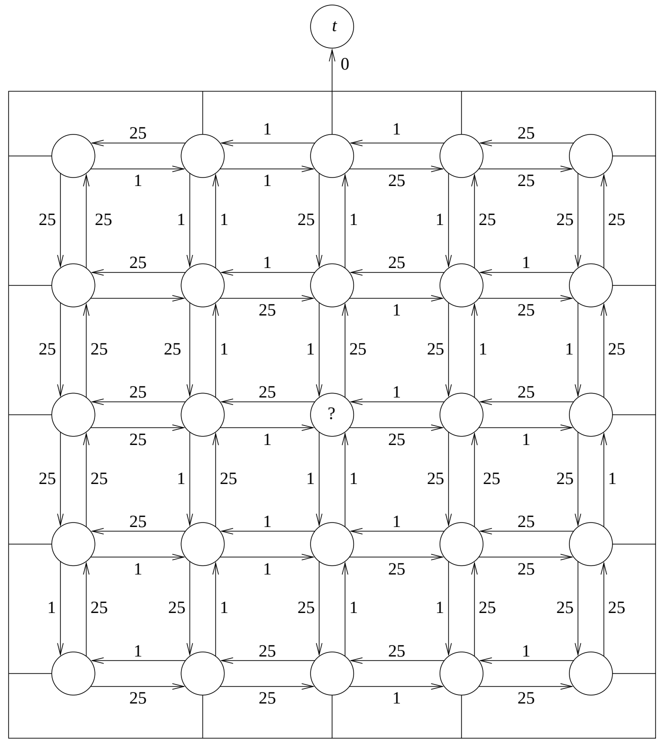
- Mallinnetaan paon suunnittelu lyhyimpänä polkuna verkossa G:
  - $V(G) =$ vankilan paikat sekä lisäpaikka t eli "vapaudessa".
  - $p \xrightarrow{w} q \in E(G)$ jos ja vain jos paikka q on paikan p naapuripaikka.

    Jokaisen reunapaikan p naapurina on myös lisäpaikka t.
  - Kaaren pituuden $w \geq 0$ määrää kohdepaikka q:

    käytävällä $w = 1$

    muurilla $w = (2 \cdot m + 1)^2$ eli vankilan koko pinta-ala
    * lyhyin polku on kehätön (koska pituudet $w \geq 0$)
    * siis yksikin kaivautuminen on huonompi kuin pisinkään hiiviskely
    * siis samaa kaivettua aukkoa ei kuljeta kahdesti

    reunalla $w = 0$ kun $q = t$.



- (Jokin) mahdollisimman hyvä pakosuunnitelma löytyy siis
  - ajamalla Dijkstran algoritmia (kalvoilta 9.1.1)
  - verkossa G
  - alkusolmusta $s = B[m][m]$ lähtien
  - kunnes käsittelyvuoroon tulee solmu t
  - jolloin pakosuunnitelma on polku

$$t \leftarrow \pi[t] \leftarrow \pi[\pi[t]] \leftarrow \cdots \leftarrow s$$

- Ristikkorakenteen vuoksi verkkoa G ei tarvitse muodostaa erillisenä tietorakenteena:
  - Solmun $B[i][j]$ kaaret vievät solmuihin $B[i \pm 1][j]$ ja $B[i][j \pm 1]$.
  - Indeksoinnin ylitys tarkoittaa lisäsolmua t.
  - Kaaren pituus katsotaan kohdesolmusta.

- Pituuksia voi käsitellä selkeämminkin:
  - Käytetäänkin luku*pareja* $\langle a, b \rangle$ missä
    * $a =$ muuri-
    * $b =$ käytävä-

    paikkojen lukumäärä polulla.
  - Kaaren pituus w on vakiopari

    käytävällä $w = \langle 0, 1 \rangle$

    muurilla $w = \langle 1, 0 \rangle$

    reunalla $w = \langle 0, 0 \rangle$.
  - $\langle a, b \rangle \leq \langle c, d \rangle$ kun parit ovat sanakirjajärjestyksessä:

$$a < c \textbf{ or } (a = c \textbf{ and } b \leq d)$$

    Käytetään tätä ehtoa polkujen vertailuun.
  - Polkujen kasvatukseen käytetään yhteenlaskua

$$\langle a, b \rangle + \langle c, d \rangle = \langle a + c, b + d \rangle .$$

11.2 Esimerkki: Numerokiekkopeli

- Tehtävää [8, Problem 9.6.2] voi mallintaa ruudukolla seuraavasti:
  - Käytetään 4-ulotteista ruudukkoa $B[0\ldots 9][0\ldots 9][0\ldots 9][0\ldots 9]$.

    Siis kiekkojen asemaa p, q, r, s vastaa ruutu $B[p][q][r][s]$.
  - Ruudulla $B[p][q][r][s]$ on 8 mahdollista naapuria

    $$\begin{aligned} &B[p \pm 1][q][r][s] \\ &B[p][q \pm 1][r][s] \\ &B[p][q][r \pm 1][s] \\ &B[p][q][r][s \pm 1] \end{aligned}$$

    vastaten kiekkojen pyöräytyksiä myötä- ja vastapäivään.
  - Koska jokainen kiekko on pyöreä ja 10-paikkainen, niin tämä indeksiaritmetiikka suoritetaan $\pmod{10}$ kalvoilta 5.4, eli

    $$\begin{aligned} -1 &\mapsto 9 \\ 10 &\mapsto 0. \end{aligned}$$

- Kielletyt kiekkojen asemat merkitään taulukkoon B.

