Polunhakualgoritmit ja -järjestelmät

Rodion Efremov

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 2. joulukuuta 2014

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Rodion Efremov

Työn nimi — Arbetets titel — Title

Polunhakualgoritmit ja -järjestelmät

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	2. joulukuuta 2014	20

Tiivistelmä — Referat — Abstract

Haettaessa lyhimpiä polkuja verkossa joudutaan väistämättä ottamaan kohdeverkon ominaisuuksia huomioon, mikäli halutaan suoriutua tehtävästä pienimmässä mahdollisessa ajassa. Toisinaan on mahdollista määritellä heuristiikkafunktio, ja siten käyttää A*-perheen algoritmeja; toisinaan tätä ei voida tehdä, ei ainakaan tehokkaasti, jolloin jäljelle jää kaksisuuntaisuus ja/tai verkon esiprosessointi haun nopeuttamiseksi. Tämän dokumentin tarkoituksena on siis antaa (hyvin suppea) katsaus asiaan liittyviin algoritmeihin ja järjestelmiin.

Avainsanat — Nyckelord — Keywords
verkot, lyhimmät polut

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Polunhaku painotetuissa tai painottamattomissa verkoissa on perustavanlaatuinen ongelma, joka ei ole mielenkiintoinen vain itsessään, vaan myös tarvittavana alioperaationa muissa algoritmeissa. Esimerkiksi Edmond-Karpin algoritmi käyttää leveyssuuntaista hakua maksimivuo-ongelman ratkaisemiseksi [1]; multiple sequence alignment -ongelmaa on ruvettu viime vuosikymmeninä ratkomaan myös heuristisin polunhakualgoritmein.

Verkko G on pari (V, A), jossa V on solmujen joukko ja $A \subset V \times V$ on (suunnattujen) kaarien joukko. Suuntaamatonta verkkoa $G' = (V, E)$ voidaan aina simuloida suunnatulla verkolla $G = (V, A)$ siten, että jokaista suuntaamatonta kaarta $\{u, v\} \in E$ kohti laitetaan A:han kaaret (u, v) ja (v, u). (Suunnattu verkko on suuntaamattoman yleistys.) Polunhakua varten verkosta erotellaan kaksi solmua: lähtösolmu s ja maalisolmu t. Jatkossa, $n = |V|$ ja $m = |E|$; näin esimerkiksi leveyssuuntaisen haun aikavaativuus on $\mathcal{O}(n+m)$. Polku on $\gamma_k = \langle u_0, u_1, \ldots, u_k \rangle$, missä mikään solmu ei esiinny yhtä kertaa enempää, ja verkossa on kaari (u_i, u_{i+1}) jokaisella $i = 0, 1, \ldots, k-1$. Polkuun liittyvä kustannus on sen kaarien painojen summa, ja mitä tulee itse painoihin, niiden oletetaan olevan ei-negatiivisia. Ei-painotettujen verkojen kohdalla jokaisen kaaren paino oletetaan olevan 1.

2 Tavallisimmat algoritmit

Keskustellessa polunhakualgoritmeista paras etenemissuunta lienee yleisimmistä tekniikoista ad-hoc ratkaisuihin. Optimaalin polun haku on intuitiivisinta painottomissa verkoissa, joissa niinkin helppo algoritmi kuten leveyssuuntainen haku (Algoritmi 2) on riittävä. Algoritmi käyttää kaksi tietorakennetta: saavutettujen solmujen FIFO-jonon (engl. *first-in, first-out*) ja hajautustaulun $\pi \colon V \to V$, joka assosioi jokaisen saavutetun solmun u kanssa se solmu v, josta haku on edennyt u:hun. Intuitio tämän takana on se, että leveyssuuntainen haun hakuavaruus etenee ”verkon taso” kerralla lähtien lähtösolmusta s, kunnes saavuttaa maalisolmun t, minkä jälkeen löydetty lyhin polku $\langle s = \pi(\pi(\ldots)), \pi(\pi(t)), \pi(t), t \rangle$ voidaan rakentaa rutiinilla Traceback-Path (Algoritmi 1).

Mitä tulee painotettuihin verkkoihin Edsger W. Dijkstra esitti vuonna 1959 kuuluisan polunhakualgoritminsa, joka toimii polynomisessa ajassa [2]. Algoritmin voidaan katsoa yhdistävän ahneuden (engl. *greedy algorithm*), dynaamisen ohjelmoinnin ja inkrementaalisen lähestymistavan. Saatuaan lähtösolmun s, algoritmi laskee lyhimpien polkujen puun lähtien solmusta s kunnes t joutuu *avoimeen listaan* (engl. *open list; search frontier*), ja sitä kautta *suljettuun listaan* (engl. *closed list; settled node list*), jolloin lyhin s, t-polku on löytynyt. Hart et al. esittivät vuonna 1968 kuuluisan A*-algoritminsa [5], joka – samoin kuten Djikstran algoritmi – ylläpitää mm. kunkin saavutetun

Algoritmi 1: TRACEBACK-PATH(x, π, π_{REV})

```
1  u = x
2  p = ⟨⟩
3  while u is not nil do
4  |   p = ⟨u⟩ ∘ p
5  |   u = π(u)
   Kaksisuuntainen haku?
6  if π_REV is not nil then
7  |   u = π_REV(x)
8  |   while u is not nil do
9  |   |   p = p ∘ ⟨u⟩
10 |   |   u = π_REV(u)
11 return p
```

Algoritmi 2: BREADTH-FIRST-SEARCH(G, s, t)

```
1  Q = ⟨s⟩
2  π(s) = nil
3  while |Q| > 0 do
4  |   u = DEQUEUE(Q)
5  |   if u is t then
6  |   |   return TRACEBACK-PATH(u, π, nil)
7  |   for (u, v) ∈ G.A do
8  |   |   if v is not yet mapped in π then
9  |   |   |   π(v) = u
10 |   |   |   ENQUEUE(Q, v)
11 return ⟨⟩
```

solmun u g-arvon $g(u)$, joka on toistaiseksi pienin kustannus lähtösolmusta s solmuun u, ja joka on taattu olemaan pienin mahdollinen heti kun u poistuu avoimesta listasta. Erona on kuitenkin se, että A* käyttää kunkin solmun u prioriteettinä sen f-arvoa, joka on $f(u) = g(u) + h(u)$, missä $h(u)$ on solmun u optimistinen (eli aliarvioitu) etäisyys maalisolmuun. Heuristiikkafunktion optimistisuus on välttämätön, mikäli halutaan taattaa, että solmun t poistuessa avoimesta listasta, vanhempainfunktio π antaa lyhimmän s, t-polun. Lisäksi, heuristiikkafunktio on *monotoninen*, jos jokaisella kaarella (u, v) on voimassa $h(u) \leq w(u, v) + h(v)$.

