Toteutusdokumentaatio

Ohjelman rakenne

Ohjelma lukee bmp-muotoisia kuvia ja hakee niissä lyhimpiä polkuja. Oletuksena polut haetaan vasemmasta yläkulmasta vasempaan alakulmaan. Tarkoituksen on vertailla erilaisten algoritmien suorituskykyä. Bittikartta-luokan avulla voidaan lukea BMP-kuva char[][] -taulukoksi. Ohjelma tukee tiettyjä värejä, jotka on määritelty Bittikartta-luokassa. Char[][]-taulukosta voidaan luoda verkko Verkko-luokan avulla. Eri värien painot on määritelty Verkko-luokassa. On mahdollista luoda painoton verkko. Tällöin annetaan Bittikartta-luokalle mustavalkoinen kuva ja annetaan Verkko-luokalle parametri seinienLäpiSaaLiikkua = false.

Verkko pitää sisällään kaksiulotteisen taulukon Solmu-olioita. Jokaisella Solmu-olioilla on lista Kaari-olioita. Jokaisella kaarella on kohdesolmu ja paino. Lista, johon kaaret on talletettu on järjestyksessä täyttyvä dynaaminen taulukko.

Verkko voidaan antaa algoritmien metodeille, jotka hakevat ja luovat lyhimmän polun annetusta aloitussolmusta annettuun loppusolmuun. Kun algoritmi on hakenut lyhimmän polun, niin ratkaisu on mahdollista piirtää uuteen tiedostoon Bittikartta-luokan avulla.

Dijkstra ja A* -algoritmit vaativat toimiakseen Minimikeko-rajapinnan toteuttavan keon. Minimikekona voi käyttää fibonaccikekoa tai binaarikekoa. Binaarikeko-luokka pitää sisällään BinaariKekosolmu-luokan olioita, jotka sisältävät viitteen algoritmien käyttämiin solmuihin. Kekosolmuihin on myös talletettu niiden indeksi binaarikeon sisäisessä taulukossa. Fibonaccikeko tallettaa Solmujen viitteet Fibonaccikekosolmu-olioihin. FibonaccikekoSolmu ja BinaarikekoSolmu toteuttavat kekoSolmu rajapinnan. Jokaisella solmulla on viite kekosolmuun.

BFS ja Bidirectional -algoritmit käyttävät aputietorakenteena jonoa. Jono-luokka on taulukkopohjainen automaattisesti kasvava jono, joka pitää sisällään Solmu-luokan olioita.

Algoritmien vertailu on mahdollista Suorituskykytesti-luokan metodeilla. testaaPainollinenVerkko ja testaaPainotonVerkko käyttävät testaaKentta-metodia. Tämä metodi mittaa algoritmien suoritusta. Metodi hakee lyhimmän polun annetulla algoritmilla kuvan vasemmasta yläkulmasta kuvan oikeaan alakulmaan. testaaPainollinenVerkko testaa painollisen verkon Dijkstra ja A* -algoritmeilla. testaaPainotonVerkko testaa painottoman verkon Dijkstra, A*, BFS ja Bidirectional -algoritmeilla.

Saavutetut tila- ja aikavaativuudet

Merkitään |V| = solmujen määrä, |E| kaarten määrä.

Dijkstran algoritmi toimii ajassa O((|V| + |E|) log |V|). (1) Omassa toteutuksessani rivit 39 ja 44 käyvät läpi kaikki solmut ja asettavat ne kekoon. Näiden rivien aikavaativuus on O(|V|) ja tilavaativuus on O(|V|). Rivillä 46 kutsutaan keon delMin -operaatiota, joka toimii Fibonacci- ja binäärikeossa ajassa O(log |V|). Toistolauseessa jokainen solmu haetaan keosta, joten tästä aikavaativuudeksi tulee yhteensä O(|V|log|V|). Jokaisen solmun jokaiselle kaarelle tehdään

decreaseKey-operaatio. Binaarikeossa se toimii ajassa O(log |V|). (3) Fibonaccikeossa se on keskimäärin vakioaikainen, mutta pahimmassa tapauksessa logaritminen. (2) Binäärikeolla decreaseKey-operaation kutsumisesta seuraa O(|E| * log |V|) aikavaativuus. Fibonaccikeolla decreaseKeyn kutsumisesta kaarten määrän verran seuraa aikavaativuus O(E * log |V|). Rivillä 45 alkavan toistolauseen aikavaativuus on siis binäärikeolla O(|V|log|V| + |E|log|V|) eli O((|V| + |E|)log |V|). Fibonaccikeolla aikavaativuus on sama O(|V|log|V| + |E|log|V|) eli O((|V| + |E|)log |V|). Saavutettu aikavaativuus on siis sama kuin tavoiteltu. Tilavaativuus on O(|V|) koska aputietorakenteena käytetään kekoa, johon kaikki solmut laitetaan.

