Toteutusdokumentti

Aineopintojen harjoitustyö: Tietorakenteet ja algoritmit

Helsingin yliopisto, Tietojenkäsittelytieteen laitos

Tekijä:

Tomi Virtanen

tomi.virtanen@helsinki.fi

014140105

Toteutusdokumentti

1. Ohjelman yleisrakenne

Algoritmit ohjelmassa ovat toisistaan riippumattomia. Jokainen algoritmi ja tietorakenne on omana luokkanaan. Main kutsuu sisäistä metodiaan, joka alustaa sokkelo ja algoritmi luokat. Käytettäessä osa algoritmeista tarvitsee tietorakenteita toteutuksessaan ja alustaa ja luo ne itse.

Kun algoritmeja ajetaan alustetaan aluksi alustetaan ja luodaan sokkelo -luokka, joka luo sokkelot jotka annetaan parametrina ajettaville algoritmeille. A* algoritmi saa parametrinään sokkelon ja käyttää apunaan keko- ja hajautustaulu tietorakenteita. Hajautustaulu käyttää apunaan linkitettyjä listoja yhteentörmäystilanteita varten. Dijkstra käyttää aputietorakenteenaan minimikekoa lähinten reittien läpikäymistä varten. Linkitetty lista puolestaan koostuu listasolmuista.

Sokkelo käyttää randomia luodessaan uuden matriisi muodossa esitetyn sokkelon. Sokkelo käyttää apunaan myös iteratiivista DFS -läpikäyntiä tarkastakseen että sokkelo on mahdollista ratkaista. Iteratiivinen DFS käyttää apunaan pino tietorakennetta.

2. Aika- ja tilavaativuudet

Ohjelman aika- ja tilavaativuuksista keskeisimmät kokonaisuudet ovat algoritmien vaativuudet. Näiden tarkastelemisessa keskeistä on myös selvittää niiden käyttämien tietorakenteiden operaatioiden aika- ja tilavaativuudet. Koska työssä on tarkoitus vertailla algoritmien tehokkuuksia jätetään vaativuusarvio sokkelon osalta toteuttamatta. Myöskään Bruteforce toteutusta ei lähdetä O notaation osalta tarkastelemaan suuremmalla tarkkuudella, sillä sen hakemat reitit eivät lähtökohtaisesti etsi lyhintä reitti kuten tarkasteltavat vaan sen koko toteutus perustuu sattumanvaraisuuteen, joten sen paras tapaus voi olla todella huono ja paras tapaus todella hyvä. Brute force algoritmin osalta saadaan kuitenkin vertailukohtaa silloin kun vertaillaan algoritmien tehokkuutta. Jos bruteforcessa haluttaisiin etsiä lyhin reitti tulisi se käydä mahdollisesti n kertaa läpi, jolloin jopa eksponentiaaliset vaativuudet olisivat mahdollisia huonoimmassa tapauksessa.

Bellman-Ford

Bellman-Ford algoritmin aikavaativuus on lähtökohtaisesti reunat kertaa solmujen määrä. eli O(mn). Algoritmin alustusoperaation eli kun kaikki solmut käydään läpi ja alustetaan 'äärettömään' on O(|V|) eli solmujen määrä. Algoritmin päärunko eli kaikkien solmujen ja reunojen läpikäynnin aikavaativuus on O(mn), sillä kaikki kaaret m käydään läpi n -solmua kertaa. Kaarten relaksointi ja validiuden tarkastus on myös selvästi vakioaikainen operaatio. Aikavaativuus luokaksi jää tällöin O(mn). Algoritmia suorittaessa ei muodostu rekursiopinoa ja merkityksellisin tallessa pidettävä asia on solmujen määrä n. Vaikka tallessa pidettäviä taulukoita on kaksi ei algoritmin tilavaativuusluokka kuitenkaan kasva vaan algoritmin tilavaativuus on selvästi O(n).

Dijkstra

Dijkstran algoritmin aikavaativuus on O((m+n) log n)). Tässä merkityksellinen tekijä on kaarten määrä eli aikavaativuudeksi muodostuu O(m log n), jossa m on kaarten määrä ja n on solmujen määrä. Algoritmi käyttää apunaan minimikekoa, josta erityisesti heap insert, heap del min ja heap dec key metodeja. Kekoehdon nojalla keko-operaatioiden aikavaativuuksiksi muodostuu O(log n), jos keossa on n alkiota.

Toistolauseessa kutsuttavaa heap del min operaatiota suoritetaan n kertaa. Eli sen aikavaativuudeksi muodostuu n log n. Jokaiselle solmulle kutsutaan relax operaation ehtolauseessa heap insert operaatio maksimissaan m kertaa eli sen aikavaativuudeksi muodostuu m log n. Toisessa ehtolausekkeessa päivitetään solmuja, eli kutsutaan heap dec key maksimissaan m kertaa eli aikavaativuudeksi muodostuu m log n. Algoritmin kokonaisaikavaativuudeksi muodostuu tällöin O((n+m) log n).