  Nämä kielletyt naapurit jätetään pois.

- Sitten ongelma
  - on löytää vähäkaarisin polku annetusta lähtösolmusta annettuun kohdesolmuun
  - ratkeaa kalvojen 8.5 leveyssuuntaisella läpikäynnillä

  tässä naapuriverkossa.

(Tämän ratkaisun laati Niko Kiirala vuoden 2004 olympiavalmennusleirillä.)

11.3 Kolmiot ja kennot

- Joskus pelilaudan ruudut ovatkin tasasivuisia kolmioita neliöiden sijasta, kuten seuraavassa kuvassa.

  [8, kuva 12.1]

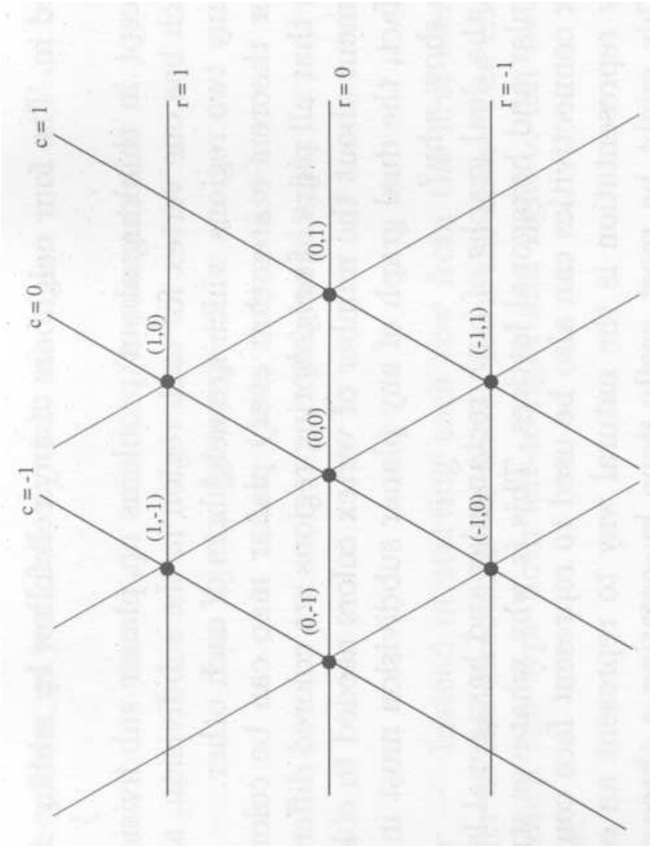
- Näiden ruutujen kulmapisteet voidaan esittää kahdella koordinaatilla kuvan mukaisesti:

  Ensin rivinumero r vastaa y-koordinaattia.

  Sitten sarakenumero c ilmoittaa sen x-koordinaatin, jossa vastaava nouseva suora leikkaa rivin $r = 0$.

- Silloin kulmapisteen $\langle r, c\rangle$ 6 naapuria ovat

  $$\langle r, c \pm 1\rangle, \langle r \pm 1, c\rangle \quad \text{ja} \quad \langle r \pm 1, c \mp 1\rangle.$$



- Usein käsitellään vain niitä kulmapisteitä, jotka ovat
  - ylös
  - oikealle

  origosta $\langle 0, 0 \rangle$.

- Silloin rivi $r \geq 0$ koostuu sarakkeista numero

$$c = -\left\lfloor \frac{r}{2} \right\rfloor, -\left\lfloor \frac{r}{2} \right\rfloor + 1, -\left\lfloor \frac{r}{2} \right\rfloor + 2, \ldots$$

- Siitä saadaan kompakti talletusjärjestys matriisiin $M[0 \ldots, 0 \ldots]$:

  Käsiteltävä kulmapiste $\langle i, j \rangle$ talletetaan paikkaan

$$M\left[i, j + \left\lfloor \frac{i}{2} \right\rfloor\right].$$

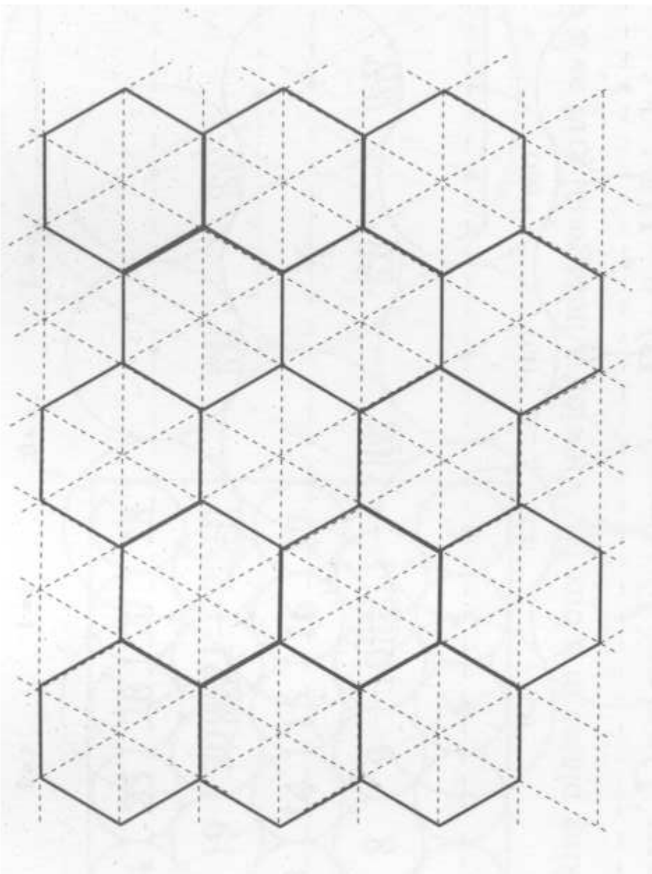
- Esimerkiksi strategiapeleissä laudan ruudut ovatkin kuusikulmioita ("hunajakenno"), kuten seuraavassa kuvassa.

