Polunhakualgoritmit ja -järjestelmät

Rodion Efremov

Kandidaatintutkielma-aine
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 6. marraskuuta 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-aine	6. marraskuuta 2014	13

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. Lisäksi, on olemassa ad-hoc ratkaisuja mitä erikoisempia sovelluksia ajatellen: jump point -haku on todennäköisesti tehokkain online-algoritmi ruudukkoverkoilla; toisaalta IDDP (iterative deepening dynamic programming) pystyy linjaamaan jopa yhdeksän aminohapposekvenssia multiple sequence alignment -ongelmassa. 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 leveyssuuntaisen haun maksimivuo-ongelman ratkaisemiseksi; multiple sequence alignment -ongelmaa on ruvettu viime vuosikymmeninä ratkomaan myös heuristisin polunhakualgoritmein.

Verkoista puhuttaessa verkko G on kaksikko (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 $O(n + m)$. 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

Edsger W. Dijkstra esitti vuonna 1959 kuuluisan polunhakualgoritminsa, joka toimii polynomisessa ajassa [1]. 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 [3], joka – samoin kuten Djikstran algoritmi – ylläpitää mm. kunkin saavutetun 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-arvo, joka on siis $f(u) = g(u) + h(u)$, missä $h(u)$ on solmun u optimistinen (eli aliarvioitu) etäisyys maalisolmuun. 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 1). Erotuksena rivillä 3 $g(x)$:n sijasta on $f(x)$, jolle siis $f(x) = g(x) + h(x)$. Molemmat kutsuvat Traceback-Path-rutiinin, joka siis muodostaa lyhimmän polun "edeltäjä-

Algoritmi 1: 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 = arg min_{x∈OPEN} g(x)
4      if x is t then
5          return TRACEBACK-PATH(t, π, nil)
6      OPEN = OPEN − {u}
7      CLOSED = CLOSED ∪ {x}
       Jokaisella solmun x lapsisolmulla u, tee...
8      for (x, u) ∈ G.A do
9          if u ∈ CLOSED then
10             continue
11         g′ = g(x) + w(x, u)
12         if u ∉ OPEN then
13             OPEN = OPEN ∪ {u}
14             g(u) = g′
15             π(u) = x
16         else if g(u) > g′ then
17             g(u) = g′
18             π(u) = x
   Ei s, t -polkua verkossa G.
19 return ⟨⟩
```

puusta" (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 $u \in V$.

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

Algoritmi 2: TRACEBACK-PATH(x, π, π_{REV})

```
1  u = x
2  p = ⟨⟩
3  while u is not nil do
4  |   lisää u p:n alkuun
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  |   |   lisää u p:n loppuun
10 |   |   u = π_REV(u)
11 return p
```

ajan

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

kun kaksisuuntainen vaatii vaan

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

Ylläoleva pätee leveyssuuntaiseen hakuun sellaisenaan, ja painotetun haun kohdalla voidaan saada yläraja kertomalla kunkin summan termin tekijällä $O(\log n)$.

3.1 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. Alla μ on toistaiseksi lyhimmän polun kustannus, joka suorituksen alussa on ∞. Kun algoritmi löytää toistaiseksi lyhimmän polun hakuavaruuksien kohdatessa, "välisolmu" m ja sen implikoiva kustannus μ päivitetään. Haku jatkuu siihen asti, kunnes molempien avointen listojen minimialkioiden kustannusten summa on vähintään μ. Rutiini UPDATE tarkistaa, että yhden hakusuunnan solmu on toisen suljetussa listassa, ja jos asia on niin, yrittää päivittää välisolmun. Laajentumisoperaattoreina (engl. *expansion operator*) kaksisuuntaisessa haussa on määriteltävä funktiot $e, e_{REV} : V \to \mathcal{P}(V)$, joille $e(u) = \{v \in V : (u, v) \in A\}$ ja $e_{REV}(u) = \{v \in V : (v, u) \in A\}$. Nyt siis

Algoritmi 3: EXPAND(OPEN, CLOSED, $\text{CLOSED}_2, g, g_2, \pi, \mu, m, e, w$)