Lause 1. *Jos heuristiikkafunktio $h\colon V \to \mathbb{R}$ on monotoninen, se on myös optimistinen (engl. admissible).*

Todistus. Olkoon solmut $v, t \in V$ annettu. Nyt $d(v, t)$ on lyhimmän polun

$\langle v_0 = v, v_1, \ldots, v_k = t \rangle$ kustannus

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

Heuristiikkafunktion monotonisuuden nojalla

$$\begin{aligned} h(v) &= h(v_0) \\ &\leq w(v_0, v_1) + h(v_1) \\ &\leq w(v_0, v_1) + w(v_1, v_2) + h(v_2) \\ &\ldots \\ &\leq \sum_{i=0}^{k-1} w(v_i, v_{i+1}) + h(v_k) \\ &= d(v_0, v_k) + h(v_k) \\ &= d(v, t) + h(t) \\ &= d(v, t), \end{aligned}$$

sillä $h(t) = 0$. □

Intuitio tämän järjestelyn takana on se, että A* "tietää" mihin suuntaan haku on suunnattava, jota päästäisiin maalisolmuun, ainakin paremmin kuin Dijkstran algoritmi, jonka hakuavaruus kasvaa laajenevan pallon tavoin "kaikkiin suuntiin". A*:n pseudokoodi on tasan sama kuin Dijkstran algoritmin (Algoritmi 3). Erotuksena riveillä 14 ja 18 $g(x)$:n sijasta on $f(x)$, jolle siis $f(x) = g(x) + h(x)$. Molemmat kutsuvat TRACEBACK-PATH-rutiinin, joka muodostaa lyhimmän polun edeltäjäpuusta (engl. *predecessor tree*) ajassa $\Theta(N)$, missä N on lyhimmän polun solmujen määrä. On huomattava, että A* palautuu Dijkstran algoritmiin määrittelemällä $h(u) = 0$ jokaisella $u \in V$, sillä tuolloin $f(x) = g(x)$.

2.1 Prioriteettijonon valinta

Polkua haettaessa painotetussa verkossa joudutaan käyttäämään prioriteettijonoja, jotka ovat tarpeellisia pitääkseen haut optimaaleina, ja joiden oletetaan tarjoavan ainakin seuraavat operaatiot:

1. INSERT(H, x, k) tallettaa solmun x sen prioriteettiavaimen k kera,
2. DECREASE-KEY(H, x, k) päivittää solmun x talletetun prioriteettiavaimen (pienemmäksi),
3. EXTRACT-MINIMUM(H) poistaa pienimmän prioriteetin omaava solmu,
4. MIN(H) palauttaa, muttei poista keosta solmun, jolla on pienin prioriteettiavain, ja

Algoritmi 3: DIJKSTRA-SHORTEST-PATH(G, s, t, w)

```
   Monikkosijoitus
 1 OPEN, CLOSED, g, π = ({s}, ∅, {(s, 0)}, {(s, nil)})
 2 while |OPEN| > 0 do
 3     u = EXTRACT-MINIMUM(OPEN)
 4     if u is t then
 5         return TRACEBACK-PATH(u, π, nil)
 6     CLOSED = CLOSED ∪ {u}
       Jokaisella solmun u lapsisolmulla x, tee...
 7     for (u, x) ∈ G.A do
 8         if x ∈ CLOSED then
 9             continue
10         g' = g(u) + w(u, x)
11         if x ∉ OPEN then
12             g(x) = g'
13             π(x) = u
14             INSERT(OPEN, x, g(x))
15         else if g(x) > g' then
16             g(x) = g'
17             π(x) = u
18             DECREASE-KEY(OPEN, x, g(x))
   Ei s, t -polkua verkossa G.
19 return ⟨⟩
```

5. SIZE(H) palauttaa jonossa olevien alkioiden määrän.

Operaatiot (4) ja (5) voidaan aina toteuttaa siten, että ne toimivat vakioajassa. Helpoin prioriteettijonorakenne (jatkossa vain ”keko”), jonka operaatiot (1) - (3) käyvät logaritmisessa ajassa, on binäärikeko. Tällaisella keolla Dijkstran ja A*-algoritmit käyvät kumpikin ajassa $\mathcal{O}((m+n)\log n)$. Teoriassa edelläoleva ylläraja voidaan parantaa käyttämällä Fibonacci-kekoa, jonka lisäysoperaatio (1) käy eksaktissa vakioajassa, päivitysoperaatio (2) tasoitetussa vakioajassa, ja poisto-operaatio (3) tasoitetussa ajassa $\mathcal{O}(\log n)$, jolloin haut voidaan suorittaa ajassa $\mathcal{O}(m + n\log n)$. Huomaa, että kaikki tähän asti mainitut keot perustuvat vertailuihin, ja teoriassa enintään yksi operaatiosta INSERT tai EXTRACT-MINIMUM voi käydä eksaktissa tai tasoitetussa vakioajassa, ja toisen on käyttävä ajassa $\Omega(\log n)$, koska muuten algoritmi 4 tällaisella keolla rikkoisi vertailuihin perustuvan lajittelemisen informaatioteoreettisen rajan, joka on $\Omega(n\log n)$.

Jos kuitenkin kaarien painot ovat kokonaislukuja, $\mathcal{O}(m + n\log n)$-rajaa voidaan parantaa: Mikkel Thorup esitti vuonna 2003 keon, jonka poisto-

operaatio käy ajassa $\mathcal{O}(\log\log\min n)$ ja muut operaatiot vakioajassa [12]. Jos kuitenkin kokonaislukupainot ovat väliltä $[0, N)$, poisto-operaatio voidaan suorittaa ajassa $\mathcal{O}(\log\log\min\{n, N\})$. Nyt selvästi haun aikavaativuus tällaisella keolla on $\mathcal{O}(m + n\log\log\min\{n, N\})$. Lisäksi Dial on esittänyt liki triviaalin prioriteettijonon, joka soveltuu erityisesti kokonaislukuprioriteettiavaimiin väliltä $[0, U]$, missä U ei ole suuri (esimerkiksi alle 200), ja jonka lisäys- ja päivitysoperaatiot toimivat vakioajassa ja poisto-operaatio ajassa $\mathcal{O}(U)$, jolloin tällainen rakenne voi olla toivottava heuristisessa painottamattomassa haussa, kuten esimerkiksi $(n^2 - 1)$-palapelissä (engl. *8-puzzle, 15-puzzle, ...*).

3 Kaksisuuntainen haku

Vaikka A* on tyypillisesti tehokkaampi kuin Dijkstran algoritmi, käyttämällä *kaksisuuntaista* hakua, voidaan päästää verrattavissa olevaan suorituskykyyn. Ajatus kaksisuuntaisuuden takana on se, että algoritmi kasvattaa kaksi hakupuuta: yhden normaaliin tapaan ja toisen maalisolmusta ihan kuin kaaret olisi "käännetty" päinvastaiseen suuntaan, kunnes kaksi hakuavaruutta "kohtaavat" keskellä. Nyt jos lyhin polku koostuu N kaaresta, ja verkon solmujen keskiarvoinen aste on d, tavallinen, eli yksisuuntainen haku vaatii ajan

$$\sum_{i=0}^{N} d^i,$$

kun kaksisuuntainen vaatii

$$2\sum_{i=0}^{\lceil N/2\rceil} d^i.$$

Ylläoleva pätee leveyssuuntaiseen hakuun sellaisenaan, ja painotetun haun kohdalla voidaan saada yläraja kertomalla kunkin summan termin tekijällä $\mathcal{O}(\log n)$, joka liittyy algoritmien käyttämään prioritettijonon operaatioihin.

