Lyhimmät polut: kypsyysnäyte

Rodion ”rodde” Efremov

5. marraskuuta 2014

Haettaessa lyhimpiä polkuja verkossa on tärkeää olla tietoinen siitä, että tehokkain algoritmivalinta riippuu juuri annetusta verkosta ja mahdollisesti muun informaation saatavuudesta, kuten esimerkiksi heuristiikkafunktion. Toiset huomioon otettavat seikat ovat mm. solmujen määrä ja kaaritodennäköisyys (kaarien täyttöaste). Esimerkiksi, jos verkossa on noin 1000 solmua, kaikkien parien algoritmin ajo on realistinen ajatus, jolloin saadaan "edeltäjämatriisi", josta N solmun polku voidaan rakentaa ajassa $\Theta(N)$. Selvästikin, on vaikea rakentaa lyhin polku tuon nopeammin, mutta tuhannen solmun suuruusluokka ei ole riittävä juuri koskaan mallintamaan kokonaisen valtion reittiverkostoa. Toinen ongelma on se, että käytännön reitinhakujärjestelmät käsittelevät dynaamisia verkkoja, jotka mallintavat liikenneinfrastruktuureja, ja joissa lyhin polku ei riipu pelkästään lähtö- ja maalisolmuista, vaan myös ajasta, jona matka aloitetaan. Käytännössä juuri tällaisissa järjestelmissä kaaripainojen lisäksi käytetään myös solmupainoja. Esimerkiksi, Pekka lähtee tietystä osoitteesta A lähimpään bussipysäkkiin B, jossa hän odottaa ajan t ennen kuin bussi saapuu. Nyt, reitinhakualgoritmi ei tallenna B:n prioriteetiksi vain $t' = w_{kvely}(A, B)$, vaan $t' + t$. Yllä olevan lisäksi, dynaamisten verkkojen kohdalla ei näytä olevan mitään ilmeistä tapaa kaksisuuntaista hakualgoritmin, koska käännetty haku maalisolmusta vaatii saapumisajankohdan ollakseen optimaali, ja sen saa selville vain ajamalla yksisuuntaisen algoritmin lähtösolmusta maalisolmuun. On huomattava, että järjestely vastaa käyttötapausta, jossa käyttäjän määrittämä aika on lähdön aika. Toisessa käyttötapauksessa käyttäjä määrittää saapumisajan, jolloin taas joudutaan tyytymään yksisuuntaiseen hakuun "takaperin" maalisolmusta.

Staattisten verkkojen kohdalla on paljon enemmän valinnanvaraa mitä tulee algoritmeihin. Painottamattomassa verkossa jo niinkin triviaali algoritmi kuin leveyssuuntainen haku on riittävä löytämään lyhin polku ajassa $O(m+n)$, missä m on kaarien määrä syöteverkossa ja n on solmujen määrä. Painotetussa verkossa kuuluisa Dijkstran algoritmi Fibonacci-keolla löytää lyhimmän polun ajassa $O(m+n \log n)$. Mikäli on saatavilla vakioajassa toimiva tapa arvioida kunkin solmun optimistinen (aliarvioitu) etäisyys maalisolmuun (heuristiikkafunktio), A$*$-algoritmi pystyy karsimaan hakuavaruuden oleellisesti, sillä, intuitiivisesti ottaen, A$*$ "tietää" mihin suuntaan kannattaa mennä, jotta saavuttaisiin maalisolmun, toisin kuin Dijkstran algoritmi, joka puolestaan kasvattaa hakuavaruuden "kaikkiin suuntiin" laajenevan pallon tavoin.

Mitä tulee staattisiin verkoihin, kaikki kolme yllä esitettyä algoritmia voidaan toteuttaa kaksisuuntaisina. Ajatus on ajaa kaksi alternoivaa hakua: yksi

normaaliin tapaan lähtösolmusta ja toinen ”takaperin” maalisolmusta. Siinä vaiheessa, kun kaksi hakuavaruutta kohtaavat toisensa ”keskellä”, voidaan muodostaa lyhin polku. A$*$ ei hyödy paljon järjestelystä, mutta Dijkstran algoritmi, ja varsinkin leveyssuuntainen haku, usein nopeutuvat suuruusluokan verran.

Toisinaan verkon rakenne on helppoa kategorisoida erikoistapaukseksi ja suunnitella hakualgoritmi, joka hyötyy verkon erikoisrakenteesta. Ruudukkoverkon tapauksessa jump point -haku on ylivertainen algoritmi, joka – samoin kuten A$*$ – käyttää heuristiikkafunktiota, mutta etenee hakuavaruudessa ”hyppien” vain niiden solmujen yli, joissa saataa olla tarvetta muuttaa etenemissuunta. Toisaalta, multiple sequence alignment -ongelmassa voi käyttää esim. A$*$:n löytääkseen sekvenssien optimaalin rivityksen, mutta jo kahdeksalla sekvenssilla A$*$ kuluttaa liian paljon muistia, jolloin laskenta päätyy ennen kuin rivitys on selvillä. Parannus A$*$:iin on tullut PEA$*$-nimisen algoritmin myötä, joka uhraa vähän laskenta-aikaa pitääkseen muistinkulutuksen pienempänä, ja pystyy rivittämään kahdeksan sekvenssia. Edellistä vahvempi tulos on IDDP-niminen algoritmi, joka kykenee rivittämään optimaalisesti jopa yhdeksän sekvenssia, mutta eroaa muista sikäli, että IDDP ei talenna prioriteettijonoonsa solmuja, vaan kaareja, mikä karsii joitakin sovellusmahdollisuuksia.

Kaikki kaikkiaan, mikään algoritmi ei ole ”tehokkain” jokaisen verkon yli, vaan valintaa tehdessä joudutaan ottamaan syöteverkon ominaisuudet ja muun tiedon saatavuuden huomioon, mikäli halutaan saada tulokset nopeammin. Toisaalta toisinaan erityisen tehokkaan ja spesifin algoritmin (esimerkiksi IDDP:n) toteuttaminen voi olla niin haastavaa, että helpomman, mutta vähemmän tehokkaamman algoritmin toteuttaminen voi olla parempi vaihtoehto, jos otetaan muut asiat huomioon, kuten ohjelmointiaika.