14 Lyhimmät painotetut polut

BFS löytää lyhimmän polun lähtösolmusta graafin saavutettaviin solmuihin.

Se ei kuitenkaan enää suoriudu tehtävästä, jos kaarien läpi kulkeminen maksaa askelta enemmän.

Tässä luvussa käsitellemme lyhimpien painotettujen polkujen etsintää graafista, jonka kaaripainot ovat positiivisia ja voivat poiketa ykkösestä.

- negatiivisten kaaripainojen hallitsemiseen tarvitaan monimutkaisempia algoritmeja, esimerkiksi Bellman-Ford algoritmi

14.1 Lyhin polku

Graafin kaarilla voi olla ominaisuus nimeltä *paino(weight)*.

- paino voi edustaa vaikkapa reitin pituutta tai tilasiirtymän kustannusta
- Graafin $G = (V, E)$ painofunktio $w : E \to \mathbb{R}$ kaarilta reaalilukupainoille
- Polun $p = \langle v_0, v_1, ..., v_k \rangle$ paino $w(p)$ on sen muodostavien kaarten painojen summa $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Määritelmä: lyhimmän polun paino $\delta(u, v)$:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{jos on polku} u\text{:sta } v\text{:hen,} \\ \infty & \text{muuten.} \end{cases}$$

ja siten lyhin polku u:sta v:hen mikä tahansa polku p, jolle $w(p) = \delta(u, v)$

Tämä mutkistaa lyhimmän reitin etsintää merkittävästi.

- lyhin reitti on se lähtösolmusta etsittyyn solmuun kulkeva polku, jonka kaarien painojen summa on mahdollisimman pieni
- jos jokaisen kaaren paino on 1, tehtävä voidaan ratkaista käymällä lähtösolmusta saavutettavissa oleva graafin osa läpi leveyteen ensin -järjestyksessä
- jos painot saattavat olla < 0, voi olla, että tehtävään ei ole ratkaisua, vaikka polkuja olisi olemassakin
  - jos graafissa on silmukka, jonka kaaripainojen summa on negatiivinen saadaan mielivaltaisen pieni painojen summa kiertämällä silmukkaa tarpeeksi monta kertaa

14.2 Dijkstran algoritmi

Suunnatun, painotetun graafin $G = (V, E)$, jossa kaaripainot ovat ei-negatiivisia, lyhimmät painotetut polut lähtösolmusta voi etsiä Dijkstran algoritmilla.

- etsii lyhimmät polut lähtösolmusta s kaikkiin saavutettaviin solmuihin, painottaen kaarien pituuksia w:n mukaan
- valitsee joka tilanteessa tutkittavakseen lyhimmän polun, jota se ei ole vielä tutkinut

  $\Rightarrow$ se on siis ahne algoritmi
- oletus: $w(u, v) \geq 0 \; \forall (u, v) \in E$

```
DIJKSTRA(s, w)                          (algoritmi saa parametrinaan aloitussolmun s)
1   ▷ alussa kaikkien solmujen kentät ovat arvoiltaan colour = WHITE, d = ∞, π = NIL
2   s→colour := GRAY                    (merkitään alkutila löydetyksi)
3   s→d := 0                            (etäisyys alkutilasta alkutilaan on 0)
4   PUSH(Q, s)                          (työnnetään alkutila prioriteettijonoon)
5   while Q ≠ ∅ do                      (jatketaan niin kauan kun solmuja riittää)
6       u := EXTRACT-MIN(Q)             (otetaan prioriteettijonosta seuraava tila)
7       for each v ∈ u→Adj do           (käydään u:n naapurit läpi)
8           if v→colour = WHITE then    (jos solmussa ei ole käyty . . . )
9               v→colour := GRAY        (. . . merkitään se löydetyksi)
10              PUSH(Q, v)              (työnnetään tila jonoon odottamaan käsittelyä)
11          RELAX(u, v, w)
12      u→colour := BLACK               (merkitään tila u käsitellyksi)

RELAX(u, v, w)
1   if v→d > u→d + w(u, v) then         (jos löydettiin uusi lyhyempi reitti tilaan v...)
2       v→d := u→d + w(u, v)            (...pienennetään v:n etäisyyttä lähtösolmusta)
3       v→π := u                        (merkitään, että v:n tultiin u:sta)
```

Algoritmin käyttämä tietorakenne Q on prioriteettijono (luentomonisteen kohta 3.2).

w sisältää kaikkien kaarien painot.