3.1 Kaksisuuntainen leveyssuuntainen haku

Leveyssuuntainen haku hyötyy kaksisuuntaisuudesta, vaikka toteutus vaatii kuusi erilaista tietorakennetta: kaksi FIFO-jonoa Q ja Q_{REV} (kumpikin on riittävä mallintamaan prioriteettijonoja painottomassa haussa). Lisäksi tarvitaan edeltäjäkuvaukset kumpaakin hakusuuntaa varten: π ja π_{REV}. Etäisyyskuvauksien d ja d_{REV} rooli on analoginen Dijkstran algoritmin g-kuvauksen kanssa: $d(u)$ on lyhimmän s, u-polun kaarien määrä (ja siten myös kustannus) ja $d_{REV}(u)$ on lyhimmän u, t-polun kustannus. Aina kun tietyssä hakusuunnassa poistetaan jonosta solmu x, tarkistetaan joko vastakkaisen hakusuunnan kuvauksissa on kuvaus solmulle x. Jos asia on niin, on löydetty välisolmu, johon molemmat hakusuunnat ovat edenneet, ja jos edellä mainittu x, jonka implikoima polun kustannus on $d(x) + d_{REV}(x)$, parantaa

toistaiseksi parhaan tunnetun polun kustannuksen μ, x talletetaan välisolmuksi m, josta myöhemmin rutiini TRACEBACK-PATH (Algoritmi 1) pystyy muodostomaan lyhimmän polun. Yksi mahdollinen rajoite kaksisuuntaisen haun käyttöönotolle on se, että jos painottamattomassa verkossa on useampi lyhin polku, kaksisuuntainen versio ei välttämättä löydä samaa polkua kuten tavallinen. Tämä saattaa olla joissakin sovelluksissa ongelmallista mikäli on tarvetta laajentaa kunkin jonosta poistetun solmun seuraajasolmut tietyssä järjestyksessä.

3.2 Kaksisuuntainen Dijkstran algoritmi

Ylläolevan analyysin nojalla, on selvä, että Dijkstran algoritmi hyötyy kaksisuuntaisuudesta, eikä edellytä minkäänlaista verkon esiprosessointia. Lisäksi, algoritmin vahvuutena suhteessa A*:iin ei ole pelkästään verrattavissa oleva tehokkuus, vaan myös heuristiikkafunktion tarpeettomuus. μ on toistaiseksi lyhimmän polun kustannus, joka suorituksen alussa on ∞. Kun algoritmi löytää toistaiseksi lyhimmän polun hakuavaruuksien kohdatessa, "välisolmu" m ja sen implikoima kustannus μ päivitetään. Haku jatkuu siihen asti, kunnes molempien avointen listojen minimialkioiden kustannusten summa on vähintään μ.

Molemmat hakusuunnat käyttävät alirutiinin EXPAND, mutta eri parametrein: listat OPEN ja CLOSED kuuluvat saman suunnan dataan (suunta d), CLOSED$_2$ on vastakkaisen hakusuunnan $\hat{d}$ suljettu lista, g, π ovat suunnan d kustannuskuvaus ja vanhempainkuvaus vataavasti, m on toistaiseksi lyhimmän polun "välisolmu", jossa kaksi hakuavaruutta kohtasivat, μ on m:n implikoiman polun kustannus, $*$ on laajentumisoperaattori, jolle $*(u)$ antaa solmun u seuraajasolmut, ja kuvaus $w\colon A \to \mathbb{R}_{\geq 0}$ antaa kaarien painot. Parametrit otaksutaan tässä rutiinissa viiteparametreiksi, jolloin esimerkiksi μ ja m ovat päivitettävissä.

Rutiini UPDATE tarkistaa, että yhden hakusuunnan solmu on toisen suljetussa listassa, ja jos asia on niin, yrittää päivittää välisolmun ja siihen liittyvän toistaiseksi pienimmän kustannuksen μ ja se vaatii parametreikseen hakusuunnan d solmun x, suunnan $\hat{d}$ CLOSED-listan, solmun x g-arvo g_x suunnassa d, suunnan $\hat{d}$ g-kuvauksen, välisolmun m ja sen implikoiman kustannuksen μ. Tämänkin apurutiinin kohdalla parametrit otaksutaan viiteparametreiksi, jolloin niiden päivittäminen rutiinissa on mahdollista.

Laajentumisoperaattoreina (engl. *expansion operator*) kaksisuuntaisessa haussa on määriteltävä funktiot $e, e_{REV}\colon V \to \mathcal{P}(V)$, joille $e(u) = \{v \in V\colon (u,v) \in A\}$ ja $e_{REV}(u) = \{v \in V\colon (v,u) \in A\}$. Nyt siis $e(u)$ antaa solmun u lapsisolmut, ja $e_{REV}(u)$ sen vanhempainsolmut, jolloin e käytetään normaalissa ja e_{REV} käännetyssä haussa. Rutiini 8 määrittelee kaksisuuntaisen Dijkstran algoritmin pysähtymisehdon, joka on tarpeeksi vahva pitämään polut optimaaleina ja laskenta-ajan kohtuullisena.

Algoritmi 4: GENERIC-HEAP-SORT(S, H)

```
   Tyhjennä keko
1  H = ∅
2  for i = 1 to |S| do
       S[i] on itsensä prioriteetti.
3      INSERT(H, S[i], S[i])
4  for i = 1 to |S| do
5      S[i] =EXTRACT-MINIMUM(H)
```

Algoritmi 5: BIDIRECTIONAL-BREADTH-FIRST-SEARCH(G, s, t)

```
1  Q, π, d = (⟨s⟩, (s, nil), (s, 0))
2  Q_REV, π_REV, d_REV = (⟨t⟩, (t, nil), (t, 0))
3  m, μ = (nil, ∞)
4  while |Q| > 0 and |Q_REV| > 0 do
5      if m is not nil and d(HEAD(Q)) + d_REV(HEAD(Q_REV)) ≥ μ
       then
6          return TRACEBACK-PATH(m, π, π_REV)
7      u = DEQUEUE(Q)
8      if u is mapped in π_REV and μ > d(u) + d_REV(u) then
9          μ = d(u) + d_REV(u)
10         m = u
11     for (u, v) ∈ G.A do
12         if v is not yet mapped in π then
13             π(v) = u
14             d(v) = d(u) + 1
15             ENQUEUE(Q, v)
16     u = DEQUEUE(Q_REV)
17     if u is mapped in π and μ > d(u) + d_REV(u) then
18         μ = d(u) + d_REV(u)
19         m = u
20     for (v, u) ∈ G.A do
21         if v is not yet mapped in π_REV then
22             π_REV(v) = u
23             d_REV(v) = d_REV(u) + 1
24             ENQUEUE(Q_REV, v)
25 return ⟨⟩
```

Algoritmi 6: EXPAND(OPEN, CLOSED, $\text{CLOSED}_2, g, g_2, \pi, \mu, m, *, w$)

```
1  u = EXTRACT-MINIMUM(OPEN)
2  CLOSED = CLOSED ∪ {u}
3  for x ∈ *(u) do
4      if x ∈ CLOSED then
5          continue
6      g' = g(u)
7      if * is e_REV then
           Käännetty haku.
8          g' = g' + w(x, u)
9      else
           Normaali haku.
10         g' = g' + w(u, x)
11     if x ∉ OPEN then
12         g(x) = g'
13         π(x) = u
14         INSERT(OPEN, x, g(x))
15         UPDATE(x, CLOSED_2, g(x), g_2, μ, m)
16     else if g(x) > g' then
17         g(x) = g'
18         π(x) = u
19         DECREASE-KEY(OPEN, x, g(x))
20         UPDATE(x, CLOSED_2, g(x), g_2, μ, m)
```