A* on muokattu Dijkstran algoritmista.(4) Ainoa ero on heuristiikan käyttö solmuja valitessa. Tästä seuraa se, että myös A* toimii ajassa O((|V| + |E|) log |V|). Loppuetäisyyden arviointi tehdään samalla kun kaikki solmut käydään läpi ja lisätään kekoon. Arviointi on vakioaikainen operaatio, joten se ei aiheuta muutosta aikavaativuuteen. Toteuttamani A*-algoritmin aikavaativuus on siis sama kuin Dijkstran algoritmin aikavaativuus O((|V| + |E|)log |V|). Myös tilavaativuus on sama O(|V|).

Bellman-Ford algoritmin aikavaativuus on O(|V||E|). (5) Toteuttamassani algoritmissa kutsutaan vakioaikaista relax-operaatiota jokaisella kaarelle |V| - 1 kertaa. Aikavaativuus on on siis O((|V| - 1)|E|) eli O(|V||E|). Algoritmi käyttää ainoastaan muutamaa apumuuttujaa, joten tilavaativuus on O(1).

BFS-algoritmin aikavaativuus kuuluisi olla luokkaa O(|V|+|E|). (6) Toteuttamani BFS-algoritmi lisää solmut jonoon korkeintaan kerran. Jonossa on siis korkeintaan |V| alkioita. Tilavaativuus on siis O(|V|). Jokainen solmu poistetaan ja lisätään jonoon korkeintaan kerran. Enqueue ja Dequeue -operaatiot ovat vakioaikaisia, joten jono operaatioihin kuluu aikaa O(|V|). Kunkin solmun vieruslista käydään läpi kun solmu poistetaan jonosta, eli korkeintaan kerran. Vieruslistojen läpikäyntiin kuluu siis aikaa O(|E|). Aikavaativuudeksi saadaan siis O(|V| + |E|).

Bidirectional-algoritmi toimii ajassa O(|V| + |E|). Algoritmissa suoritetaan kaksi leveyshakua, jotka pahimmassa tapauksessa käyvät molemmat läpi puolet solmuista. Algoritmin tilavaativuus on O(|V|). Pahimmillaan kaikki solmut lisätään jonoon. Ensimmäisen haun läpikäymät solmut tulee tallettaa taulukkoon, jonka koko on |V|.

Merkitään alkioiden määrää kirjaimella n.

Binaarikeossa insert, decreaseKey ja delMin -operaatiot ovat toimivat logaritmisessa ajassa. (3) Toteuttamani heapify siirtää alkiota korkeintaan keon korkeuden verran joten se toimii logaritmisessa ajassa. delMin-operaatiossa poistetaan indeksissä 0 oleva solmu ja sen paikalle siirretään keon viimeisessä indeksissä oleva solmu. Tähän kuluu aikaa O(1). Tämän jälkeen kutsutaan logaritmista heapify-operaatiota. DelMin-operaation aikavaativuus on siis O(log n). Min-operaatio on vakioaikainen koska pienin alkio on aina indeksissä 0. Toteuttamassani decreaseKey-operaatiossa solmua siirretään keossa ylöspäin korkeintaan keon korkeuden verran joten se toimii ajassa O(log n). Insert-operaatiossani lisättyä alkiota siirretään tarvittaessa ylöspäin keon korkeuden verran joten se toimii ajassa O(log n).

Fibonaccikeossa insert toimii ajassa O(1). DecreaseKey on keskimäärin vakioaikainen. DelMin on keskimäärin logaritminen operaatio. (2) Toteuttamassani fibonaccikeossa insert-operaatiossa lisätään solmu linkitettyyn juurilistaan. Insert toimii siis ajassa O(1). Delmin operaatiossa koko juurilista käydään läpi ja solmuja asetetaan toisten lapsiksi. Kaikki juurilistan alkiot lisätään listaan ja se käydään läpi. Operaation tilavaativuus on siis keskimäärin O(log n), mutta pahimmassa tapauksessa O(n). Delmin-operaation jälkeen juurilistassa on korkeintaan koon fii-kantaisen logaritmin verran solmuja. Siis delmin toimii keskimäärin logaritmisessa ajassa. Parhaimmassa tapauksessa decreaseKey-operaatio leikkaa solmun vanhemmastaan ja lisää sen juurilistaan. Tämä on vakioaikainen operaatio. Keskimäärin decreaseKey toimii siis ajassa O(1).