  [8, kuva 12.2]

- Jos naapurikuusikulmioiden keskipisteet yhdistää viivalla, niin saa saman kolmiokoordinaatiston kuin yllä.

- Kolmio- ja kuusikulmiokoordinaatistot ovatkin toistensa *duaalit:*

  ruudun keskipisteet toisessa vastaavat kulmapisteitä toisessa.



12 Laskennallisesta geometriasta

[1, luku 33; 8, luku 14]

Laskennallisessa geometriassa (Computational Geometry) syötteenä annetaan jokin

- yleensä xy-tason eli *2-ulotteinen kuvio*
- *pisteinä* eli koordinaattipareina $\langle x, y \rangle$
- näiden pisteiden avulla määriteltyinä rakenteina:
  - vektoreina
  - janoina
  - suorina
  - monikulmioina
  - ...

- Tehtävänä on tyypillisesti

  tunnistaa jokin kuvion ominaisuus

  kuten "leikkaavatko kuviossa olevat janat toisiaan" [1, luku 33.2]

  "piirtää" kuvioon lisää tulostamalla ne pisteet, jotka pitää yhdistää viivoilla

  kuten "ympyröi kuvion pisteet" kalvoilta 12.2.

12.1 Ristitulo

- Kahden paikkavektorin *ristitulo* (cross product) [1, luku 33.1]

$$\langle x_1, y_1\rangle \times \langle x_2, y_2\rangle = \text{det}\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 \cdot y_2 - x_2 \cdot y_1$$

  ratkaisee monet pulmat *peruslaskutoimituksilla*.

- Kun $p_1 \times p_2$ on

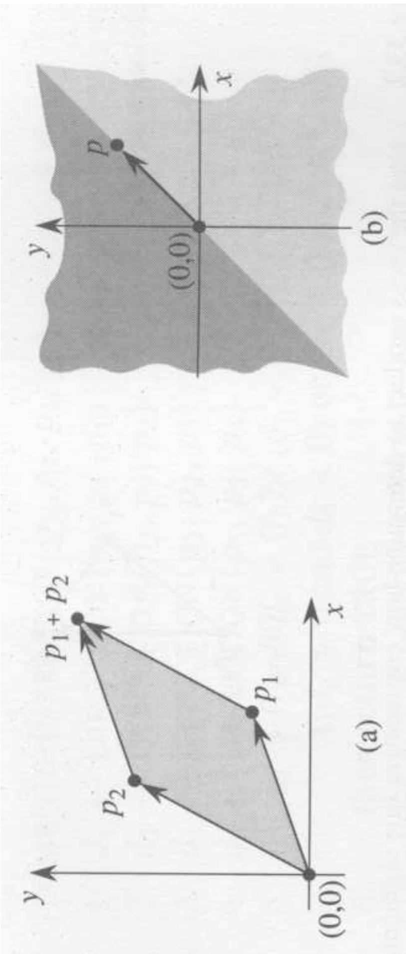
  positiivinen niin piste p_1 näkyy kääntämällä päätä *oikealle/myötäpäivään*

  negatiivinen niin *vasemmalle/vastapäivään*

  nolla niin suoraan *edessä tai takana* eli samalla suoralla

  kun origosta katsotaan kohti pistettä p_2

  [1,kuva 33.1 (b)]



- Onko suunnattu jana $\overrightarrow{p_0p_1}$ myötä- vai vastapäivään suunnatusta janasta $\overrightarrow{p_0p_2}$ kun katsotaan niiden yhteisestä alkupisteestä p_0?

  Siirretään alkupiste origoon!

  Eli lasketaan $(p_1 - p_0) \times (p_2 - p_0)$.

- Kääntyvätkö peräkkäiset janat $\overline{p_0p_1}$ ja $\overline{p_1p_2}$ oikealle vai vasemmalle niiden yhteisessä pisteessä p_1?

  Ratkaistaan onko suunnattu jana $\overrightarrow{p_0p_2}$ myötä- vai vastapäivään suunnatusta janasta $\overrightarrow{p_0p_1}$

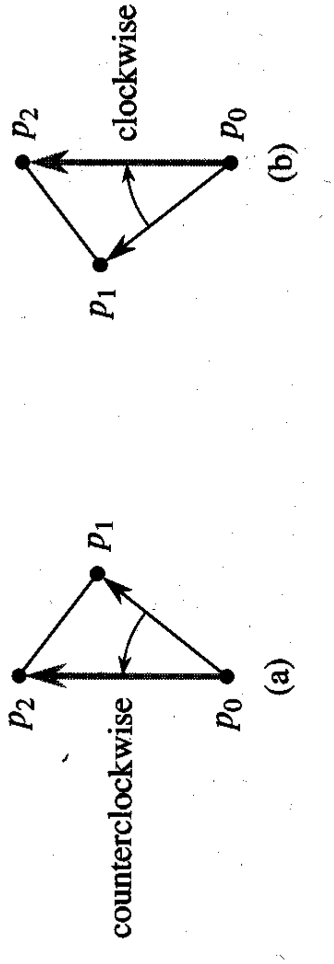
  [1, kuva 33.2]

