# Kisakoodarin käsikirja

Antti Laaksonen

5. marraskuuta 2016

# Sisältö

# Osa I

# Perusasiat

# Osa II

# Verkkoalgoritmit

# Luku 13

# Lyhimmät polut

Lyhimmän polun etsiminen alkusolmusta loppusolmuun on keskeinen verkko-ongelma, joka esiintyy usein käytännön tilanteissa. Esimerkiksi tieverkostossa tyypillinen ongelma on selvittää, mikä on lyhin reitti kahden kaupungin välillä, kun tiedossa ovat kaupunkien väliset tiet ja niiden pituudet.

Jos verkon kaarilla ei ole painoja, polun pituus on sama kuin kaarten määrä polulla, jolloin lyhimmän polun voi etsiä leveyshaulla. Tässä luvussa keskitymme kuitenkin tapaukseen, jossa kaarilla on painot. Tällöin lyhimpien polkujen etsimiseen tarvitaan kehittyneempiä algoritmeja.

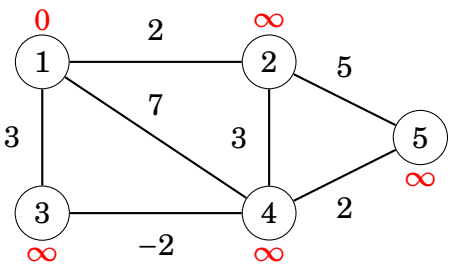
## 13.1 Bellman-Fordin algoritmi

Bellman-Fordin algoritmi etsii lyhimmän polun alkusolmusta kaikkiin muihin verkon solmuihin. Algoritmi toimii kaikenlaisissa verkoissa, kunhan verkossa ei ole sykliä, jonka kaarten yhteispaino on negatiivinen. Jos verkossa on negatiivinen sykli, algoritmi huomaa tilanteen.

Algoritmi pitää yllä etäisyysarvioita alkusolmusta kaikkiin muihin verkon solmuihin. Alussa alkusolmun etäisyysarvio on 0 ja muiden solmujen etäisyysarvio on ääretön. Algoritmi parantaa arvioita etsimällä verkosta kaaria, jotka lyhentävät polkuja, kunnes mitään arviota ei voi enää parantaa.

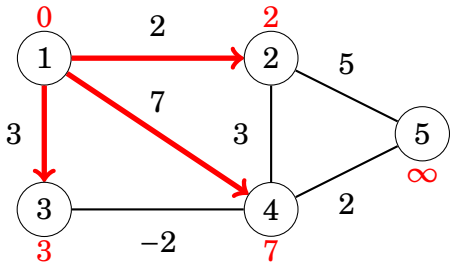
### Toiminta

Tarkastellaan Bellman-Fordin algoritmin toimintaa seuraavassa verkossa:

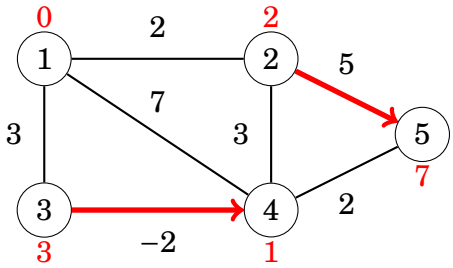



Verkon jokaiseen solmun viereen on merkitty etäisyysarvio. Alussa alkusolmun etäisyysarvio on 0 ja muiden solmujen etäisyysarvio on ääretön ($\infty$).

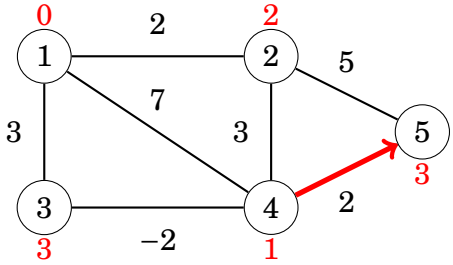
Algoritmin etsii verkosta kaaria, jotka parantavat etäisyysarvioita. Aluksi kaikki solmusta 0 lähtevät kaaret parantavat arvioita:




Sitten kaaret $2 \to 5$ ja $3 \to 4$ parantavat arvioita:

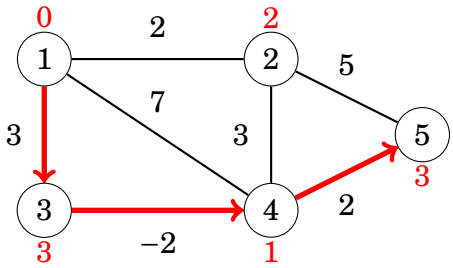



Lopuksi tulee vielä yksi parannus:




Tämän jälkeen mikään kaari ei paranna etäisyysarvioita. Tämä tarkoittaa, että etäisyydet ovat lopulliset, eli joka solmussa on nyt pienin etäisyys alkusolmusta kyseiseen solmuun.