Algoritmi 7: UPDATE(x, CLOSED, g_x, g, μ, m)

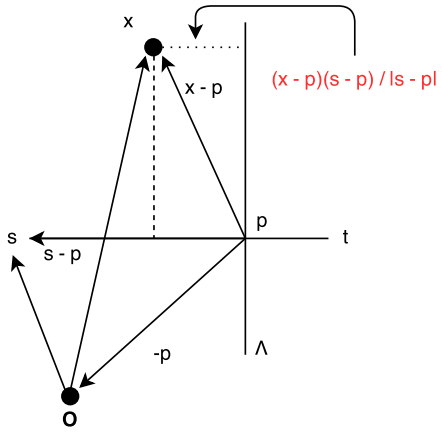
```
1  if x ∈ CLOSED then
2      p = g_x + g(x)
3      if μ > p then
4          μ = p
5          m = x
```

Algoritmi 8: TERMINATE(OPEN, $\text{OPEN}_{REV}, g, g_{REV}, \pi, \pi_{REV}, \mu, m$)

```
1  if g(MIN(OPEN)) + g_REV(MIN(OPEN_REV)) ≥ μ then
2      return TRACEBACK-PATH(m, π, π_REV)
3  return nil
```

Algoritmi 9: BIDIRECTIONAL-DIJKSTRA-SHORTEST-PATH(G, s, t, w)

```
1  OPEN, CLOSED, g, π = {s}, ∅, {(s, 0)}, {(s, nil)}
2  OPEN_REV, CLOSED_REV, g_REV, π_REV = {t}, ∅, {(t, 0)}, {(t, nil)}
3  μ = ∞
4  m = nil
5  while |OPEN| > 0 and |OPEN_REV| > 0 do
6      if m is not nil then
7          p = TERMINATE(OPEN, OPEN_REV,
8                        g, g_REV,
9                        π, π_REV,
10                       μ, m)
11         if p is not nil then
12             return p
       Triviaali kuormantasaus
13     if |OPEN| < |OPEN_REV| then
14         EXPAND(OPEN,
15                CLOSED,
16                CLOSED_REV,
17                g, g_REV, π, μ, m, e, w)
18     else
19         EXPAND(OPEN_REV,
20                CLOSED_REV,
21                CLOSED,
22                g_REV, g, π_REV, μ, m, e_REV, w)
23 return ⟨⟩
```



Kuva 1: $l(c(x))$

3.3 Kaksisuuntainen A*

Kaksisuuntaisen A*:n saa aikaan muuttamalla algoritmin EXPAND (Algoritmi 6) riveillä 14 ja 19 esiintyvät $g(x)$ $f(x)$:ksi ja muuttamalla TERMINATE-rutiinin ehto seuraavanlaiseksi:

$$\max(f(\text{MIN}(\text{OPEN})), f_{REV}(\text{MIN}(\text{OPEN}_{REV}))) \geq \mu,$$

joka on siis Ira Pohlin vuonna 1971 ehdotetun BHPA-algoritminsa pysähtymisehto [9]. Yllä $f_{REV}(u) = g_{REV}(u) + h_{REV}(u)$, missä $h_{REV}(u)$ antaa optimistisen arvion solmun u etäisyydestä lähtösolmusta s.

Vuonna 2007 Taeg-Keun Whangbo ehdotti toisenlaisen kaksisuuntaisen heuristisen hakualgoritmin [13], joka olettaa, että kullakin solmulla u on koordinaatit tasossa $c(u) = (u_1, u_2) \in \mathbb{R}^2$. h-arvon sijasta, määritellään

$$l(x) = \frac{[c(s) - c(p)] \cdot [c(x) - c(p)]}{|c(s) - c(p)|}, x \in \text{OPEN},$$
$$l_{REV}(x) = \frac{[c(t) - c(p)] \cdot [c(x) - c(p)]}{|c(t) - c(p)|}, x \in \text{OPEN}_{REV}.$$

Kun kaksi hakuavaruutta kohtaavat solmussa p, piirretään p:n kautta kulkeva suora Λ, joka on normaali sen suoran kanssa, joka kulkee solmujen s, t kautta. Nyt esimerkiksi jokaisella $x \in \text{OPEN}$, $l(c(x))$ on solmun x etäisyys Λ:sta, kuten kuvasta **??** ilmenee, ja uusi pysähtymisehto on

$$L^1_{\min} \leq \min_{x \in \text{OPEN}} (g(x) + l(x)),$$
$$L^2_{\min} \leq \min_{x \in \text{OPEN}_{REV}} (g_{REV}(x) + l_{REV}(x)),$$

missä $L^1_{\min}$ on lyhimmän s, p -polun kustannus normaalin haun puussa, ja $L^2_{\min}$ analogisesti lyhimmän p, t -polun kustannus vastakkainsuuntaisessa haussa. Kun algoritmin toiminta päättyy, lyhimmän polun kustannus on

$L^1_{\min} + L^2_{\min}$. Whangbo raportoi tavallisen A*:n vievän yhteensä 372 aikayksikköä laskettuna yhteen yli joukon hakuja. Samalla datalla BHPA vie 509 aikayksikköä, ja Whangbon variantti 209 aikayksikköä (huomaa BHPA:n vievän enemmän aikaa kuin tavallinen A*).

4 Kaikkien parien lyhimmät polut

Toisinaan on annettu n solmua ja halutaan löytää lyhimmät polut kaikkien solmuparien välillä. Yksi tehokkaimmista algoritmeista on Floyd-Warshallin algoritmi, joka käy ajassa $\Theta(n^3)$, eikä sen toiminta riipu kaarien määrästä m. Ellei kohdeverkko ole täysi ($m = o(n^2)$), Johnsonin algoritmi saattaa olla parempi valinta, sillä Fibonacci-keolla edellämainittu käy ajassa $\mathcal{O}(n^2 \log n + nm)$.

4.1 Floyd-Warshall -algoritmi

Vaikka Floyd-Warshall -algoritmin aikavaativuus ei ole koskaan Johnsonin algoritmin aikavaativuutta parempi, käytännössä Floyd-Warshall saattaa olla tehokkaampi vaihtoehto, sillä sen toteutus on vain kolme sisäkkäistä silmukkaa, joista kukin iteroi n kertaa, ja sisimmän silmukan runkossa tehdään vain yksinkertainen testi, jolloin koko algoritmiin aikavaativuuteen liittyvät vakiokertoimet ovat pieniä. Polunmuodostusrutiini BUILD-PATH

Algoritmi 10: FLOYD-WARSHALL(n, w)

```
1  d = ℝ^{n×n}
2  π = ℕ^{n×n}
3  for i = 1 to n do
4      for j = 1 to n do
5          d(i, j) = w(i, j)
6          if w(i, j) ≠ ∞ then
7              π(i, j) = j
8          else
9              π(i, j) = nil
10 for k = 1 to n do
11     for i = 1 to n do
12         for j = 1 to n do
13             if d(i, j) > d(i, k) + d(k, j) then
14                 d(i, j) = d(i, k) + d(k, j)
15                 π(i, j) = π(i, k)
16 return (d, π)
```

(Algoritmi 7) eroaa yllä esitetystä rutiinista TRACEBACK-PATH (Algoritmi 1). Yllä on oletettu, että kokonaisluvut $1, 2, \ldots, n$ esittävät solmujoukon: voidaan aina johtaa bijektio $\{1, 2, \ldots, n\} \to V_{domain}$, jolla kuvataan yhden esitystavan solmut toisen esitystavan solmuihin.