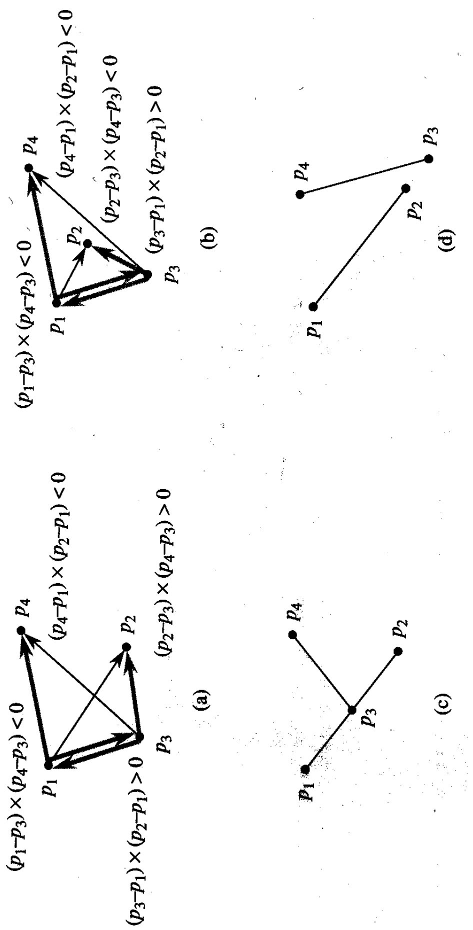
- Kulkeeko (straddles) jana $\overline{p_1p_2}$ yli sen suoran s jolla on jana $\overline{p_3p_4}$?

  Eli ovatko p_1 ja p_2 suoran s eri puolilla? (Tai rajatapauksena toinen niistä suoralla.)

  Kyllä ovat jos suunnatut janat $\overrightarrow{p_3p_1}$ ja $\overrightarrow{p_3p_2}$ ovat eri puolilla suunnattua janaa $\overrightarrow{p_3p_4}$

  [1, kuvat 33.3 (a) ja (b)].





- Leikkaavatko janat $\overline{p_1p_2}$ ja $\overline{p_3p_4}$?

  Kyllä jos ainakin toinen seuraavista pätee:

  - Kumpikin leikkaa toisensa suoran.
  - Toisen päätepiste on toisen sisällä (rajatapauksena).

  [1, kuvat 33.3 (c) ja (d)]

  Algoritmissa [1, sivu 937] haaraudutaan eri tapauksiin.

- Lisäksi $|p_1 \times p_2|$ antaa sen suunnikkaan alan, jonka kärkipisteet ovat origo, p_1, p_2 ja $p_1 + p_2$

  [1, kuva 33.1 (a)]

- Silloin kolmen pisteen välisten janojen muodostaman kolmion ala saadaan

  - siirtämällä yksi pisteistä origoon
  - laskemalla kahden muun pisteen suunnikkaan ala
  - puolittamalla tulos.

12.2 Kupera peite

[1, luku 33.3; 6, luku 3.3]

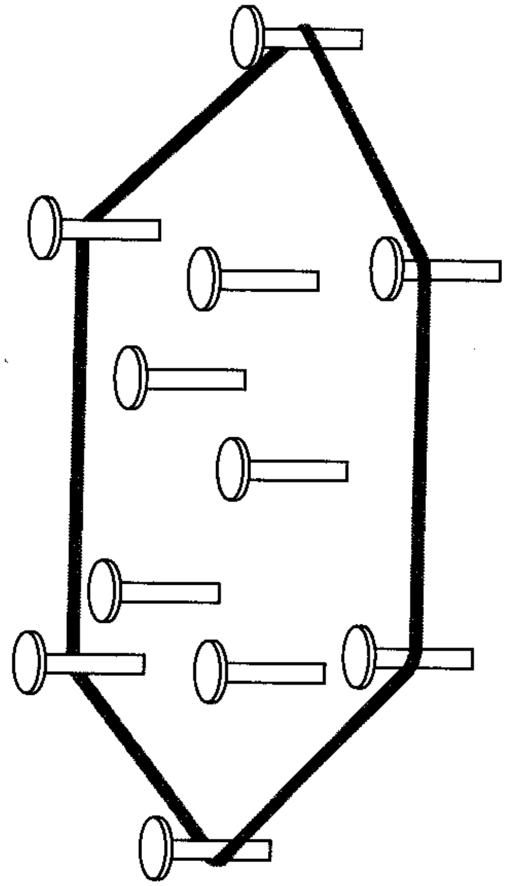
- Tason pistejoukon Q *kupera peite* (Convex Hull) on pienin sellainen monikulmio CH(Q), joka täyttää seuraavan ehdon:

  > Otammepa mitkä tahansa kaksi pistettä $p_1, p_2 \in Q$, niin niiden välinen jana $\overline{p_1p_2}$ ei astu ulos monikulmiosta CH(Q).

- Siis *aitaamme* joukon Q mahdollisimman tiukasti [6, kuva 3.5].

- Silloin tämä aita on ulkopuolelta katsoen kupera.

- Ongelmana on tunnistaa joukosta Q tämän aidan *kulmapisteet* (kiertojärjestyksessä myötä- tai vastapäivään).