Toteuttamani jonon enqueue-operaatiossa asetataan solmu taulukkoon ajassa O(1), jos jono ei ole täynnä. Jos jono on täynnä täytyy taulukon kokoa kasvattaa, johon kuluu aikaa O(n). Enqueue-operaatio toimii siis keskimäärin ajassa O(1) ja pahimmassa tapauksessa ajassa O(n). Dequeue-operaatiossa haetaan solmu tietyllä indeksillä taulukosta. Tämä toimii ajassa O(1). Lista-luokka on samankaltainen kuin jono. Lista on myös itsestään kasvava. Tällöin Lista-olioon lisääminen on keskimäärin vakioaikaista, mutta pahimmassa tapauksessa lineaarista.

Suorituskyky ja O-analyysivertailu

Testausdokumentissa piirrettiin käyrät eri vakiokertoimilla $|V| + |E|$ ja $(|V| + |E|) \log |V|$ kuvaajille. Ilmeni, että toteutetut algoritmit toimivat odotetussa ajassa, sillä niiden suoritusaika jäi aina kuvaajien alapuolelle.

Suorituskykytestauksessa ilmeni, että painottomille verkoille paras algoritmi on BFS. Tämän kokoisilla syötteillä Bidirectional-algoritmin käyttäminen ei tuonut merkittävää etua.

Painottomilla verkoilla A* suoriutui huomattavasti paremmin verkoissa, joissa heuristiikan käytöstä on hyötyä. Muuten Dijkstra oli nopeampi.

Näin pienillä ainestoilla binaarikeko päihitti fibonaccikeon kaikilla alueilla.

Parannusehdotukset

Ohjelmasta saisi mielenkiintoisemman, jos kenttiä saisi piirtää kuka tahansa. Tällöin jokaiselle värille tulisi määrittää oma painonsa. Ehkä kaaren paino voisi olla vaikkapa pikselin tummuus? Tämä olisi helppo toteuttaa. Jos jokaisen värille haluaa oman painon, niin tähän voisi tarvita hajautustaulun toteuttamista.

Vertailuun voi lisätä lisää algoritmeja. Painottomissa verkoissa voisi testata lisäksi jump point search -algoritmia. Painollisissa verkoissa voisi testata lisäksi D* tai Fringe-search -algoritmeja.

Voisiko vertailuun lisätä kekoja? Pairing-keko on yksinkertaistettu fibonaccikeko, jolla on pienemmät vakiokertoimet. Sen suorituskykyä voisi olla mielenkiintoista vertailla.

Bellman-Ford -algoritmia ei vertailtu, sillä se oli aivan liian hidas. Tuloksista ei voinut ottaa mitään keskiarvoa. Mihin Bellman-Ford -algoritmia voisi vertailla? Verkossa, jossa on negatiivisia

kaaripainoja, toimivia algoritmeja on vähän. Floyd-Warshall ja Johnson -algoritmien toimintaa olisi voinut vertailla, sillä ne molemmat selvittävät pienimmät polut kaikkien kaaripainojen välillä.

Algoritmien suorituskykyä Javan toteutuksilla ei ollut mielekästä testata, sillä Priorityqueue ei tue decreaseKey-operaatiota. PriorityQueue-luokkaa käytettäessä algoritmit olivat aivan liian hitaita. Toteutettuja tietorakenteita sen sijaan voisi vertailla. PriorityQueue -luokkaa voisi verrata esimerkiksi toteutettuun binaarikekoon. Insert ja delMin-operaatioita olisi mahdollista vertailla.

Lähteet

1. https://en.wikipedia.org/?title=Dijkstra%27s_algorithm, katsottu 14.6.2015
2. https://en.wikipedia.org/?title=Fibonacci_heap, katsottu 14.6.2015
3. https://en.wikipedia.org/wiki/Binary_heap, katsottu 14.6.2015
4. https://en.wikipedia.org/wiki/A*_search_algorithm, katsottu 14.6.2015
5. https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm, katsottu 14.6.2015
6. https://en.wikipedia.org/wiki/Breadth-first_search, katsottu 14.6.2015