```
1  u = arg min_{x∈OPEN} g(x)
2  OPEN = OPEN - {u}
3  CLOSED = CLOSED ∪ {u}
4  for x ∈ e(u) do
5      if x ∈ CLOSED then
6          continue
7      g' = g(u)
8      if e(u) gives child nodes of u then
           "Normaali" haku.
9          g' = g' + w(u, x)
10     else
           Käännetty haku.
11         g' = g' + w(x, u)
12     if x ∉ OPEN then
13         OPEN = OPEN ∪ {x}
14         g(x) = g'
15         π(x) = u
16         UPDATE(x, CLOSED_2, g', g_2, μ, m)
17     else if g(x) > g' then
18         g(x) = g'
19         π(x) = u
20         UPDATE(x, CLOSED_2, g', g_2, μ, m)
```

$e(u)$ antaa solmun u lapsisolmut, ja $e_{REV}(u)$ sen vanhempainsolmut, jolloin e käytetään normaalissa ja e_{REV} käännetyssä haussa.

3.2 Kaksisuuntainen A*

Kaksisuuntaisen A*:n saa aikaan muuttamalla algoritmin 3 rivillä 1 esiintyvä $g(x)$ $f(x)$:ksi ja muuttamalla TERMINATE-rutiinin ehto seuraavanlaiseksi:

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

joka on siis Ira Pohlin vuonna 1971 ehdotetun BHPA-algoritminsa pysähtymisehto [6].

Vuonna 2007 Taeg-Keun Whangbo ehdotti toisenlaisen kaksisuuntaisen

Algoritmi 4: UPDATE$(x, \text{CLOSED}, g', g, \mu, m)$

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

heuristisen hakualgoritmin [9]. h-arvon sijasta, määritellään

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

Kun kaksi hakuavaruutta kohtaavat solmussa p, piirretään p:n kautta kulkeva viiva Λ, joka on normaali sen viivan kanssa, joka kukee solmujen s, t kautta. Nyt esimerkiksi jokaisella $x \in \text{OPEN}_{REV}$, $l_{REV}(x)$ on solmun x etäisyys Λ:sta, 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. 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 Prioriteettijonon valinta

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

1. INSERT(H, x, k) tallettaakseen solmun x sen prioriteetin k kera,
2. DECREASE-KEY(H, x, k) päivittääkseen solmun x talletetun prioriteetin (pienemmäksi),
3. EXTRACT-MINIMUM(H) poistaakseen pienimmän prioriteetin omaava solmu, ja

Algoritmi 5: 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| · |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 ⟨⟩
```

4. IS-EMPTY(H) varmistaakseen, että jonossa on vielä alkioita.

Helpoin prioriteettijonorakenne (jatkossa vain ”keko”), jonka operaatiot (1) - (3) käyvät logaritmisessa ajassa, on binäärikeko. Tällaisella keolla Dijkstran ja A*-algoritmit käyvät kumpikin ajassa $O((m+n)\log n)$. Teoriassa edelläoleva ylläraja voidaan parantaa käyttämällä Fibonacci-kekoa, jonka lisäysoperaatio (1) käy eksaktissa vakioajassa, päivitysoperaatio (2) tasoitetussa vakioajassa, ja poisto-operaatio (3) tasoitetussa ajassa $O(\log n)$, jolloin haut voidaan suorittaa ajassa $O(m + n\log n)$. 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 7 tällaisella keolla rikkoisi vertailuihin perustuvan lajittelemisen informaatioteoreettisen rajan, joka on $\Omega(n\log n)$. Jos kuitenkin kaarien painot ovat kokonaislukuja, $O(m + n\log n)$-rajaa voidaan parantaa: Mikkel Thorup esitti vuonna 2003

Algoritmi 6: TERMINATE(OPEN, $\text{OPEN}_{REV}, g, g_{REV}, \pi, \pi_{REV}, \mu, m$)

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

Algoritmi 7: GENERIC-HEAP-SORT(S, H)

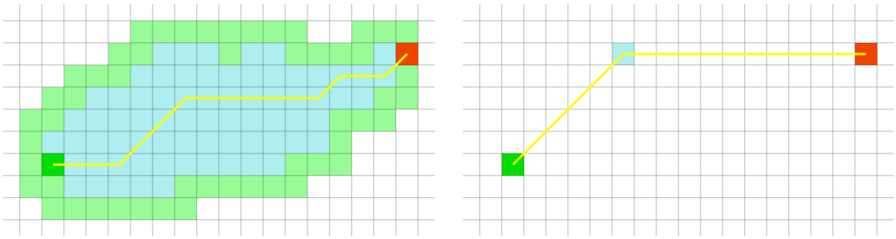
```
   Tyhjennä keko
1  H = ∅
2  for i = 1 to |S| do
       S[i] on itsensä prioriteetti.
3      INSERT(H, S[i], S[i])
4  for i = 1 to |S| do
5      S[i] =EXTRACT-MINIMUM(H)
```

keon, jonka poisto-operaatio käy ajassa $O(\log \log \min n)$ ja muut operaatiot vakioajassa [8]. Jos kuitenkin kokonaislukupainot ovat väliltä $[0, N)$, poisto-operaatio voidaan suorittaa ajassa $O(\log \log \min\{n, N\})$. Nyt selvästi haun aikavaativuus tällaisella keolla on $O(m + n \log \log \min\{n, N\})$.

5 Kaikkien parien lyhimmät polut

Toisinaan on annettu n solmua ja halutaan löytää lyhimmät polut kaikkien kahden eri solmun 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 $O(n^2 \log n + nm)$. Ajatus Johnsonin algoritmissa on ensin tarkistaa, ettei verkossa ole sykliä, jonka kokonaispaino on negatiivinen, minkä jälkeen algoritmi ajaa n kertaa Dijkstran algoritmin jokaisesta solmusta, ja jokaisella kerralla hakee kokonaisen lyhimpien polkujen puun.