- Tähän ongelmaan on *raakaan (laskenta)voimaan* (Brute Force) perustuva algoritmi [6, luku 3.3] joka perustuu seuraavaan oivallukseen:

  Olkoot $p_1, p_2 \in Q$. Niiden välinen jana $\overline{p_1 p_2}$ kuuluu monikulmion CH(Q) reunaan täsmälleen silloin kun kaikki muut jokon Q pisteet ovat janan samalla puolella (tai rajatapauksessa janan kautta kulkevalla suoralla).

  Tämäkin ehto voidaan testata kalvojen 12.1 menetelmillä.

- Saadaan

  $$O(|Q|^3) \qquad \text{askeleessa}$$

  toimiva algoritmi:

  Käy läpi kaikki pisteparit $p, p' \in Q$ ja testaa jokaiselle kaikki muut pisteet $p'' \in Q$.

- Tehokkaampia menetelmiä ovat esimerkiksi

  Jarvisin marssi (Jarvis' march) eli "käärepaperimenetelmä" [1, luku 33.3] joka vie

  $$O(|Q| \cdot |\mathrm{CH}(Q)|) \qquad \text{askelta}$$

  Pikapeite (quickhull) [6, luku 4.6]

  Grahamin pyyhkäisy (Graham's scan) [1, luku 33.3] jotka vievät

  $$O(|Q| \cdot \log_2(|Q|)) \qquad \text{askelta.} \qquad (12)$$

- Esitellään Grahamin pyyhkäisy tarkemmin.

1. Etsitään sellainen alkupiste $p_0 \in Q$, joka on varmasti yksi kysytyn monikulmion CH(Q) kulmapisteistä.
   - Muut pisteet ovat tästä alkupisteestä p_0 katsoen samalla puolitasolla, eli nähtävissä yhdellä silmäyksellä.
   - Riittää $p_0 =$ alimmista pisteistä vasemmanpuoleisin.

2. *Järjestetään* muut syötepisteet siten, kuin ne näkyvät oikealta vasemmalle alkupisteestä p_0 katsottaessa.
   - Eli CH(Q) kierretään vastapäivään.
   - Käytetään vertailuoperaatiossa kalvojen 12.1 havaintoja.
   - Aikavaatimus (12) voidaan saavuttaa nopealla järjestämisalgoritmilla.

     (Suositus: kalvojen 3.2 pikajärjestäminen.)
3. Jos alkupisteestä p_0 katsottaessa samalle suoralle sattuu monta eri pistettä, niin
   - säilytetään niistä vain *kaukaisin* ja unohdetaan muut
   - koska ne eivät voi olla monikulmion CH(Q) kulmapisteitä.

   Tämä on helppoa nyt kun pisteet on järjestetty askeleessa 2.

4. Olkoot jäljellä olevat pisteet järjestyksessä
$$p_1, p_2, p_3, \ldots, p_m \in Q.$$
Käydään ne läpi tässä järjestyksessä.

[1, kuva 33.7].

- Käytetään työ*pino*a kalvoilta 2.1 kehittyvän CH(Q) tallentamiseen.
- Työpino alustetaan pisteillä

