Petri Mäkinen
014027765

Tiralabra toteutusdokumentti

BFS ja a –tähden toteuttamista varten tein ensin graph nimisen luokan, jonka avulla voidaan tehdä matriisi johon on tallennettu solmuolioita. Näillä solmuolioilla on aina id, joka on kullekin yksilöllinen ja riippuu olion paikasta matriisissa, sekä koordinaatit. Seuraavassa esimerkkimatriisi verkosta, jota on käytetty jonkin verran yksikkötesteissä.

1	null	3	4	5
6	null	8	9	10
11	null	null	null	15
16	null	18	19	20
21	22	23	24	25

Tässä matriisin soluissa joissa on numero, on siis verkon solmu, jonka id:tä numero esittää. Solut joissa on null ovat ”seiniä” eli niiden läpi verkossa ei pääse kulkemaan. Verkossa esimerkiksi solmun 4 naapureita ovat 3, 5 ja 9. Diagonaalisesti verkossa ei siis voi kulkea.

Kuten määrittelydokumentissa mainittiin pitäisi tekemieni BFS ja a-tähti toteutuksien toimia aikavaativuuksilla O(|V| + |E|) ja O((|V| +|E|) log|V|) ja tilavaativuudella O(|V|) kummankin algoritmin osalta. Tämä tavoite saavutettiin mikä perustellaan seuraavassa pseudokoodia apuna käyttäen. Otetaan ensin käsittelyyn BFS algoritmi.

BFS käy solmuja läpi nimensä mukaisesti leveyssuuntaisesti. Ensin käydään alkusolmun naapurisolmut. Sitten alkusolmun naapurien naapurit ja niin edelleen. Seuraavassa pseudokoodi:

```
BFS(alkusolmu, maalisolmu)
        avoimet                         // lista solmuista joita ollaan käymässä seuraavaksi läpi
        kaydyt;                         // lista solmuista jotka on jo käyty läpi
        edellinen[];                    // taulukko joka kertoo mistä solmusta ollaan tultu solmuun i. Solmu i indeksissä i-1
        avoimet.lisaa(alkusolmu)
        while(avoimet ei ole tyhjä)
                solmu = avoimet.pollFirst();        //otetaan avoimet listasta ensimmäinen ja poistetaan se avoimista
                kaydyt.lisaa(solmu)
                if(solmu == maalisolmu)
                        return ratkaisu(edellinen, solmu, alkusolmu);
                lisaa(solmu.getNaapurit(), avoimet, kaydyt, edellinen, solmu);
        return (”ei ratkaisua”);

Lisaa(naapurilista, avoimet, kaydyt, edellinen[], solmu)        //solmu on solmu jonka naapureita ollaan käsittelemässä
        for(käy naapurilista läpi)
                naapuri = naapurilista.get(i);
                if(naapuri ei ole avoimessa listassa && naapuri ei ole käytyjen listassa)
                        avoimet.lisaa(naapurilista.get(i))
                        edellinen[naapuri.getId() -1] = solmu.getId();      //laitetaan muistiin mistä solmusta tultiin

Ratkaisu(edellinen[], solmu, alkusolmu)

        kaanteinenreitti;               //pino johon talletetaan reitti edellinen taulukosta
        kaanteinenreitti.push(solmu)    //laitetaan viimeinen solmu ensin pinoon
        reitti[];                       //taulukko johon tallennetaan löydetty reitti
```

Petri Mäkinen
014027765

```
while(solmu != alkusolmu)
        stack.push(edellinen[solmu-1]);
while(kaanteinenreitti ei ole tyhjä)
        reitti[i] = kaanteinenreitti.pop();
        i++;
return reitti;
```

Algoritmi siis lähtee liikkeelle lisäten avoimet listaan alkusolmun jonka jälkeen sen naapurit lisätään avoimeen listaan ja alkusolmu itse läpikäytyjen listaan. Aina ennen solmun lisäämistä läpikäytyihin se poistetaan avoimesta listasta. Näin jatketaan eikä avoimeen listaan ikinä lisätä enää jo läpikäytyjä solmuja. Näin jokainen solmu käydään siis (pahimmassa tapauksessa) täsmälleen kerran läpi kunnes päästään maalisolmuun, jonka jälkeen käydään vielä ”edellinen” taulukko läpi. Taulukossa ei kuitenkaan voi olla kuin korkeintaan |V| solmua.

Olen toteuttanut avoimet listan kahteen suuntaan linkitetyllä listalla, jolloin listaan lisääminen ja ensimmäisen alkion hakeminen ja poistaminen ovat vakioaikaisia operaatioita. Itse solmut on talletettu matriisiin solmuolioina. Matriisissa solmun yläpuolella, oikealla, vasemmalla sekä alapuolella olevat solmuoliot ovat solmun naapureita, joten ne saadaan selvitettyä myöskin vakioajassa.

Toteutukseni käyttää lisäksi kahta totuusarvomatriisia joihin on tallennettu tieto siitä kuuluuko solmu käytyihin vai avoimiin. Tämä tehtiin siksi että saatiin toteutettua vakioaikaisesti operaatio joka tarkistaa löytyykö tietty solmu avoimista tai jo käydyistä listoista. Tämä olisi voitu toteuttaa pienemmällä tilavaativuudella esimerkiksi käyttäen linkitettyä hajautustaulua, jolloin ylimääräistä totuusarvomatriisia ei olisi tarvittu vaan avoimet ja käydyt listat olisivat riittäneet. Toki tällöinkin tilavaativuus olisi edelleen ollut suoraan suhteessa solmujen määrään.

