Polunhakualgoritmit ja -järjestelmät

Rodion Efremov

Kandidaatintutkielma-aine
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 14. lokakuuta 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	14. lokakuuta 2014	9

Tiivistelmä — Referat — Abstract

Tiivistelmä.

Avainsanat — Nyckelord — Keywords

a, bb, ccc

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 on toisinaan tarvittava alioperaatio muissa algoritmeissa. Esimerkiksi Edmond-Karpin algoritmi käyttää leveyssuuntaisen haun ratkaistaessaan maksimivuo-ongelmaa; 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. Suuntaamaton verkko $G' = (V, E)$ 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 suuntamattoman 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, ne 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 käy polynomisessa ajassa [1]. Algoritmi voidaan pitää 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, 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 [3]. Erona on kuitenkin se, että $A*$ käyttää kunkin solmun u prioriteettinä sen f-arvo, joka on siis $f(u) = g(u) + h(u)$, jossa $h(u)$ on solmun u optimistinen (eli aliarvioitu) etäisyys maalisolmuun. Intuitio tämän järjestelyn takana on se, että $A*$ "tietää" mihin suuntaan haku on suunnattava, jota pääsisi maalisolmuun, ainakin paremmin kuin Dijkstran algoritmi, jonka hakuavaruus kasvaa laajenevan pallon tavoin "kaikkiin suuntiin".

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     OPEN = OPEN − {u}
 5     if x is t then
 6         return TRACEBACK-PATH(t, π, nil)
 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 ⟨⟩
```

$A*$:n pseudokoodi on tasan sama kuin Dijkstran algoritmin, paitsi että rivillä 3 $g(x)$:n sijasta on $f(x)$, jolle siis $f(x) = g(x) + h(x)$. Molemmat kutsuvat TRACEBACK-PATH-rutiinia, joka siis muodostaa lyhimmän polun "edeltäjäpuusta" (engl. *predecessor tree*) ajassa $\Theta(N)$, missä N on lyhimmän polun sol-

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
```

mujen määrä.

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, yksi normaaliin tapaan ja toinen maalisolmusta ihan kuin kaaret olisivat "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 tekee työn

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

kun kaksisuuntainen olisi tehnyt vain

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

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.

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
16         UPDATE(x, CLOSED_2, g', g_2, μ, m)
17     else if g(x) > g' then
18         g(x) = g'
19         π = x
20         UPDATE(x, CLOSED_2, g', g_2, μ, m)
```

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
```

Rutiini UPDATE tarkistaa, että yhden hakusuunnan solmu on toisen sulje-

Algoritmi 5: BIDIRECTIONAL-DIJKS

```
1  OPEN, CLOSED, g, π = {s}, ∅, {(s,
2  OPEN_REV, CLOSED_REV, g_REV, π_R
3  μ = ∞
4  m = nil
5  while |OPEN| · |OPEN_REV| > 0 do
6      if m is not nil then
7          p = TERMINATE(OPEN, OPE
8                        g, g_REV
9                        π, π_RE
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_f,
18     else
19         EXPAND(OPEN_REV,
20                CLOSED_REV,
21                CLOSED,
22                g_REV, g, π_REV, μ, m
23 return ⟨⟩
```

tussa listassa, ja jos asia on niin, yrittää päivittää välisolmun.

Yllä, e_f on kuvaus, jolle $e_f(u) = \{v \in G.V : (u, v) \in G.A\}$ jokaisella $u \in G.V$, ja $e_b(u) = \{v \in G.V : (v, u) \in G.A\}$. Molemmat siis määrittelevät "laajentumisoperaattorit" (engl. *expansion operator*): e_f normaalissa haussa, ja e_b käännetyssä haussa.

Algoritmi 6: TERMINATE(OPEN, OPEN_{REV}, g, g_{REV}, π, π_{REV}, μ, 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
```

3.2 Kaksisuuntainen $A*$

Kaksisuuntaisen $A*$:n saa aikaan muuttamalla algoritmin 3 rivi 1 seuraavanlaiseksi:

$$u = \arg\min_{x \in \text{OPEN}} f(x),$$

ja korvaamalla rivin 7 kutsu kutsulla TERMINATE$*(\dots)$, jonka määritelmä on alla.

Algoritmi 7: TERMINATE$*$(OPEN, OPEN$_{REV}$, f, f_{REV}, π, π_{REV}, μ, m)

```
1 if μ ≤ max( min_{x∈OPEN} f(x) , min_{x∈OPEN_REV} f_REV(x) ) then
2     return TRACEBACK-PATH(m, π, π_REV)
3 return nil
```

4 Prioriteettijonon valinta

Polkua hakiessa painotetuissa verkoissa 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
4. IS-EMPTY(H) varmistaakseen, että jonossa on vielä alkioita.

Helpoin tehokkaaksi kutsuttu prioriteettijonorakenne (jatkossa vain "keko") on binäärikeko, jonka operaatiot (1) - (3) käyvät ajassa $O(\log n)$, jolloin 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 8 tällaisella keolla rikkoisi 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 keon, jonka poisto-operaatio käy ajassa $O(\log\log\min n)$ ja muut operaatiot vakioajassa [5]. 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 $O(m + n\log\log\min\{n, N\})$.

Algoritmi 8: 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)
```

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 negatiivisen kustannuksen omaava sykli, minkä jälkeen algoritmi ajaa n kertaa Dijkstran algoritmin jokaisesta solmusta, ja jokaisella kerralla hakee kokonaisen lyhimpien polkujen puun.

Ylläesitetyistä aikavaativuuksista ilmenee, ettei asianomaiset algoritmit ole tarpeeksi tehokkaita jo n:n arvoilla yli 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 $\Omega(N)$.)

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. Intuitiivisest ottaen, ruuduk-

koverkoilla heuristinen haku on kannattavampaa kuin kaksisuuntaisuus, ja tavallinen $A*$ on hyvä valinta. Toisaalta, juuri tällaisella verkkotyypillä $A*$ kärsii polkujen "symmetriasta". Kuten yleensä erityistapauksiin 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 kaarivipuilla

Vuonna 2007 Möhring et al. esittivät mielenkiintoisen tavan nopeuttaa Dijkstran algoritmi [4]. 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 $r(b)$:s bitti päälle. Tuloksena syntyvässä järjestelmässä, hakiessa 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.

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 on jakaa pienin, kaikki solmut sisältävä suorakulmio w sarakkeeseen ja h riviin. Tämä ei kuitenkaan ole vailla ongelmia: esimerkiksi viidenkymmenen neliökilometrin osio pääkaupungiseudulla sisältäisi paljon enemmän infrastruktuuria kuin jokin samankokoinen alue Kainuun maakunnassa.

8 Polunhaku ja multiple sequence alignment -ongelma

9 Lyhimmät polut ja rinnakkaisuus

Toistaiseksi lyhimpien polkujen haku ei 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 $g(u) \in [(i-1)\Delta, i\Delta)$.

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] 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`.

[5] 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`.