Yllä esitetyistä aikavaativuuksista ilmenee, etteivät asianomaiset algoritmit ole tarpeeksi tehokkaita jo 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 $\Omega(N)$.)



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

Kuva 1: Polkusymmetria

6 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 $\{(x, y) \colon 1 \leq x \leq w, 1 \leq y \leq h\}$, 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 1a 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 [2].

7 Dijkstran algoritmi kaarivivuilla

Vuonna 2007 Möhring et al. esittivät mielenkiintoisen tavan nopeuttaa Dijkstran algoritmi [5]. 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.

7.1 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 $V_* = \{x \in V\colon x * x_{mid}\}$, 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 nelipuulle annettu parametri).

Neljäs tapa, jota Möhring et al. ovat tarkastelleet, on vuonna 1998 kehitetty METIS [4], joka ei edes tarvitse solmukoordinaatteja. Järjestelmä toimii siten, että syöteverkosta G_i muodostetaan verkko G_{i+1} siten, että G_i:ssä korvataan "tiheästi" kytketyt solmut yhdellä solmulla verkossa G_{i+1}, ja niin jatkaen pääsee tarpeeksi pieneen verkkoon G_{min}, jonka osittaminen on tarpeeksi tehokasta. Kun G_{min} on ositettu, kuljetaan redusointiketjussa takaperin ja laajennetaan väliverkot $G_{min-1}, \ldots, G_1$, kunnes päästään alkuperäiseen verkkoon $G_0 = G$.

7.2 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.

8 Polunhaku ja Multiple Sequence Alignment -ongelma

Multiple sequence alignment -ongelmassa on annettu κ sekvenssiä yli aakkoston Σ, kustannusmatriisi $c\colon \Sigma^2 \to \mathbb{Z}$ ja "välisakko" (engl. *gap penalty*), joka liittyy merkkiin -. Aakkostona on tyypillisesti 20 aminohappoa, ja tarkoituksena on laittaa eri sekvensseissa välimerkit "-" siten, että jokainen (mahdollisesti) pidennetty sekvenssi sisältää saman määrän merkkejä, ja koko "linjaus" (engl. *alignment*) omaa optimaalin kustannuksen. Jos M on $\kappa \times L$-matriisi, jossa kukin rivi on tietty sekvenssi välimerkkeineen, kunkin sarakkeen s kustannus määritellään olevan

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

ja koko linjauksen kustannus on

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

Lisäksi, määritellään $c(-,-) = 0$, ja aina kun jompikumpi merkki $a \in \Sigma$ ja toinen on väli -, käytetään merkkiparin kustannuksena em. välisakko. (Näin siis optimaalissa linjauksessa ei ole sarakkeita, joiden jokainen merkki on -.)

8.1 Ongelman muotoileminen verkkona

Kun on annettu sekvenssit $S_1, \ldots, S_\kappa$, kukin niistä asetetaan κ-ulotteisen "hilan" (engl. *lattice*) akseleiksi. Kukin solmu voidaan ajattella omistavan koordinaatit $(x_1, \ldots, x_\kappa)$, ja kun siitä siirrytään seuraavaan solmuun $(y_1, \ldots, y_\kappa)$, kukin y_i on joko x_i tai $x_i + 1$, jolloin niiden koordinaattien y_i kohdalla, jotka eroavat koordinaatista x_i, luetaan i:nnen sekvenssin y_i:des merkki, ja muiden kohdalla ei lueta mitään, vaan laitetaan samaan sarakkeeseen välimerkki -. Selvästi, jokaisella hilan "sisäsolmulla" on tasan $2^\kappa - 1$ lapsisolmua, ja vaikka hilaverkko on syklitön, jo muutaman sekvenssin hilaa ei voida säilyttää muistissa eksplisiittisesti, vaan solmut joudutaan generoimaan vanhempainsolmuja laajennettaessa (sekvenssien pituus on sadan merkin suuruusluokkaa). Nyt ongelma palautuu reitinhakuongelmaksi, jossa on löydettävä lyhin polku hilan solmusta $(0, \ldots, 0)$ solmuun $(|S_1|, \ldots, |S_\kappa|)$.

8.2 Ratkaisu

Yoshizumi et al. raportoivat A*:n pystyvän käsittelemään enintään seitsemmän sekvenssia, sillä sen avoimet ja suljetut listat kasvavat liian isoiksi. He ehdottavat PEA* nimistä algoritmia (engl. *Partial Expansion A**), jonka voidaan ajatella uhraavan hieman aikaa pitääkseen listat pienempinä, jolloin algoritmi pystyy käsittelemään 8 sekvenssia [10]. 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 $C = \infty$.

9 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 asteen 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 [7]. 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 edellinen.

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 nopeutus $1/(1-P)$.

Lähteet

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

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

[3] 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.

[4] 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`.

[5] 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`.

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

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

[8] 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.

[9] 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.

[10] 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.