Esimerkiksi pienin etäisyys 3 solmusta 1 solmuun 5 toteutuu käyttämällä seuraavaa reittiä:



### Toteutus

Seuraava Bellman-Fordin algoritmin toteutus etsii lyhimmät polut solmusta $x$ kaikkiin muihin verkon solmuihin. Koodi olettaa, että verkko on tallennettuna vieruslistoina taulukossa

```
vector<pair<int,int>> v[N];
```

niin, että parissa on ensin kaaren kohdesolmu ja sitten kaaren paino.

Algoritmi muodostuu $n-1$ kierroksesta, joista jokaisella algoritmi käy läpi kaikki verkon kaaret ja koettaa parantaa etäisyysarvioita Algoritmi laskee taulukkoon e etäisyyden solmusta $x$ kuhunkin verkon solmuun. Koodissa oleva alkuarvo $10^9$ kuvastaa ääretöntä.

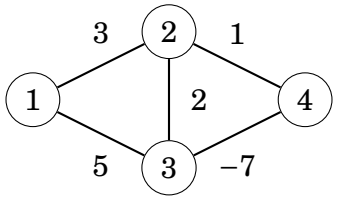
```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (int a = 1; a <= n; a++) {
        for (auto b : v[a]) {
            e[b.first] = min(e[b.first],e[a]+b.second);
        }
    }
}
```

Algoritmin aikavaativuus on $O(nm)$, koska se muodostuu $n-1$ kierroksesta ja käy läpi jokaisen kierroksen aikana kaikki $m$ kaarta. Jos verkossa ei ole negatiivista sykliä, kaikki etäisyysarviot ovat lopulliset $n-1$ kierroksen jälkeen, koska jokaisessa lyhimmässä polussa on enintään $n-1$ kaarta.

Käytännössä kaikki lopulliset etäisyysarviot saadaan usein laskettua selvästi alle $n-1$ kierroksessa, joten hyvä tehostus algoritmiin on lopettaa heti, kun mikään etäisyysarvio ei parane kierroksen aikana.

### Negatiivinen sykli

Bellman-Fordin algoritmin avulla voi myös tarkastaa, onko verkossa sykliä, jonka pituus on negatiivinen. Esimerkiksi verkossa




on negatiivinen sykli $2 \to 3 \to 4 \to 2$, jonka pituus on $-4$.

Jos verkossa on negatiivinen sykli, sen kautta kulkevaa polkua voi lyhentää äärettömästi toistamalla negatiivista sykliä uudestaan ja uudestaan, minkä vuoksi lyhimmän polun käsite ei ole mielekäs.

Negatiivisen syklin voi tunnistaa Bellman-Fordin algoritmilla suorittamalla algoritmia $n$ kierrosta. Jos viimeinen kierros parantaa jotain etäisyysarviota,

verkossa on negatiivinen sykli. Huomaa, että tässä mikään solmuista ei ole alkusolmu ja algoritmi etsii negatiivista sykliä koko verkon alueelta.

### SPFA-algoritmi

SPFA-algoritmi (*Shortest Path Faster Algorithm*) on Bellman-Fordin algoritmin muunnelma, joka on usein alkuperäistä algoritmia tehokkaampi. Se ei tutki joka kierroksella koko verkkoa läpi parantaakseen etäisyysarvioita, vaan valitsee tutkittavat kaaret älykkäämmin.

Algoritmi pitää yllä jonoa solmuista, joiden kautta saattaa pystyä parantamaan etäisyysarvioita. Algoritmi lisää jonoon aluksi aloitussolmun $x$ ja valitsee aina seuraavan tutkittavan solmun $a$ jonon alusta. Aina kun kaari $a \to b$ parantaa etäisyysarviota, algoritmi lisää jonoon solmun $b$.