Intuitio Floyd-Warshallin algoritmin takana on se, että se tutkii, voidaanko parantaa nykyisen i, j-polun kustannus menemällä solmun k kautta, ja jos voi molemmat matriisit päivitetään kuvastamaan sitä tilannetta, ja koska Floyd-Warshall iteroi kaikkien kolmikkoiden $(k, i, j) \in \{1, 2, \ldots, n\}^3$ yli, se ratkaisee kaikkien-parien lyhimmät polut optimaalisesti.

4.2 Johnsonin algoritmi

Johnsonin algoritmi lisää syöteverkon solmujoukkoon uuden solmun s, kaaret (s, v) kaikilla $v \in V$, ja asettaa kunkin edellä mainitun kaaren painoksi 0. Sen jälkeen algoritmi ajaa Bellman-Ford -algoritmin solmusta s lähtien: jos on löytynyt negatiivisen painon omaava sykli, Johnsonin algoritmin toiminta päätyy. Negatiivisen painon omaava sykli on jono $\langle v_1, v_2, \ldots, v_k \rangle$, missä $v_1 = v_k$, muut solmut kuin $v_1 = v_k$ esiintyvät vain kerran ja

$$\sum_{i=1}^{k-1} w(v_i, v_{i+1}) + w(v_k, v_1) < 0.$$

Mikäli verkossa ei ole negatiivisia sykleja, rakennetaan jokaisesta solmusta lähtien kokonaiset lyhimpien polkujen puut ja sen mukaan kun jokainen puu valmistuu, poimitaan siitä asianomaisten polkujen painot ja talletetaan ne kustannus- ja edeltäjämatriiseihin. Koska Dijkstran algoritmi olettaa kunkin kaaren painon ei-negatiiviseksi, jos verkossa on sellaisia, Johnsonin algoritmi joutuu käyttämään toisenlaista painofunktiota $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ (uudelleenpainotus), missä $h(u) = \delta(s, u)$ ja s on solmu, josta lähtien Bellman-Ford oli ajettu, ja $\delta(s, u)$ on jälkimmäisen laskema lyhin kustannus s, u-polulle.

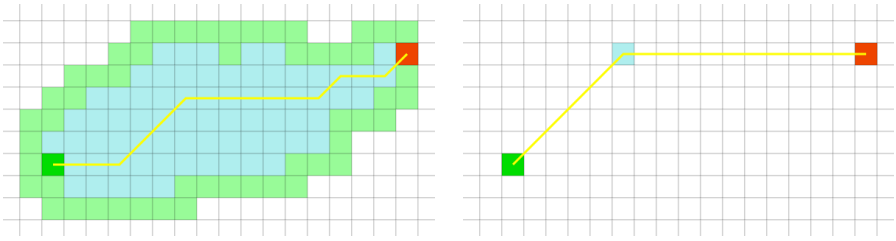
Yllä esitetyistä aikavaativuuksista ilmenee, etteivät asianomaiset algoritmit ole tarpeeksi tehokkaita ainakin n:n arvoilla ≥ 10000. Jos kuitenkin ongelman koko sallii kaikkien parien algoritmin ajon, algoritmi palauttaa "edeltäjämatriisin" (engl. *predecessor matrix*), josta N:n solmun lyhin polku voidaan rakentaa ajassa $\Theta(N)$, mitä ei pysty parantamaan tuon enempää, ei ainakaan ilman edistynempää algoritmiikkaa. (Ja vaikka voisikin, polun tulostaminen ja/tai piirtäminen on jo vähintään $\Omega(N)$.)

5 Ruudukkoverkko ja jump point -haku

Ruudukkoverkko (engl. *grid graph*) on suuntamattoman verkon erikoistapaus, ja sen rakenne voidaan määritellä siten, että kullakin solmulla on kahden kokonaisluvun koordinaatti, eli solmujoukko on $\{(x, y) \colon 1 \leq x \leq w, 1 \leq$

Algoritmi 11: BUILD-PATH(π, i, j)

```
1 if π(i, j) = nil then
2     return ⟨⟩
3 p = ⟨i⟩
4 while i ≠ j do
5     i = π(i, j)
6     p = p ∘ ⟨i⟩
7 return p
```



(a) A*-haku (b) Jump point -haku

Kuva 2: Polkusymmetria

$y \leq h\}$, missä w on ruudukkoverkon leveys ja h on sen korkeus. Nyt kun on annettu kaksi solmua (x_1, y_1) ja (x_2, y_2), jos vain yksi koordinaateista eroavat yksikön verran, kyseessä on vaaka- tai pystysuuntainen kaari ja sen painoksi asetetaan 1. Toisaalta, kun molemmat koordinaatit eroavat yksikön verran, kyseessä on vino kaari, jonka painoksi asetetaan $\sqrt{2}$. On selvää, että jo leveyssuuntainen haku on optimaali tällaisella verkolla, sillä aina kun se etenee vinottain, esim. solmusta (x, y) solmuun $(x + 1, y - 1)$, se ohittaa kahden kaaren siirron solmun $(x + 1, y)$ tai $(x, y - 1)$ kautta, joka pidentäisi lyhimmän polun pituuden kahdella yksiköllä $\sqrt{2}$ sijaan. Heuristinen haku on kannattavaa tällaisella verkkotyypillä, sillä heuristiset funktiot kuten euklidinen (tai Manhattan-metriikka, ellei vinoja kaareja sallita) on helppoa toteuttaa ilman esiprosessointia. Toisaalta, juuri tällaisella verkkotyypillä $A*$ kärsii polkujen ”symmetriasta”, kuten kuvasta 2a ilmenee. Jos siis solmujen s ja t molemmat koordinaatit eroavat, $A*$ kasvattaa suunnikkaanmuotoisen hakuavaruuden solmusta s solmuun t, sillä niiden välissä on monta saman kustannuksen omaavaa polkua. Kuten yleensä erikoistapauksiin rajoittuessa, polunhaku ruudukolla voidaan tehdä paljon tehokkaammin. Vuonna 2011 Harabor ja Grastien esittivät ”jump point search” -nimisen $A*$:n muunnelman, joka karsii pois polkusymmetriat ja etenee ”hyppien” yli monen solmun siinä, missä muut algoritmit etenevät jokaisen välisolmun kautta [4].