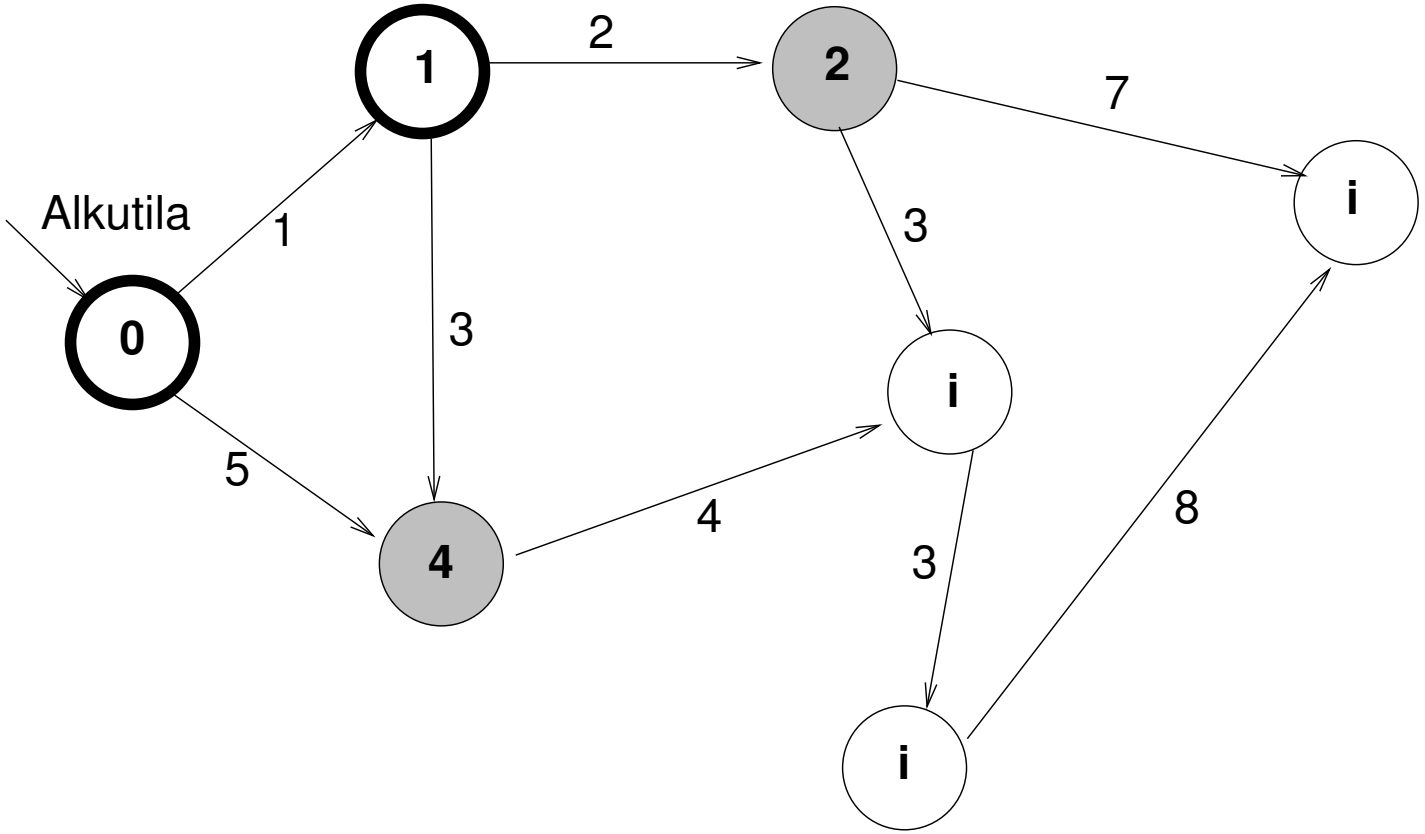
Dijkstran algoritmi käyttää apufunktiota RELAX

- kaaren (u, v) *relaksointi* (*relaxation*) testaa, voiko lyhintä löydettyä v:hen vievää polkua parantaa reitittämällä sen loppupää u:n kautta, ja tarvittaessa tekee niin

Muilta osin algoritmi muistuttaa huomattavasti leveyteen ensin -hakua.