Seuraavassa toteutuksessa jonona on queue-rakenne q. Lisäksi taulukko z kertoo, onko solmu valmiina jonossa, jolloin algoritmi ei lisää solmua jonoon uudestaan.

```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
q.push(x);
while (!q.empty()) {
    int a = q.front(); q.pop();
    z[a] = 0;
    for (auto b : v[a]) {
        if (e[a]+b.second < e[b.first]) {
            e[b.first] = e[a]+b.second;
            if (!z[b]) {q.push(b); z[b] = 1;}
        }
    }
}
```

SPFA-algoritmin tehokkuus riippuu verkon rakenteesta: algoritmi on keskimäärin hyvin tehokas mutta sen pahimman tapauksen aikavaativuus on edelleen $O(nm)$ ja on mahdollista laatia syötteitä, jotka saavat algoritmin yhtä hitaaksi kuin tavallisen Bellman-Fordin algoritmin.

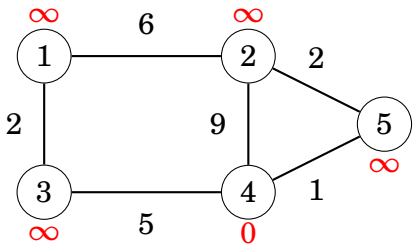
## 13.2 Dijkstran algoritmi

Dijkstran algoritmi etsii Bellman-Fordin algoritmin tavoin lyhimmät polut alkusolmusta kaikkiin muihin solmuihin. Dijkstran algoritmi on tehokkaampi kuin Bellman-Fordin algoritmi, minkä ansiosta se soveltuu suurten verkkojen käsittelyyn. Algoritmi vaatii kuitenkin, ettei verkossa ole negatiivisia kaaria.

Dijkstran algoritmi vastaa Bellman-Fordin algoritmia siinä, että se pitää yllä etäisyysarvioita solmuihin ja parantaa niitä algoritmin aikana. Algoritmin tehokkuus perustuu siihen, että sen riittää käydä läpi verkon kaaret vain kerran hyödyntäen tietoa, ettei verkossa ole negatiivisia kaaria.

### Toiminta

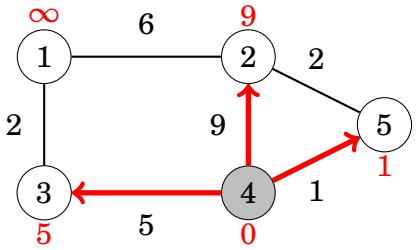
Tarkastellaan Dijkstran algoritmin toimintaa seuraavassa verkossa:




Bellman-Fordin algoritmin tavoin alkusolmun etäisyysarvio on 0 ja kaikissa muissa solmuissa etäisyysarvio on aluksi $\infty$.

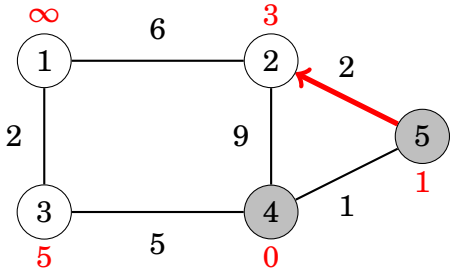
Dijkstran algoritmi ottaa joka askeleella käsittelyyn sellaisen solmun, jota ei ole vielä käsitelty ja jonka etäisyysarvio on mahdollisimman pieni. Alussa tällainen solmu on solmu 4, jonka etäisyysarvio on 0.

Kun solmu tulee käsittelyyn, algoritmi käy läpi kaikki siitä lähtevät kaaret ja parantaa etäisyysarvioita niiden avulla:

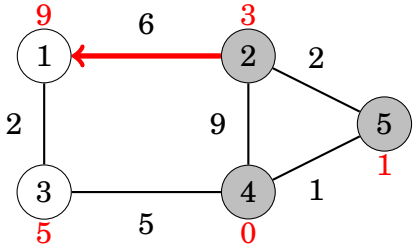



Solmun 4 käsittely paransi etäisyysarvioita solmuihin 2, 3 ja 5, joiden uudet etäisyydet ovat nyt 9, 5 ja 1.