6 Dijkstran algoritmi kaarivivuilla

Vuonna 2007 Möhring et al. esittivät mielenkiintoisen tavan nopeuttaa Dijkstran algoritmi [7]. Ajatuksena on osittaa suunnatun verkon $G = (V, A)$ solmujoukko osioihin $V_1, \ldots, V_p$ siten, että

$$\bigcup_{i=1}^{p} V_i = V,$$

ja $V_i \cap V_j = \emptyset$ jokaisella $i \neq j$. Ositus voidaan toteuttaa kuvauksella $r\colon V \to \{1, \ldots, p\}$. Kustakin osiosta V_i puhuttaessa, sen "rajasolmut" (engl. *boundary nodes*) määritellään joukkona

$$B_i = \{v \in V_i\colon \exists (u, v) \in A \text{ siten että } r(v) \neq r(u)\}.$$

Lisäksi, järjestelmä liittää jokaiseen verkon kaareen "kaarivipuvektorin" (engl. *arc-flag vector*), jota voidaan toteuttaa p:n bitin bittivektorina. Nyt jokaisella osiolla V_i järjestelmän esiprosessointialgoritmi ajaa "takaperin" tavallisen Dijkstran algoritmi kustakin rajasolmusta $b \in B_i$, ja asettaa tuloksena syntyvässä lyhimpien polkujen puussa jokaisen kaaren a kohdalla a:n vipuvektorin $r(b)$:nnen bitin päälle. Tuloksena syntyvässä järjestelmässä, haettaessa polkua solmuun t, nopeutettu Dijkstran algoritmi voi karsia kaikki ne kaaret, joiden vektorin $r(t)$:s bitti ei ole päällä, ainakin niin kauan kunnes haku pääsee samaan osioon solmun t kanssa. Tekniikka voidaan siis nähdä tasopainoilevan tavallisen Dijkstran algoritmin ($V_1 = V$) ja kaikkien parien algoritmin (kukin solmu on osio) välillä.

6.1 Ositustekniikat

Kuten ylläolevasta kävi ilmi, "kaarivipu"-Dijkstra vaatii verkon solmujen osituksen, minkä jälkeen joudutaan esiprosessoimaan koko verkko. Koska esiprosessoinnin aika riipuu lineaarisesti kaikkien osioiden kaikkien rajasolmujen yhteenlasketusta määrästä, jälkimmäisen minimointi on toivottavaa.

Mikäli on annettu kunkin solmun koordinaatit tasossa, helpoin tapa osioida verkko ("ruudukointi") on jakaa pienin, kaikki solmut sisältävä suorakulmio w sarakkeeseen ja h riviin. Tämä ei kuitenkaan ole vailla ongelmia: esimerkiksi viidenkymmenen neliökilometrin osio pääkaupunkiseudulla sisältäisi paljon enemmän infrastruktuuria kuin jokin samankokoinen alue Kainuun maakunnassa. Asia voidaan parantaa käyttämällä "nelipuita" (engl. *quad-tree*): koko suorakulmio jaetaan neljään, samankokoiseen suorakulmioon, minkä jälkeen jaetaan jälkimmäiset, ja niin edelleen pysäyttäen jaon niiden suorakulmioiden kohdalla, joissa on enintään κ solmua (κ annetaan nelipuualgoritmille parametrina). Tämä ottaa solmujakauman jo paremmin kuin ruudukointi, mutta ei niin hyvin kuin *kd*-puu (engl. *kd-tree*), joka lajittelee kaikkien solmujen listan ensin esimerkiksi x-koordinaattien perusteella, poimii mediaanialkion x-koordinaatin x_{mid}, ja implisiittisesti jakaa koko listan

kahteen osalistaan $V_\leq$ ja $V_>$, missä $V_* = \{x \in V : x * x_{mid}\}$, minkä jälkeen lajitellaan $V_\leq$ ja $V_>$, mutta jo y-koordinaattien perusteella, ja jako pysähtyy niiden solmujoukkojen kohdalla, joissa on enintään κ solmua (tässäkin κ on kd-puulle annettu parametri).

Neljäs tapa, jota Möhring et al. ovat tarkastelleet, on vuonna 1998 kehitetty METIS [6], joka ei edes tarvitse solmukoordinaatteja. Järjestelmä toimii siten, että syöteverkosta G_i muodostetaan verkko G_{i+1}, joiden suhde on sellainen, että verkossa G_i yhdistetään ”tiheästi” yhdistetyt solmujoukot yhdeksi solmuksi verkossa G_{i+1}. Alunperin syötetään verkko $G_0 = G$, jolloin saadaan G_1, minkä jälkeen syötetään samaan algoritmiin G_1 ja saadaan G_2 ja niin jatketaan kunnes saadaan tarpeeksi pieni verkko G_m. Kun G_m on osioitu, ”laajennetaan takaperin” verkot $G_{m-1}, G_{m-2}, \ldots$ kunnes päästään takaisin alkuperäiseen verkkoon $G_0 = G$, joka on osioitu.

6.2 Tulokset

Koska kaksisuuntainen haku on mahdollista myös kaarivipujärjestelmässä, asia vaatii vain sen, että kuhunkin kaaren liitetään kaksi vipuvektoria, yksi kutakin hakusuuntaa varten. Tällaisella algoritmilla Möhring et al. raportoivat nopeutuksen suhteessa tavalliseen Dijkstran algoritmiin olleen yli 500 noin yhden miljoonan solmun ja 2.5×10^6 kaaren verkolla.

7 Polunhaku ja Multiple Sequence Alignment -ongelma

Multiple sequence alignment -ongelmassa on annettu κ sekvenssiä yli aakkoston Σ (useimmiten 20 aminohappoa), kustannusfunktio $c\colon \Sigma^2 \to \mathbb{Z}$ ja ”välisakko” (engl. *gap penalty*), joka liittyy merkkiin -. Useimmiten tarkoitus on arvioida eri organismien samasta ilmiöstä vastaavien geenien evolutiivinen yhteys. Ongelma on mahdollista muotoilla polunhakuongelmana siten, että kukin sekvenssi laitetaan κ-ulotteisen ”hilan” (engl. *lattice*) akseleiksi ja kukin solmu x voidaan ajatella olevan vektori $(x_1, \ldots, x_\kappa)$, missä x_i on $i$nnesta sekvenssista luettujen merkkien määrä. Kun solmusta x siirrytään solmuun $y = (y_1, \ldots, y_\kappa)$, jokaisella $i = 1, \ldots, \kappa$ $y_i = x_i + 1$ tai $y_i = x_i$, jolloin maksimaalinen solmusta lähtevien kaarien määrä on tasan $2^\kappa - 1$. Jos kahden vierekkäisen solmun koordinaateista jotkut eivät eroa, niittä vastaavista sekvensseistä ei ”lueta” merkkiä, vaan laitetaan sen sijaan välimerkki -. Optimaali rivitys näin olleen löytyy hakemalla lyhin polku solmusta $(0, \ldots, 0)$ solmuun $(|S_1|, \ldots, |S_\kappa|)$. Esimerkiksi, sekvenssien `BACB`, `BCD`, `DB` optimaali rivitys voi olla seuraavanlainen:

B	A	C	B	-
B	-	C	-	D
-	-	-	B	D

jolloin vastaava polku on

$$\langle s = (0,0,0), (1,1,0), (2,1,0), (3,2,0), (4,2,1), (4,3,2) = t \rangle.$$