Algoritmissa pidetään tallessa kahta solmujen määrän kokoista matriisia, eikä algoritmissa muodostu rekursiopinoa. Tällöin tilavaativuudeksi tulee selvästi 2n eli O(n).

A* algoritmi

A* algoritmi on muuten hyvin samanlainen Dijkstran algoritmin kanssa, paitsi että sen pahimman tilanteen aikavaativuus on todella huono ja se on keskimäärin nopeampi kuin Dijkstra. A*:n heuristiikka ohjaa sitä keskimäärin paremmin kohti maalia ja vähentää läpikäytävien solmujen ja kaarten määrää, jos heuristiikka on oikein valittu. Huonoimmassa tilanteessa taas A* joutuu käymään joka tapauksessa kaikki tilanteet läpi jolloin se on raskaammasta toteutuksestaan johtuen jopa huonompi kuin Dijkstra.

3. Suorituskyky ja O-analyysivertailu

Suorituskyky erot riippuvat todella paljon syötteestä jota ajetaan. On huomattavissa selkeitä eroja maalin sijainnin suhteen. Jos maali sijaitsee aivan päinvastaisessa nurkassa kuin lähtöpiste saavuttavat dijkstra ja varsinkin A* suhteessa paljon huonompia tuloksia kuin voisivat.

Vertailua suoritetaan pienillä, keskisuurilla ja todella suurilla syötteillä. Suurimmissa syötteissä pysytään järkevyyden rajoissa laskentaan käytettyjen aikojen suhteen. On myös huomioitavaa että vertailua suoritetaan fuksikannettavalla eikä alunperin suunnitellulla pöytäkoneella, jolla oltaisiin päästy tehokkaampiin tuloksiin. Kuitenkin ideana on vertailu joten myös mini kannettavalla tulee selkeästi näkyviin eri syötteiden vaikutus algoritmien laskennoissa.

Seuraavaksi on esitetty graafisesti tuloksia algoritmien vertailusta erilaisilla syötteillä sekä samat sokkelot erilaisilla maalipisteillä. Tuloksista on huomattavissa selkeää eroja erityisesti dijkstran ja A*:n kohdalla. Varsinkin jälkimmäisen. Onkin varsin selvää että A* sopii paremmin sellaisiin sokkeloihin joissa piste ei sijaitse missään ääripäässä vaan heuristiikalla on saatavilla selvää hyötyä. Kaikkein pienimmän 15x15 sokkelon, jolla vertailuja suoritettiin tulokset on jätetty pois ja keskitytty keskisuureen ja suurempaan sokkeloon, joilla erot ovat selkeämmin näkyvillä.

Lopussa on liitteenä graafisia kuvaajia algoritmien suoritusajoista eri sokkeloiden maalipisteillä. Kuviossa sininen palkki kertoo suoriutumisen kun maali on matriisin oikeassa alanurkassa, punainen kertoo kun maali on oikeassa ylänurkassa ja vihreä kertoo tilanteen kun maali on jossai välillä.

Kun tarkastellaan kuviota 1 voidaan havaita selvä ero Bellman-Fordista Dijkstraan ja A*:iin. Vasemman puoleisin pykälä kertoo algoritmin suoriutumisen silloin kun maalipiste sijaitsee aivan päinvastaisimmassa nurkassa. On huomattavaa miten Bellman-Fordin suoriutumiseen ei oikeestaan ollenkaan vaikuta se missä pisteessä maali sijaitsee, sillä se joutuu joka tapauksessa käymään kaikki lävitse.

Dijkstrassa ja A*:ssa on nähtävillä selvä ero suoriutumisessa kun vertaillaan suorituksia siten missä pisteessä niiden maali sijaitsee. A* pärjää selkeästi huonommin kuin Dijkstra jos se joutuu käymään jokatapauksessa suuren määrän alkioita lävitse eikä heuristiikkaa päästä hyödyntämään. A*:n suoriutuminen paranee taas selvästi silloin kun heuristiikkaa on mahdollista sokkelossa hyödyntää. Dijkstra pärjääkin todella hyvin sellaisissa sokkeloissa joissa ei voida olettaa mitään maalipisteen sijainnilta. A*:n etu tulee vasta heuristiikan hyödyntämisen osalta. Dijkstra suoriutuukin A*:a paremmin niissä tilanteissa joissa heuristiikkaa ei päästä suuremmin hyödyntämään.

Kuviossa 2 ero Bellman-Fordista Dijkstraan ja A*:n kasvaakin todella selvästi. Bellman-Fordin suoritus kestää jopa 140.000 ms, kun Dijkstra:lla menee n 2000ms ja A*:lla 4000ms. Erot kasvavat vielä huomattavasti kun vertaillaan sokkeloita joissa maalipiste on sijoitettu niin ettei kaikkia solmuja jouduta käymään lävitse. Kun Dijkstra ja A* pysähtyvät maaliin joutuu Bellman-Ford silti käymään kaikki mahdolliset vaihtoehdot lävitse ja tämän huomaa suoritusajassa todella selvästi.