(a) p_0 pohjalle

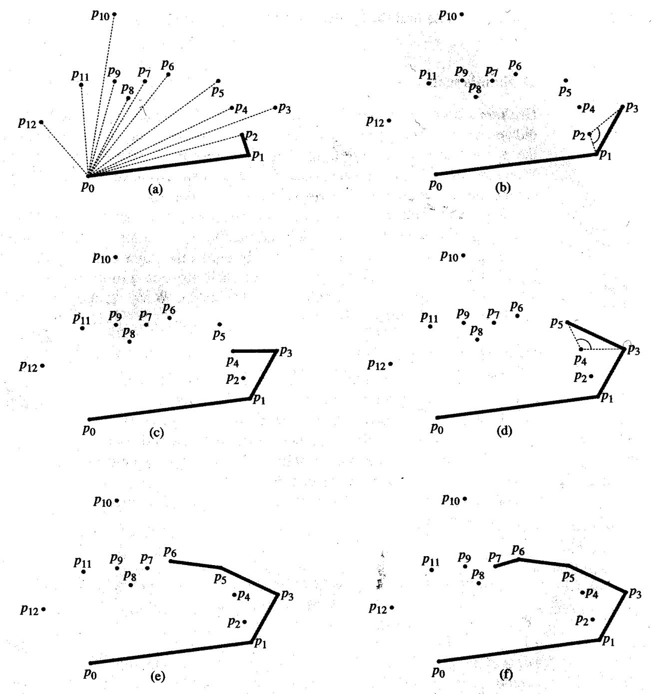
(b) p_1 keskelle

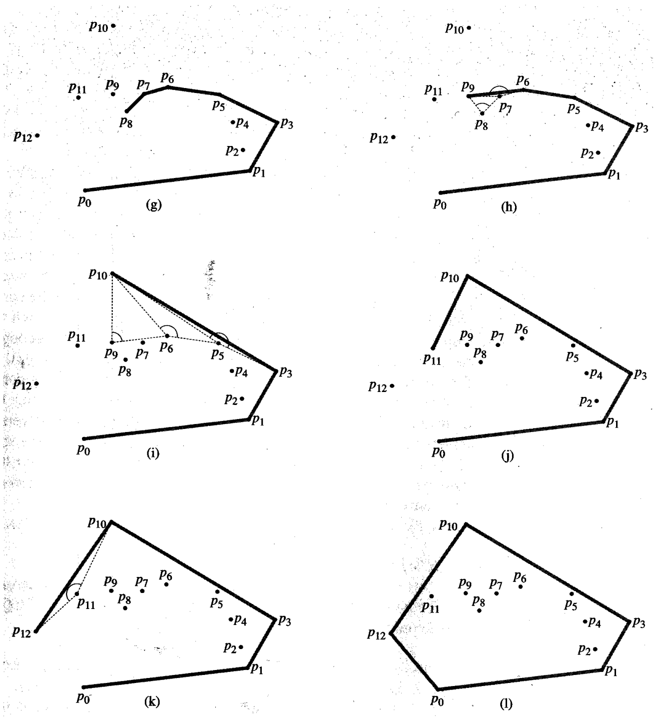
(c) p_2 päällimmäiseksi.

(Jos $m < 2$ niin CH(Q) on surkastunut janaksi $\overline{p_0 p_1}$ tai pisteeksi p_0.)

5. Seuraava piste p_i voidaan viedä pinoon vasta kun pätee ehto:

- janojen $\overline{p''p'}$ ja $\overline{p'p_i}$ pitää kääntyä aidosti vasemmalle.
- missä p' on työpinon ylin ja p'' toiseksi ylin piste.





- Ehdon 5 tutkimiseen käytetään kalvojen 12.1 havaintoja.
- Työpinon päältä poistetaan pisteitä kunnes ehto 5 saadaan pätemään.
- Työpino ei voi tulla liian tyhjäksi, koska konstruktion nojalla $\overline{p_0 p_1}$ on monikulmion CH(Q) yksi reuna.
- Kun pistettä p_i aletaan käsitellä, työpino sisältää täsmälleen monikulmion CH($\{p_0, p_1, p_2, \ldots, p_{i-1}\}$) pisteet.
- Näin ollen
  - kun viimeinen piste p_m on lisätty työpinoon
  - niin työpino on täsmälleen vastaukseksi haluttu aita CH(Q)
  - vastapäivään kierrettynä
  - eli myötäpivään luettavissa.

12.3 Monikulmion kolmiointi

- Annettu (itseään leikkaamaton) tason monikulmio P täytyy usein *kolmioida:*
  - vetää uusia viivoja monikulmion P kärkipisteiden välille
  - siten että monikulmion P sisään jäävä alue jakautuu kolmioiksi
  - ilman uusien kärkipisteiden syntymistä alueen sisälle.
- Esimerkiksi jos täytyy laskea monikulmion P pinta-ala.
- Helppoa, jos P on kupera kalvojen 12.2 mielessä:

  Piirretään yhdestä kärjestä viuhka muihin.
- Hahmotellaan yleiseen tapaukseen yksinkertainen *Van Gogh*in algoritmi.

  [8, luku 14.4.1]

- Nopeampiakin algoritmeja tunnetaan.
- Sovitaan monikulmion P reunan *kulkusuunnaksi* vastapäivään.

  (Myötäpäiväänkin voi toki kulkea.)
- Silloin jokaisella kärkipisteellä v on kulkusuunnassa
  - edellinen$[v]$
  - seuraava$[v]$

  kärkipiste.
- Monikulmion P kärkipiste v on *korvannipukka*, jos kolmio

  $$\text{korva}[v] = \langle \text{edellinen}[v], v, \text{seuraava}[v] \rangle$$

  on kokonaan monikulmion P sisällä.
- Fakta: Jos P on monimutkaisempi kuin kolmio, niin sillä on korvannipukka.

  (Jopa vähintään kaksi.)

- Korvannipukka v voidaan *leikata pois* monikulmiosta P

  poistamalla v ja siihen liittyvät reunaviivat

  $$\overline{\text{edellinen}[v]\ v} \quad \text{ja} \quad \overline{v\ \text{seuraava}[v]} \qquad (13)$$

  lisäämällä niiden tilalle uusi reunaviiva

  $$\overline{\text{edellinen}[v]\ \text{seuraava}[v]}$$

  ilmoittamalla korva$[v]$ kolmioinnin yhdeksi kolmioksi.

u := monikulmion P jokin kärkipiste;
while P on monimutkaisempi kuin kolmio **do**
  repeat
    v := u;
    u := seuraava$[v]$
  until v on korvannipukka;
  Leikkaa v pois
end while;
Tulosta kolmio P.

- Korvannipukan tunnistaminen:
  - Peräkkäisten janojen (13) pitää olla kiertymättä oikealle.

    Eli kärkipiste v ei saa "painua sisäänpäin" monikulmion P reunalla.
  - Monikulmion muiden pisteiden w pitää pysytellä aidosti kolmion korva$[v]$ ulkopuolella.

    Eli mikään piste w ei saa olla yhtä aikaa kaikkien viivojen (13) ja

    $$\overline{\text{seuraava}[v]\ \text{edellinen}[v]}$$

    vasemmalla puolella (eikä viivoilla).
  - Kaikki nämä testit voidaan tehdä kalvojen 12.1 keinoin.