Seuraavaksi käsittelyyn tulee solmu 5, jonka etäisyys on 1:

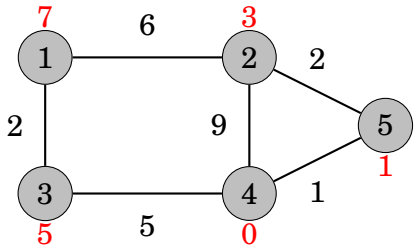



Tämän jälkeen vuorossa on solmu 2:



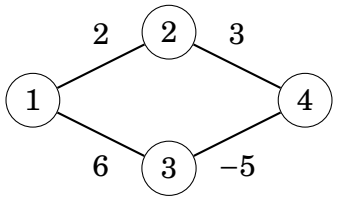
Dijkstran algoritmissa on hienoutena, että aina kun solmu tulee käsittelyyn, sen etäisyysarvio on siitä lähtien lopullinen. Esimerkiksi tässä vaiheessa etäisyydet 0, 1 ja 3 ovat lopulliset etäisyydet solmuihin 4, 5 ja 2.

Algoritmi käsittelee vastaavasti vielä kaksi viimeistä solmua, minkä jälkeen algoritmin päätteeksi etäisyydet ovat:




### Negatiiviset kaaret

Dijkstran algoritmin tehokkuus perustuu siihen, että verkossa ei ole negatiivisia kaaria. Jos verkossa on negatiivinen kaari, algoritmi ei välttämättä toimi oikein. Tarkastellaan esimerkkinä seuraavaa verkkoa:




Lyhin polku solmusta 1 solmuun 4 kulkee $1 \to 3 \to 4$, ja sen pituus on 1. Dijkstran algoritmi löytää kuitenkin keveimpiä kaaria seuraten polun $1 \to 2 \to 4$. Algoritmi ei pysty ottamaan huomioon, että alemmalla polulla kaaren paino $-5$ kumoaa aiemman suuren kaaren painon 6.

### Toteutus

Seuraava Dijkstran algoritmin toteutus laskee pienimmän etäisyyden solmusta $x$ kaikkiin muihin solmuihin. Verkko on tallennettu taulukkoon v vieruslistoina, joissa on pareina kohdesolmu ja kaaren pituus.

Dijkstran algoritmin tehokas toteutus vaatii, että verkosta pystyy löytämään nopeasti vielä käsittelemättömän solmun, jonka etäisyysarvio on pienin. Sopiva tietorakenne tähän on prioriteettijono, jossa solmut ovat järjestyksessä etäisyysarvioiden mukaan. Prioriteettijonon avulla seuraavaksi käsiteltävän solmun saa selville logaritmisessa ajassa.

Seuraavassa toteutuksessa prioriteettijono sisältää pareja, joiden ensimmäinen kenttä on etäisyysarvio ja toinen kenttä on solmun tunniste:

```
priority_queue<pair<int,int>> q;
```

Pieni hankaluus on, että Dijkstran algoritmissa täytyy saada selville pienimmän etäisyysarvion solmu, kun taas C++:n prioriteettijono antaa oletuksena

suurimman alkion. Helppo ratkaisu on tallentaa etäisyysarviot *negatiivisina*, jolloin C++:n prioriteettijonoa voi käyttää suoraan.

Koodi merkitsee taulukkoon z, onko solmu käsitelty, ja pitää yllä etäisyysarvioita taulukossa e. Alussa alkusolmun etäisyysarvio on 0 ja jokaisen muun solmun etäisyysarviona on ääretöntä vastaava $10^9$.

```
for (int i = 1; i <= n; i++) e[i] = 1e9;
e[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (z[a]) continue;
    z[a] = 1;
    for (auto b : v[a]) {
        if (e[a]+b.second < e[b]) {
            e[b] = e[a]+b.second;
            q.push({-e[b],b});
        }
    }
}
```

Yllä olevan toteutuksen aikavaativuus on $O(n+m\log n)$, koska algoritmi käy läpi kaikki verkon solmut ja lisää jokaista kaarta kohden korkeintaan yhden etäisyysarvion prioriteettijonoon. Huomaa, että $O(\log m) = O(\log n)$, koska $n$ solmun verkossa on enintään $O(n^2)$ eri kaarta ja $\log(n^2) = 2\log n$.