Mitä tulee ratkaisuun, algoritmi palauttaa κ samanpituuista merkkijonoa yli aakkoston $\Sigma \cup \{-\}$, joista $i$des merkkijono vastaa $i$dennettä sekvenssiä, johon mahdollisesti on laitettu eri kohdissa välimerkit, jolloin kunkin merkkijonon pituus on $L \geq \max(|S_1|, \ldots, |S_\kappa|)$. Ajatus välimerkkien takana on se, että se mahdollistaa sekvenssien osien siirtäämisen eteenpäin kohtiin, joissa kustannus pienenee. Tarkemmin ilmaistuna koko rivityksen (tulosmerkkijonojen) kustannus on

$$\sum_{i=1}^{L} \mathfrak{C}(i),$$

missä

$$\mathfrak{C}(s) = \sum_{1 \leq i < j \leq \kappa} c(M_{i,s}, M_{j,s}).$$

Siis rivityksen kustannus on sen sarakkeiden kustannuksien summa ja kunkin sarakkeen kustannus on sen kaikkien merkkiparien kustannuksien summa. Jos jonkin merkkiparin merkeistä vain yksi on välimerkki, käytetään sen kustannuksena edellä mainittu välisakko; muuten merkkiparin kustannus on annettu kuvauksessa c, ja jos kumpikin merkki on väli, kustannus on $c(-,-) = 0$. Ongelman parametrisoinnin yhteydessä, kustannus ei välttämättä ole niin yksinkertainen: usein on tarpeen käyttää affiini (engl. *affine*) kustannus, joka liittää jokaiseen n:n välimerkin vaakasuuntaiseen sekvenssiin kustannuksen $a + bn$, koska on todennäköisempää, että yhteen kohtaan tulee kerralla n välimerkkiä, kuin se, että tulisi esimerkiksi n kertaa merkki kerrallaan suurinpiirtein samaan kohtaan. Lisäksi, toisinaan halutaan olla ottamatta huomioon ne välimerkit, jotka sijoittuvat rivityksessä sekvenssien alkuun tai loppuun. Edellämainitut seikat vaikeuttavat MSA-algoritmien suunnittelua ja toteutusta.

7.1 Ratkaisutekniikat

Vaikka MSA-ongelmaa määrittävä hilaverkko on syklitön, se koettelee myös kehittyneiden polunhakualgoritmien rajoja, sillä jo muutamalla sekvenssilla verkko on liian iso, jotta sen voisi säilyttää tietokoneen muistissa eksplisiittisesti, jolloin jäljelle jää solmujen generointi laajennusoperaattorin yhteydessä. Toinen – myös muistiin liittyvä – rajoite on se, että algoritmien tietorakenteet paisuvat niin suuriksi, että keskusmuisti loppuu kesken: Yoshizumi et al. raportoivat A*:n pystyvän käsittelemään enintään seitsemmän sekvenohjssia.

He ehdottavat PEA* nimistä algoritmia (engl. *Partial Expansion A**), jonka voidaan ajatella uhraavan hieman aikaa pitääkseen listat pienempinä, jolloin algoritmi pystyy käsittelemään 8 sekvenssia [14]. Algoritmi lajittelee avoimen listan solmut ei f-, vaan F-arvojen perusteella, missä $F(u)$ on solmun u "ei-lupaavien" solmujen pienin f-arvo. Myös on annettu ei-negaativinen katkaisuarvo C; (engl. *cutoff value*). Lapsisolmu pidetään "lupaavana", jos sen f-arvo ei ole suurempi kuin vanhempainsolmunsa F-arvon ja C:n summa. Alunperin kunkin solmun F-arvo on sen f-arvo. Lisäksi, jos solmulla u on ei-lupaavat lapsisolmut, u laitetaan takaisin avoimeen listaan. Kaikki kaikkiaan, C:n arvolla 50, PEA*, vähentää muistintarpeen 87%, vaikka käyttää vain 20% enemmän aikaa kuin A* hilassa, jossa solmun ja sen lapsen f-arvot eroavat enintään 396 yksikköä. Lisäksi, PEA*:n suhde A*:iin on se, että edellinen palautuu jälkimmäiseksi, kun $C = \infty$.

Toinen varteenotettava MSA-algoritmi on Schroedlin kehittämä IDDP (engl. *Iterative-Deepening Dynamic Programming*) [11]. IDDP on ad-hoc ratkaisu, joka eroaa muista polunhakualgoritmeista sikäli, että se tallettaa suljettuun ja avoimeen listoihin ei solmuja, vaan kaareja. Lisäksi, IDDP karsii suorituksen aikana joitakin suljetussa listassa olevia kaareja, ja koska algoritmi vaatii yhtenä parametreistaan kustannuksen ylärajan U, se myös karsii kaaret, joiden implikoima f-arvo on suurempi kuin U.

Schroedl vertaili IDDP:n, PEA$*$:n ja A$*$:n keskenään. Kahden gigatavun keskusmuistin koneella A$*$ pystyi rivittämään maksimissaan yhdeksän sekvenssiä, ja PEA$*$ vaatii vain noin yhden prosentin siitä muistitilasta ja kykenee rivittämään samalla alustalla 12 sekvenssiä. 12 sekvenssillä IDDP vaatii vain noin 67 prosenttiä PEA$*$:n laskenta-ajasta ja käyttää vain kuudennen osan siitä muistitilasta, mitä PEA$*$.

8 k lyhimmät polut

Toisinaan lyhin s, t-polku ei ole riittävä, vaan halutaan löytää $k \in \mathbb{N}$ lyhintä s, t-polkua. Tällainen tarve voi tulla silloin kun vaaditaan kokonaisvaltaisempaa kuvaa verkkona muotoilemasta ilmiöstä. Esimerkiksi multiple sequence alignment -ongelmassa muutama optimaalein rivitys voi paljastaa enemmän informaatio kuin yksi, esimerkiksi siten, että muutamasta rivityksestä voi paljastua merkityksellinen malli. k lyhimmät polut -ongelmalla on kuitenkin kaksi variaatiota: polut ovat syklittömiä (engl. *simple*) tai poluissa sallitaan syklit. Eppsteinin mukaan ensimmäinen variantti on toista algoritmisesti "vaikeampi" [3] ja hän keskittyy toiseen esittäen algoritmin, joka toimii ajassa ja tilassa $\mathcal{O}(m + n \log n + k)$.

9 Haku dynaamisilla verkoilla

Tähän asti käsitellyt verkot olivat staattisia: ne rakennetaan kerran ja niiden yli ajetaan useammat polunhakukyselyt, joiden palauttama lyhin polku riippuu vain lähtö- ja maalisolmuista. Dynaamisissa verkoissa lyhin polku riippuu myös siitä ajasta, jona lähdetään lähtösolmusta/saavutaan maalisolmuun ja tyypillisesti halutaan optimoida reittiin liittyvä ajankäyttö. Käytännön sovellus dynaamisista verkoista ja hakualgoritmista on Reittiopas-palvelu. Haku dynaamisilla verkoilla eroaa staattisesta versiosta myös siten, että painot liitetään myös solmuihin, jotka edustavat linja-auto- tai raitiovaunupysäkkejä ja metroasemia. Esimerkiksi, jos Pekka saapuu hetkellä τ_{Pekka} lähimpään pysäkkiin P odottamaan bussia, joka saapuu sinne hetkellä $\tau > \tau_{\text{Pekka}}$, liitetään dynaamisessa hakualgoritmissa P:tä edustaavaan solmuun paino $\tau - \tau_{\text{Pekka}}$. Toinen epätriviaali vaatimus on tilan säilyttäminen ja päivittäminen; saman tieosuuden voidaan joissain verkon kohdissa kulkea useammantyypisellä ajoneuvolla, ja lisäksi joillakin pysäkkeillä on tarpeen vaihtaa ajoneuvosta toiseen.