- se löytää lyhimmät polut kasvavan pituuden mukaisessa järjestyksessä
- kun solmu u otetaan Q:sta, sen painotettu etäisyys s:stä varmistuu $u \rightarrow d$:ksi
  - jos prioriteettijonosta otettu tila on maalitila, voidaan algoritmin suoritus lopettaa saman tien

Alla olevassa kuvassa nähdään Dijkstran algoritmi tilanteessa, jossa mustalla ympyröidyt solmut on käsitelty.



Suoritusaika:

- while-silmukka käy enintään $O(V)$ kierrosta ja for-silmukka yhteensä enintään $O(E)$ kierrosta
- prioriteettijonon voi toteuttaa tehokkaasti keon avulla tai vähemmän tehokkaasti listan avulla

	kekototeutuksella:	listatoteutuksella:	
rivi 4:	$\Theta(1)$	$\Theta(1)$	(tyhjään tietorakenteeseen lisääminen)
rivi 5:	$\Theta(1)$	$\Theta(1)$	(onko prioriteettijono tyhjä)
rivi 6:	$O(\lg V)$	$O(V)$	(Extract-Min)
rivi 10:	$\Theta(1)$	$\Theta(1)$	(valkoisen solmun prioriteetti on ääretön, joten sen oikea paikka on keon lopussa)
rivi 11:	$O(\lg V)$	$\Theta(1)$	(relaksoinnissa solmun prioriteetti voi muuttua)

- käytettäessä kekototeutusta jokaisella while- ja for-silmukan kierroksella suoritetaan yksi $O(\lg V)$ aikaa kuluttava operaatio

  $\Rightarrow$ algoritmin suoritusaika on $O((V + E) \lg V)$

14.3 A*-algoritmi

Dijkstran algoritmi etsi lyhimmän painotetun reitin kartoittamalla solmuja lyhimmästä reitistä alkaen reitin pituusjärjestyksessä. Eli: Dijkstra käyttää hyväkseen vain jo kuljetuista kaarista saatavaa tietoa.

A*-algoritmi tehostaa tätä lisäämällä *heuristiikan* (=oletuksen) lyhimmästä mahdollisesta etäisyydestä maaliin. (Esim. maantiereitin haussa etäisyys linnuntietä).

- etsii lyhimmän painotetun polun lähtösolmusta s annettuun maalisolmuun g. **Ei** kartoita lyhintä reittiä kaikkiin solmuihin (kuten Dijkstra), vain maalisolmuun.
- edellyttää, että painot ovat ei-negatiivisia (kuten Dijkstrakin)
- edellyttää, että jokaiselle solmulle voidaan laskea sen minimietäisyys maalista (ts. löytynyt lyhin reitti ei voi olla lyhempi).
- valitsee joka tilanteessa tutkittavakseen ei-tutkitun solmun, jossa (lyhin etäisyys lähdöstä solmuun + arvioitu minimietäisyys maaliin) on pienin.

Ainoa ero A*:n ja Dijkstran välillä on relaksointi (ja se, että A* kannattaa lopettaa heti maalisolmun löydyttyä, koska se ei kartoita lyhimpiä etäisyyksiä kaikkiin solmuihin).

RELAX-A*(u, v, w)

```
1  if v→d > u→d + w(u, v) then     (jos löydettiin uusi lyhyempi reitti tilaan v...)
2       v→d := u→d + w(u, v)       (...uusi pituus tähän saakka...)
3       v→de := v→d + min_est(v, g)  (...ja minimiarvio koko reitistä)
4       v→π := u                   (merkitään, että v:n tultiin u:sta)
```

A*:n käyttämässä prioriteettijonossa käytetään prioriteettina koko reitin pituusarviota $v{\rightarrow}de$.

(Dijkstran algoritmi on A*:n erikoistapaus, jossa $min\_est(a, b)$ on aina 0.)

14.4 Kevyin virittävä puu

Graafin $G = (V, E)$ kevyin virittävä puu on sen asyklinen aligraafi, joka yhdistää kaikki graafin solmut niin, että aligraafin kaarien painojen summa on pienin mahdollinen.

Puun löytämiseksi on kaksi algoritmia: Primin ja Kruskalin

Prim muistuttaa Dijkstran algoritmin kun taas Kruskal lähestyy ongelmaa luomalla metsän, jossa on puu jokaiselle puulle ja sitten yhdistämällä näitä puuksi