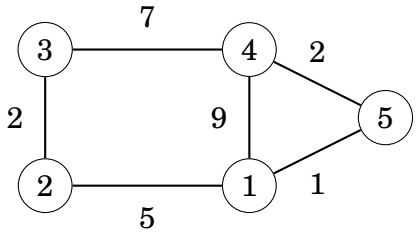
## 13.3 Floyd-Warshallin algoritmi

Floyd-Warshallin algoritmi lähestyy toisella tavalla lyhimpien polkujen etsimistä kuin Bellman-Fordin ja Dijkstran algoritmit. Siinä missä muut algoritmit etsivät lyhimpiä polkuja tietystä solmusta alkaen, Floyd-Warshallin algoritmi etsii samalla kertaa lyhimmän polun jokaisen verkon solmuparin välillä.

Algoritmi ylläpitää kaksiulotteista taulukkoa etäisyyksistä solmujen välillä. Ensin taulukkoon on merkitty etäisyydet käyttäen vain solmujen välisiä kaaria. Tämän jälkeen algoritmi päivittää etäisyyksiä, kun verkon solmut saavat yksi kerrallaan toimia välisolmuina poluilla.

### Toiminta

Tarkastellaan Floyd-Warshallin algoritmin toimintaa seuraavassa verkossa:



Algoritmi merkitsee aluksi taulukkoon etäisyyden 0 jokaisesta solmusta itseensä sekä etäisyyden $x$, jos solmuparin välillä on kaari, jonka pituus on $x$. Muiden solmuparien etäisyys on aluksi ääretön.

Tässä verkossa taulukosta tulee:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 5 | $\infty$ | 9 | 1 |
| 2 | 5 | 0 | 2 | $\infty$ | $\infty$ |
| 3 | $\infty$ | 2 | 0 | 7 | $\infty$ |
| 4 | 9 | $\infty$ | 7 | 0 | 2 |
| 5 | 1 | $\infty$ | $\infty$ | 2 | 0 |

Algoritmin toiminta muodostuu peräkkäisistä kierroksista. Jokaisella kierroksella valitaan yksi uusi solmu, joka saa toimia välisolmuna poluilla, ja algoritmi parantaa taulukon etäisyyksiä muodostaen polkuja tämän solmun avulla.

Ensimmäisellä kierroksella solmu 1 on välisolmu. Tämän ansiosta solmujen 2 ja 4 välille muodostuu polku, jonka pituus on 14, koska solmu 1 yhdistää ne toisiinsa. Vastaavasti solmut 2 ja 5 yhdistyvät polulla, jonka pituus on 6.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 5 | $\infty$ | 9 | 1 |
| 2 | 5 | 0 | 2 | **14** | **6** |
| 3 | $\infty$ | 2 | 0 | 7 | $\infty$ |
| 4 | 9 | **14** | 7 | 0 | 2 |
| 5 | 1 | **6** | $\infty$ | 2 | 0 |

Toisella kierroksella solmu 2 saa toimia välisolmuna. Tämä mahdollistaa uudet polut solmuparien 1 ja 3 sekä 3 ja 5 välille:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 5 | **7** | 9 | 1 |
| 2 | 5 | 0 | 2 | 14 | 6 |
| 3 | **7** | 2 | 0 | 7 | **8** |
| 4 | 9 | 14 | 7 | 0 | 2 |
| 5 | 1 | 6 | **8** | 2 | 0 |

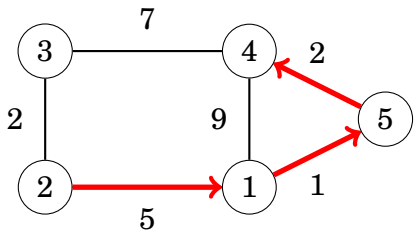
Kolmannella kierroksella solmu 3 saa toimia välisolmuna, jolloin syntyy uusi polku solmuparin 2 ja 4 välille:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 5 | 7 | 9 | 1 |
| 2 | 5 | 0 | 2 | **9** | 6 |
| 3 | 7 | 2 | 0 | 7 | 8 |
| 4 | 9 | **9** | 7 | 0 | 2 |
| 5 | 1 | 6 | 8 | 2 | 0 |