Vaikka verkon (ja myös algoritmin) dynaamisuus on väistämätön vaatimus kun halutaan mallintaa suurkaupunkien liikenneinfrastruktuuria ja tarjota reitinsuunnittelupalvelu käyttäjille, se karsii ainakin yhden tavan nopeuttaa haku: kuten Nannicini ja Liberti toteavat, kaksisuuntaistaminen ei ole reaalistinen ajatus, sillä se vaatii, että sekä lähtö- että saapumisajat ovat tiedossa ja toinen niistä on määritettävissä vain siten, että ajetaan yksisuuntainen hakualgoritmi yhden ajankohdan perusteella [8]. Dynaamisella algoritmilla on siis vain kaksi toimintatilaa: kun halutaan hakea lähtöajan perusteella, ajetaan yksisuuntainen haku ”normaalisti” lähtösolmusta maalisolmuun. Toisaalta kun haetaan saapumisajan perusteella, algoritmi hakee ajassa ”taaksepäin” lähtien maalisolmusta kunnes päättyy lähtösolmuun.

10 Lyhimmät polut ja rinnakkaisuus

Toistaiseksi lyhimpien polkujen haku ei ole juuri antanut paljon aihetta rinnakkaistamiseen. Vuonna 1998 Meyer ja Sanders esittivät Δ-stepping -nimisen algoritminsa, joka asettaa kunkin saavutetun solmun u omaan ”koriin” (engl. *bucket*) numero i aina, kun $g(u) \in [(i-1)\Delta, i\Delta)$, jolloin kukin säie käsittelee vain osan kaikista koreista. 2^{19} solmun verkolla, jonka keskiarvoinen aste on 3, Meyer ja Sanders raportoivat peräkkäisen (engl. *sequential*) version olleen 3.1 kertaa nopeampi kuin ”optimoitu” Dijkstran algoritmi, ja 16 suorittimen hajautetussa järjestelmässä nopeutus 9.2 on mitattu suhteessa peräkkäiseen Δ-stepping -algoritmiin. On huomattava, että algoritminsa toiminta riippuu Δ:n arvosta, ja Meyer et al. ehdottavat arvon $\Delta = 4/d$, missä d on keskiarvoinen solmun aste.

Rinnakkaistamiseen liittyvien käytännön ongelmista huolimatta, myös

kaksisuuntaisen A*:n variantti nimeltään NBA∗ sai rinnakkaisen version: PNBA∗ käyttää kaksi säiettä, kukin omaa hakusuuntaa varten, ja sisältää suhteellisen vähän synkronoinnin tarvetta [10]. Esimerkiksi, 15-palapelillä (engl. *15-puzzle*), NBA∗ löysi lyhimmän 58 siirron polun noin 2.5 kertaa nopeammin kuin A*, ja PNBA∗ oli noin tasan kaksi kertaa nopeampi kuin NBA∗.

On huomattava, että rinnakkaistaessa algoritmeja, ei ole mahdollista saada mielivaltaisen suuria nopeutuksia jo Amdahlin lain nojalla, jonka mukaan maksimaalinen nopeutus on

$$\frac{1}{(1-P)+\frac{P}{N}},$$

missä N on suorittimien määrä ja $P \in (0, 1]$ on sen laskennan suhteellinen osuus, jota voidaan tehdä rinnakkain, eikä P ole koskaan 0, sillä jokaisessa rinnakkaisessa laskennassa joudutaan luomaan säikeet, mikä on ainakin osittain peräkkäinen operaatio. Ottaen raja-arvon N:n kasvaessa rajatta, saadaan maksimaalinen (teoreettinen) nopeutus $1/(1-P)$.

11 Yhteenveto

Lähteet

[1] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. ja Stein, Clifford: *Introduction to Algorithms*, luku 26, sivu 727. The MIT Press, 3. painos, 2009, ISBN 0262033844, 9780262033848.

[2] Dijkstra, Edsger W.: *A note on two problems in connexion with graphs.* Numerische Mathematik, 1:269–271, 1959.

[3] Eppstein, David: *Finding the k Shortest Paths.* 1997.

[4] Harabor, Daniel ja Grastien, Alban: *Online Graph Pruning for Pathfinding on Grid Maps.* Teoksessa *25th National Conference on Artificial Intelligence. AAAI*, 2011.

[5] Hart, Peter E., Nilsson, Nils J. ja Raphael, Bertram: *A formal basis for the heuristic determination of minimum cost paths.* IEEE Transactions on Systems, Science, and Cybernetics, SSC-4(2):100–107, 1968.

[6] Karypis, George ja Kumar, Vipin: *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* SIAM J. Sci. Comput., 20(1):359–392, joulukuu 1998, ISSN 1064-8275. `http://dx.doi.org/10.1137/S1064827595287997`.

[7] Möhring, Rolf H., Schilling, Heiko, Schütz, Birk, Wagner, Dorothea ja Willhalm, Thomas: *Partitioning Graphs to Speedup Dijkstra's Algorithm.* J. Exp. Algorithmics, 11, helmikuu 2007, ISSN 1084-6654. http://doi.acm.org/10.1145/1187436.1216585.

[8] Nannicini, Giacomo ja Liberti, Leo: *Shortest paths on dynamic graphs.* 2008.

[9] Pohl, Ira: *Bi-directional search.* Machine Intelligence 6, sivut 127–140, 1971.

[10] Rios, Luis Henrique Oliveiral ja Chaimowicz, Luiz: *PNBA*: A Parallel Bidirectional Heuristic Search Algorithm.* 2011.

[11] Schroedl, Stefan: *An Improved Search Algorithm for Optimal Multiple-Sequence Alignment.* Journal of Artificial Intelligence Research, sivut 587 – 623, 2005.

[12] Thorup, Mikkel: *Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem.* Teoksessa *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, sivut 149–158, New York, NY, USA, 2003. ACM, ISBN 1-58113-674-9. http://doi.acm.org/10.1145/780542.780566.

[13] Whangbo, Taeg Keun: *Efficient Modified Bidirectional* A* *Algorithm for Optimal Route-Finding.* Teoksessa *New Trends in Applied Artificial Intelligence, 20th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2007, Kyoto, Japan, June 26-29, 2007, Proceedings*, sivut 344–353, 2007. http://dx.doi.org/10.1007/978-3-540-73325-6_34.

[14] Yoshizumi, Takayuki, Miura, Teruhisa ja Ishida, Toru: *A* with Partial Expansion for Large Branching Factor Problems.* Teoksessa *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, sivut 923–929. AAAI Press, 2000, ISBN 0-262-51112-6. http://dl.acm.org/citation.cfm?id=647288.721436.