12.4 Pyyhkäisymenetelmä

- *Pyyhkäisy*llä (sweep) [1, luku 33.2] voi kehittää geometrisia algoritmeja
- Ideana on *suora joka pyyhkäisee yli kuvion* [1, kuva 33.5].
- Pyyhkäisyn aikana pidetään yllä tietoa

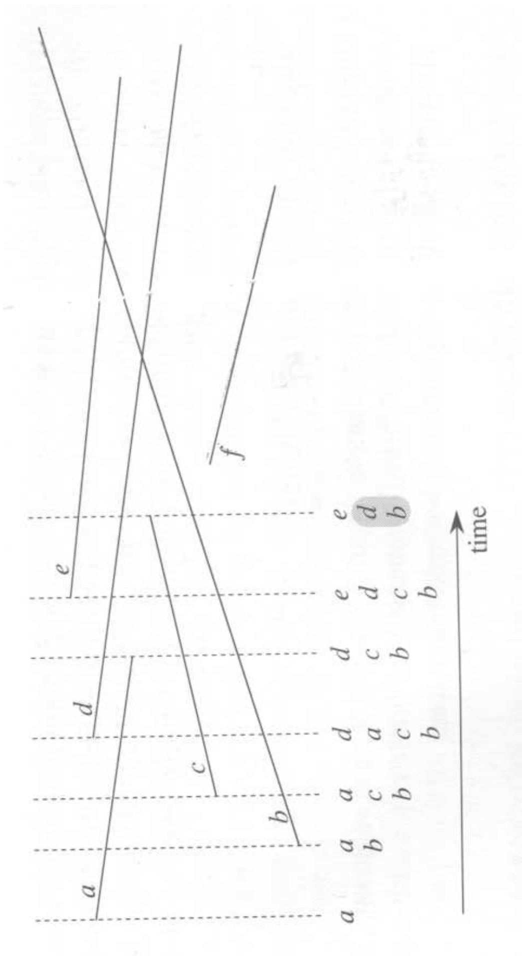
  pyyhkäisyviivan tilasta (sweep-line status)

  Se osa kuviosta, jonka läpi pyyhkäisevä suora tällä hetkellä kulkee.

  tapahtumien aikataulusta (event-point schedule)

  Se järjestys, jossa pyyhkäisyviiva kohtaa sellaiset kuvion pisteet, joissa tapahtuu jotakin "mielenkiintoista".

- Menetelmä etenee **aikataulu**n mukaan ja päivittää **tila**a joka tapahtuman kohdalla.
- 3-ulotteisissa ongelmissa käytetään pyyhkäisy*taso*a.



12.4.1 Esimerkki: Legokaukalo

- Annetaan $M \times N \times 1$ yksikön kokoinen pohjalevy.
- Levyn jokaiseen ruutuun $1 \leq i \leq M, 1 \leq j \leq N$ on pinottu torniksi korkeus$[i][j]$ kappaletta $1 \times 1 \times 1$ yksikön kokoisia kuutioita.

  Siis kootaan Lego-palikoista "linna" alustalle.
- Kuutioiden väliset saumat ovat vesitiiviitä.
- Kun rakennelma
  1. upotetaan kokonaan veteen pohjalevy edellä
  2. nostetaan samassa asennossa ylös vedestä

  niin montako yksikköä vettä jää rakennelman sisään?

- Seurataan, miten kuiva alue kutistuu sitä mukaa kun rakennelma uppoaa syvemmälle:

  Pyyhkäisytasona toimii vedenpinta rakennelmaa upotettaessa veteen.

  Tason tilana toimii se muuri W, joka erottaa märän ulko-osan ja kuivan sisäosan.

  Eli ne tornit i, j joiden jollakin sivulla on jo märkää mutta yläpinta on yhä kuiva.

  Aikatauluna toimivat ne upotussyvyydet, joilla muuri W muuttuu.

  Eli kun vettä pääsee vuotamaan sisään muurin W alimmasta aukosta.

  Eli muurissa W olevat tornit i, j kasvavassa järjestyksessä niiden korkeuden suhteen.

- Lasketaan luvut $d[i][j] =$ se (pohjalevyn yläreunasta mitattu) upotussyvyys, jonka jälkeen torni i, j on veden alla.

- Kun rakennelmaa nostetaan takaisin vedestä, ruudusta i, j valuu vettä pois, kunnes upotussyvyys on jälleen $d[i][j]$:

  Matalin reitti veden virrata ruutuun i, j on kääntäen matalin reitti sen virrata pois.

- Ruutuun i, j jää siis

  $$\text{tilavuus}[i][j] = d[i][j] - \text{korkeus}[i][j]$$

  yksikköä vettä.

  Kysytty tulos on siis näiden lukujen summa

  $$\sum_{\substack{1<i<M \\ 1<j<N}} \text{tilavuus}[i][j].$$

- Otetaan siis seuraava muurin W aukko i, j ja kastellaan kaikki ne kuivat ruudut p, q joihin tästä aukosta virtaa vettä.

Aluksi vedenpinta ulottuu pohjalevyyn mutta ei kuutioihin. Silloin jokainen upotussyvyys $d[i][j]$ on määrittelemätön ja muuri W koostuu kaikista reunaruuduista;
while $W \neq \emptyset$ **do**
  Poista muurista W sellainen ruutu i, j
  jonka korkeus$[i][j]$ on pienin;
  kastele(i, j)
end while;
return $\sum_{\substack{1<i<M \\ 1<j<N}} d[i][j] -$ korkeus$[i][j]$.

procedure kastele(p, q: **indeksi**) **is**
  if $d[p][q]$ on määrittelemätön **then**
    if korkeus$[p][q] >$ korkeus$[i][j]$ **then**
      Lisää ruutu p, q muuriin W (ellei se ole jo siellä)
    else
      $d[p][q]$:= korkeus$[i][j]$;
      for all ruudun p, q naapuri r, s **do**
        kastele(r, s)
      end for
    end if
  end if.