Algoritmin toiminta jatkuu samalla tavalla niin, että kukin solmu tulee vuorollaan välisolmuksi. Algoritmin päätteeksi taulukko sisältää lyhimmän etäisyyden minkä tahansa solmuparin välillä:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 5 | 7 | 3 | 1 |
| 2 | 5 | 0 | 2 | 9 | 6 |
| 3 | 7 | 2 | 0 | 7 | 8 |
| 4 | 3 | 9 | 7 | 0 | 2 |
| 5 | 1 | 6 | 8 | 2 | 0 |

Esimerkiksi taulukosta selviää, että lyhin polku solmusta 2 solmuun 4 on pituudeltaan 8. Tämä vastaa seuraavaa polkua:




### Toteutus

Floyd-Warshallin algoritmin etuna on, että se on helppoa toteuttaa. Seuraava toteutus muodostaa etäisyysmatriisin d, jossa d[$a$][$b$] on pienin etäisyys polulla solmusta $a$ solmuun $b$. Aluksi algoritmi alustaa matriisin d verkon vierusmatriisin v perusteella (arvo $10^9$ kuvastaa ääretöntä):

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) d[i][j] = 0;
        else if (v[i][j]) d[i][j] = v[i][j];
        else d[i][j] = 1e9;
    }
}
```

Tämän jälkeen lyhimmät polut löytyvät seuraavasti:

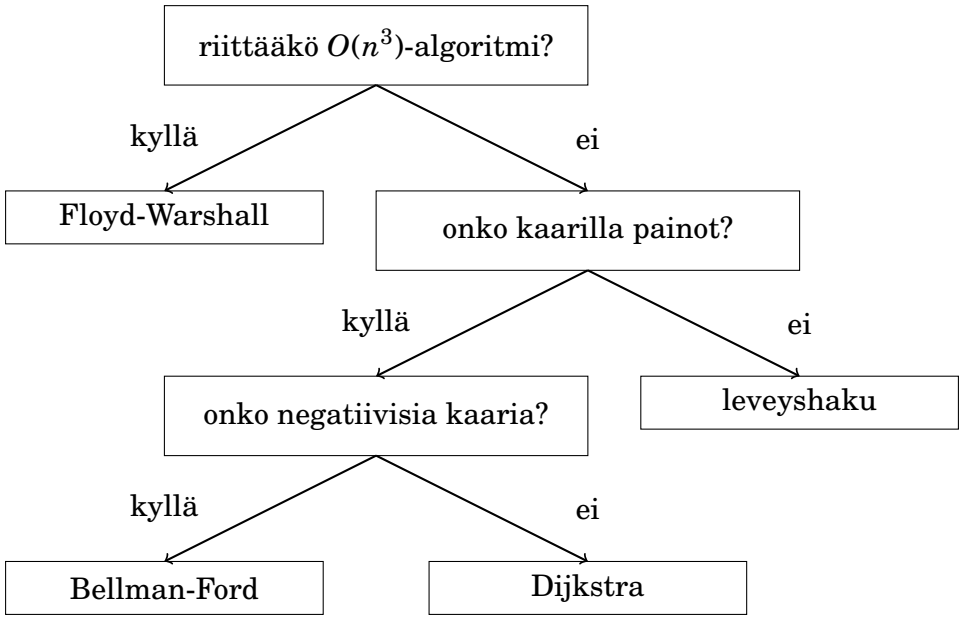
```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
        }
    }
}
```

Algoritmin aikavaativuus on $O(n^3)$, koska siinä on kolme sisäkkäistä silmukkaa, jotka käyvät läpi verkon solmut.

Koska algoritmin toteutus on yksinkertainen, se voi olla hyvä valinta jopa silloin, kun haettavana on yksittäinen lyhin polku verkossa. Tämä vaatii kuitenkin, että verkko on pieni ja kuutiollinen aikavaativuus on riittävä.

## 13.4 Yhteenveto

Olemme käyneet läpi useita algoritmeja, joilla voi etsiä lyhimmän polun verkossa solmusta $a$ solmuun $b$. Jokaisella algoritmilla on jokin etu verrattuna muihin algoritmeihin. Seuraava kaavio auttaa sopivan algoritmin valinnassa:



# Osa III

# Uusia haasteita