Suurempien syötteiden A*:n suorituksen heikkenemisen uskon johtuvan siitä että käytössä oleva hajautustaulu ei ole kaikista optimein tähän kokoluokkaan vaan vaatisi erikseen hienosäätöä. Dijkstran ja A*:n ero tuntuukin olevan se että Dijkstran implementointi on huomattavasti helpompaa kuin A*:n joka on riippuvainen tarkasti valitusta heuristiikasta ja hajautustaulun toteutuksesta. Pienemmillä syötteillä toteuttamani hajautustaulu toimi kuitenkin selvästi nopeammin kuin java:n valmis HashSet.

4. Puutteet ja parannusehdotukset

A*:n toteutus jäi vähän mietityttämään ja sen käyttämän hajautustaulun toteutus ja optimointi voisi olla järkevää jos sitä tarvittaisiin oikeaan käyttöön. Hajautusrakenteen optimointi oli kuitenkin melko vaikeaa kaikille mahdollisille sokkeloiden koille. 150x150 koon matriisille se oli hyvin optimoitu ja A* toimikin tämän kokoluokan matriiseille paremmin kuin Dijkstra.

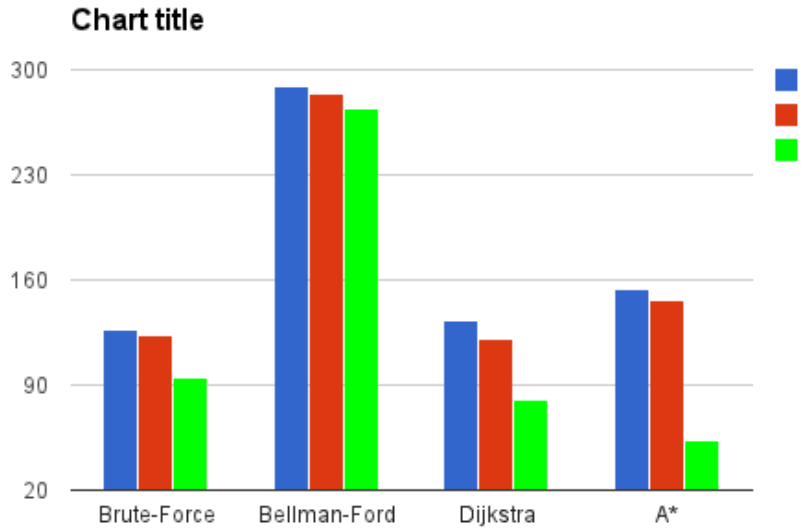
5. Lähteet

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, 3rd ed., MIT Press, 2009.

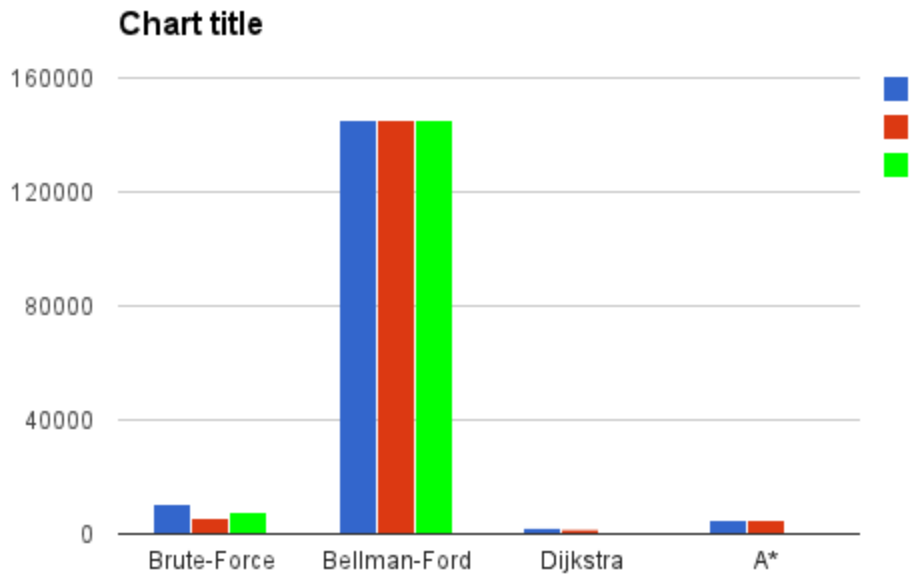
Patrik Floreen. Tietorakenteet ja algoritmit kurssimoniste.
http://www.cs.helsinki.fi/u/floreen/tira2012/tira.pdf

Patrik Floreen. Tietorakenteet ja algoritmit - verkkoalgoritmien vertailumoniste.
http://www.cs.helsinki.fi/u/floreen/tira2012/lisalehti2.pdf

6. Liitteet



KUVIO 1. 150x150 sokkelon aikavertailuja (ms) eri maalipisteillä



KUVIO 2. 1500x1500 sokkelon aikavertailuja (ms) eri loppupisteiden sijainneilla.