- Tässä tehtävässä jatkoaikataulu selviää vasta pyyhkäisyn aikana.

- Muuri W voidaan toteuttaa kalvojen 2.3 kekona.

- Jokaiseen ruutuun liitetään vielä kenttä, joka kertoo, kuuluuko se muuriin W.

- Aliohjelma kastele(i, j) tekee oleellisesti kalvojen 8.4 syvyyssuuntaisen läpikäynnin ruudun i, j naapurustossa.

  Läpikäynti jatkuu lähtötornia i, j korkeampiin torneihin saakka.

- Algoritmi vie silloin

  $$O(I \cdot \log_2(I)) \qquad \text{askelta}$$

  missä $I = M \cdot N$ on syötteen koko.

12.4.2 Leikkaavatko janat?

- Leikkaavien janojen ongelmassa [1, luku 33.2] annetaan syötteenä n janaa

  $$\overline{p_1q_1}, \overline{p_2q_2}, \overline{p_3q_3}, \ldots, \overline{p_nq_n}$$

  (eli $2n$ alku- ja loppupistettä p_i ja q_j, eli $4n$ koordinaattilukua) ja kysytään onko niiden joukossa kaksi jotka leikkaavat toisiaan.

- Kalvoilla 12.1 nähtiin, miten ristituloilla voitiin selvittää $O(1)$ askeleessa leikkaavatko kaksi janaa.

  Silloin on yksinkertainen

  $$O(n^2)$$

  askeleen algoritmi: testaa jokaista janaa kaikkiin sen jälkeen tuleviin janoihin.

- Hahmotellaan tälle ongelmalle tehokkaampaa algoritmia pyyhkäisymenetelmällä.

  Tässä tehtävässä aikataulun voi muodostaa etukäteen *järjestämällä syöte* sopivasti.

- Yksinkertaistavia oletuksia:
  - Mikään janoista ei ole pystysuora.

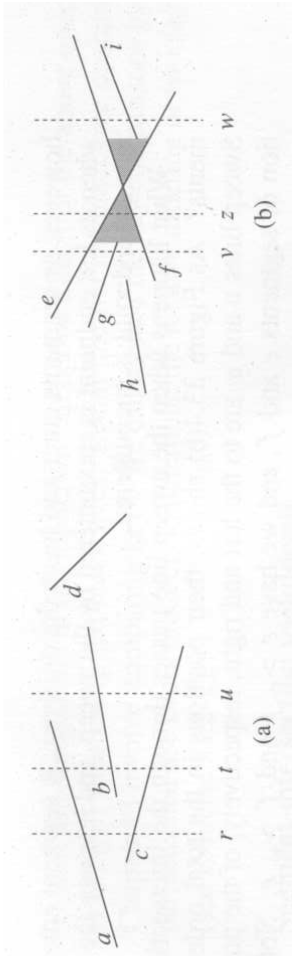
    Silloin pyyhkäisyviiva leikkaa janan korkeintaan yhdessä pisteessä.

    Jana alkaa vasemmasta päätepisteestään ja päättyy oikeaan.

    Viivan tila voi säilyttää leikkaamansa janat leikkauspisteiden mukaisessa *järjestyksessä* [1, kuva 33.4 (a)].
  - Samassa pisteessä leikkaa vain kaksi janaa.

    Viivan ohittaessa kahden janan leikkauspisteen ne *vaihtavat paikkaa* tässä järjestyksessä [1, kuva 33.4 (b)].

    Oletuksen nojalla ne ovat sillä hetkellä *naapureita* järjestyksessä.
- Oletukset voidaan poistaa, mutta algoritmi mutkistuu.



- **Aikataulu** muodostetaan *etukäteen lajittelemalla* syötejanojen alku- ja päätepisteet sopivasti:

  Piste $p = \langle x, y\rangle$ on ennen eri pistettä $p' = \langle x', y'\rangle$ jos
  - $x < x'$
  - $x = x'$ ja p alkaa mutta p' päättää janan.
  - $x = x'$, sekä p että q' ovat alkavia tai päättäviä, mutta $y < y'$.
- Voidaan tehdä
$$O(n \cdot \log n)$$
  askeleessa

  (vaikkapa kalvojen 3.2 pikajärjestämisellä).

- Jana s [1, kuva 33.5]

  lisätään viivan tilaan kun aikataulussa on päästy sen alkupisteeseen.

  Silloin kysytään janan s naapur(e)ista s' järjestyksessä leikkaavatko s ja s'.

  poistetaan kun loppupisteeseen.

  Jos s oli naapureiden s' ja s'' välissä järjestyksessä, niin kysytään leikkaavatko s' ja s''.
- Jos leikkaavat, niin algoritmi vastaa heti "kyllä".

  Jos taas päästään aikataulun loppuun, niin "ei".

- Järjestyksen ylläpitoon tarvitaan siis tietorakenne jossa
  - lisäys
  - poisto
  - naapuri(e)n haku

  vie vain

  $$O(\log n)$$

  askelta.

- Sellaisia on [1, luvut 13 ja 18; 6, luku 6.3]:
  - – Ne ovat liian vaikeita kilpailutilanteessa käsin ohjelmoitaviksi.
  - + Sellaisen sopiva toteutus voi löytyä valmiina tietorakennekirjastosta.