Lyhimpien polkujen hakualgoritmit ja -järjestelmät

Rodion Efremov

Tietojenkäsittelytieteen laitos, Helsingin yliopisto

Verkot

- Suunnattu verkko on $G = (V, A)$, missä V on solmujen joukko ja $A \subset V \times V$ on suunnattujen kaarien joukko.
- Suuntaamaton verkko $G = (V, E)$ voidaan aina simuloida suunnatulla verkolla (V, A) laittamalla A:han kaaret (u, v) ja (v, u) jokaisella suuntaamattomalla kaarella $\{u, v\} \in E$.
- Jatkossa merkitsemme $n = |V|$ ja $m = |E|$.

Verkot

- k:n kaaren polku on $\gamma = \langle u_0, u_1, \ldots, u_k \rangle$, missä kukin solmu esiintyy vain kerran ja jokaisella $i \in \{0, 1, \ldots, k-1\}$ $(u_i, u_{i+1}) \in A$.
- Painotettujen verkkojen kohdalla otaksumme kunkin kaaren (u, v) painon $w(u, v)$ olevan **ei-negatiivinen**.
- Suuntamattomassa verkossa solmun u "aste" on $d(u) = |\{\{u, v\} \colon \{u, v\} \in E\}|$.
- Suunnatussa verkossa solmun u "sisäänaste" (engl. *indegree*) on $\deg^-(u) = |\{(v, u) \colon (v, u) \in A\}|$ ja "ulosaste" (engl. *outdegree*) on $\deg^+(u) = |\{(u, v) \colon (u, v) \in A\}|$.

Leveyssuuntainen haku

- Löytää lyhimmän polun (yhden monesta mahdollisesta) painottamattomassa verkossa.
- Toteutus vaatii vain jonon ja hajautustaulun.
- Toimii ajassa $\mathcal{O}(n+m) \approx \sum_{i=0}^{N} d_i$, missä N on lyhimmän polun solmujen määrä ja d keskiarvoinen solmun aste tai solmun ulosaste verkon tyypistä riippuen.

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 ⟨⟩
```

Voiko leveyssuuntaisen haun nopeuttaa?

Voi!

Kaksisuuntainen leveyssuuntainen haku

- Aja kaksi hakuavaruutta: yksi normaalin tapaan lähtösolmusta, ja toinen ”takaperin” maalisolmusta.
- Kun kaksi yllä mainittua hakuavaruutta kohtavat jossain ”keskellä”, rakennetaan lyhin polku.
- Aikavaativuus on $2\sum_{i=0}^{\lceil N/2 \rceil} d^i$.
- Verkosta riippuen voi olla jopa ~1000 kertaa nopeampi kuin yksisuuntainen BFS.

Algoritmi 3: 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  τ, μ = (nil, ∞)
4  while |Q||Q_REV| > 0 do
5      if τ is not nil and d(HEAD(Q)) + d_REV(HEAD(Q_REV)) ≥ μ then
6          return TRACEBACK-PATH(τ, 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         τ = u
11     for (u, v) ∈ G.A do
12         if v is not yet mapped in π then
13             π(v) = u
14             ENQUEUE(Q, v)
15             d(v) = d(u) + 1
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         τ = 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             ENQUEUE(Q_REV, v)
24             d_REV(v) = d_REV(u) + 1
25 return ⟨⟩
```

Miten rakentaa polut lyhimpien polkujen puusta?

Tarvitaan vain kuvaus π (ja myös π_{REV} mikäli polku oli haettu kaksisuuntaisella haulla).

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
```

Dijkstran algoritmi

- Vuonna 1959 Edsger W. Dijkstra esitti kuuluisan polunhakualgoritminsa, joka toimii polynomisessa ajassa.
- FIFO jonon sijasta prioriteettijono; kutsutaan usein ”avoimeksi” listaksi (engl. *open set*).
- Hajautustauluun perustuva joukkorakenne; kutsutaan usein ”suljetuksi’ listaksi (engl. *closed set*).
- g-kuvaus, joka kuvaa kunkin saavutetun solmun toistaiseksi pienimpään etäisyyteen lähtösolmusta laskettuna.
- π-kuvaus, aivan kuten BFS:ssä (kuvaa solmun edeltäjäänsä lyhimpien polkujen puussa).
- Kun solmu poistetaan avoimesta listasta, sen g-arvo on optimaali.

Algoritmi 4: DIJKSTRA-SHORTEST-PATH(G, s, t, w)

```
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 u is t then
5          return TRACEBACK-PATH(u, π, nil)
6      OPEN = OPEN − {u}
7      CLOSED = CLOSED ∪ {u}
8      for (u, x) ∈ G.A do
9          if x ∈ CLOSED then
10             continue
11         g' = g(u) + w(u, x)
12         if x ∉ OPEN then
13             OPEN = OPEN ∪ {x}
14             g(x) = g'
15             π(x) = u
16         else if g(x) > g' then
17             g(x) = g'
18             π(x) = u
19 return ⟨⟩
```

Dijkstran algoritmi

- Dijkstran algoritmi voidaan mieltää BFS:n yleistykseksi painotetuissa verkoissa: samoin kuten BFS, Dijkstran algoritmi kasvattaa hakuavaruutensa ”kaikkiin suuntiin” laajenevan pallon tavoin.

A$*$ - haku

- Pseudokoodi tasan sama kuten Dijkstran algoritmilla, paitsi että rivillä 3 $g(x)$ on korvattava $f(x)$:llä, jolle siis $f(x) = g(x) + h(x)$, missä $h(x)$ on optimistinen (eli aliarvioitu) kustannus solmusta x maalisolmuun.
- Intuitio järjestelyn takana on se, että A$*$ "tietää" mihin suuntaan kannattaa lähteä kasvattamaan hakuavaruuden päästääkseen maalisolmuun nopeammin.
- Määrittämällä $h(u) = 0$ kaikilla $u \in V$, A$*$ palautuu Dijkstran algoritmiksi.

Kaksisuuntainen painotettu haku

- Myös A$*$ ja Dijkstran algoritmit voidaan kaksisuuntaista.
- Jos heuristiikkafunktio ei voida määritellä, kaksisuuntainen Dijkstran algoritmi on melkein aina parempit vaihtoehto suhteessa yksisuuntaiseen versioonsa.
- Mitä tulee A$*$:n kaksisuuntaistamiseen, algoritmi ei nopeudu erityisen paljon, sillä jo ei niin hyvä heuristiikkafunktio karsii hakuavaruuden melko hyvin.

Kaksisuuntainen Dijkstran algoritmi

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
```

Kaksisuuntainen Dijkstran algoritmi

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'
```

Kaksisuuntainen Dijkstran algoritmi

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,
```

Kaksisuuntainen A*

Tasan sama kuin kaksisuuntainen Dijkstra, paitsi että operaation Expand rivillä 1 oleva $g(x)$ korvattava lausekkeella $f(x)$, jolle siis $f(x) = g(x) + h(x)$.

Mikä mahtaa olla tehokkain tapaa hakea polut?

Kaikkien parien lyhimmät polut

(1) Aja kaikkien-parit algoritmin, joka palauttaa nk. ”edeltäjämatriisin”.

Kaikkien parien lyhimmät polut

(2) Mikä tahansa N:n solmun polku voidaan rakentaa edellä mainitusta matriisista ajassa $\mathcal{O}(N)$!

Kaikkien parien lyhimmät polut

- Floyd-Warshall toimii ajassa $\Theta(n^3)$.
- Jos $m = o(n^2)$, Johnsonin algoritmi Fibonacci-keolla on asymptoottisesti parempi: $\mathcal{O}(n^2 \log n + nm)$.
- Kumpikaan ei siis tarpeeksi tehokas prosessoimaan kokonaisen valtion tieverkkoa, sillä pelkästään solmuja on helposti yli 100000.

Dijkstran algoritmi kaarivivuilla

Jaa kaikki solmut V osituksiin $V_1, \ldots, V_k$ siten, että

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

ja $V_i \cap V_j = \emptyset$ kaikilla $i \neq j$.

Dijkstran algoritmi kaarivivuilla

k osion ositus voidaan merkitä funktiolla $r\colon V \to \{1, 2, \ldots, k\}$.

Dijkstran algoritmi kaarivivuilla

Jokaiselle kaarelle asetetaan k:n bitin bittivektorin; jos $i$des bitti on päällä, kaari on lyhimmällä polulla johonkin osion V_i solmuun.

Dijkstran algoritmi kaarivivuilla

Osion V_i "rajasolmut" ovat

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

Dijkstran algoritmi kaarivivuilla

Jokaisen osion V_i jokaisen rajasolmun $u \in B_i$ lähtien ajetaan "takaperin" tavallinen Dijkstra ja tuloksena syntyvässä lyhimpien polkujen puussa T asetetaan jokaisen kaaren $(u, v) \in T$ $i$des vipu päälle.