Sitten a-tähti algoritmin perusteluun:

A-tähti algoritmi käy solmuja läpi siinä järjestyksessä mikä solmu kulloinkin näyttää parhaalta. Solmujen paremmuuden ratkaisee kahden arvon summa. Nämä arvot ovat:

tähän astinen polkukustannus (kustannus lähtösolmusta tähän solmuun polkua pitkin jota nyt ollaan tutkimassa) = g(n).

heuristinen arvo tästä solmusta maaliin. (Tämä on siis vain arvio. Tärkeää on että tämä arvio ei yliarvioi todellista kustannusta, vaan on korkeintaan yhtä suuri kuin todellinen kustannus. Toinen tärkeä asia on että arvio on nopea laskea) = h(n).

Koska olen toteuttanut verkon niin että solmujen naapureita ovat vain suoraan solmun ylä- ja alapuolella sekä oikealla ja vasemmalla olevat solmut lasketaan myös heuristiikka ns. manhattan etäisyytenä. Toisinsanoen siis summa siitä kuinka monta riviä verkkomatriisissa on liikuttava ylös tai alas ja kuinka monta saraketta oikealle tai vasemmalle maalisolmun saavuttamiseksi. Heuristiikkaa laskettaessa ei välitetä siitä onko tälläisellä suorimmalla mahdollisella reitillä seiniä vai ei.

Petri Mäkinen
014027765

Algoritmin alkuosa on täsmälleen samanlainen kuin bfs algoritmissa (edellisellä sivulla bfs otsikon alla) lukuunottamatta sitä että avoimet lista on totetutettu linkitetyn listan sijaan minimikeolla. Mennään siis suoraan lisää metodiin joka on hieman erilainen a-tähdelle.

```
Lisää(naapurilista, avoimet, kaydyt, edellinen[], solmu)
        for(käy naapurilista läpi)
                if(naapuri on käytyjen listassa  && solmun g(n) +1 >= naapurin g(n))
                        jatka seuraavasta naapurista
                if(naapuri ei ole avoimessa listassa ||  solmun g(n) +1 < naapurin g(n))
                        laske ja aseta naapurin heuristinen arvo;
                        aseta naapurin polkukustannukseksi (g) solmun polkukustannus (g).
                        laita naapuri oikealle paikalle avoimien listassa mikäli tarpeen tai lisää se listaan jos se ei ole vielä siellä;
                        edellinen[naapuri.getId() -1] = solmu.getId();          //laitetaan taas muistiin mistä solmusta tultiin
```

Lisää metodi on kutakuinkin sama kuin Wikipedia artikkelin (http://en.wikipedia.org/wiki/A*_search_algorithm) pseudokoodi (siitä kohdasta alkaen jossa aletaan käydä läpi käsitellyn solmun naapureita), mutta if lauseissa ei ole otettu vertailuun mukaan heuristiikkaa, sillä se pysyy kullekin solmulle aina samana riippumatta siitä mistä solmusta siihen on tultu.

Kuten BFS algoritmissa, tässäkin oikea reitti saadaan lopuksi edellinen –taulukosta edellä esitetyn ratkaisu metodin avulla. Kuten tietorakenteet ja algoritmit –kurssin materiaalissa on esitetty, on a tähden aikavaativuus O((|V| +|E|) log|V|). Tässä log|V| tulee siitä että avoimet listassa olevat solmut pitää pitää järjestyksessä niin että listaan lisätyistä solmuista laitetaan ensimmäiseksi se jonka heuristisen arvon ja polkukustannuksen summa on pienin. Solmut pidetään minimikeossa joten lisäysoperaation sekä solmujen pitämisen oikeassa järjestyksessä solmun g(n) arvion muuttuessa aikavaativuus on log|n|, missä n on solmujen määrä keossa.

Toteutuksessa on toki joitain puutteita vaikka asymptoottisiin aika- ja tilavaativuuksiin päästiinkin. Toteutus ei ole loppuun asti optimoitu. Esimerkiksi toteutukseni tilavaativuus on turhan iso sillä esim. käsiteltyjen ja avoimien listat on nyt monessa kohdin toteutettu totuusarvomatriiseina jotka ovat aina yhtä isot kuin täyden verkon solmujen määrä. Tätä saataisiin toki optimoitua esimerkiksi käyttämällä hajautustauluja, jolloin tilaa ei aina tarvittaisi niin paljoa. Tilavaativuuden kertaluokkaan tämä ei kuitenkaan vaikuttaisi vaan nykyinen totetutus suurentaa ratkaisun tilavaativuutta vain vakiokertoimen verran. Samoin esimerkiksi toteuttamani arraylist ja hajautustaulu eivät pienennä käyttämäänsä taulukkoa vaikka niistä poistettaisiin alkioita. Tämä on toinen esimerkki siitä että tilankäyttöä ei ole loppuun asti optimoitu. Tämä toisaalta hieman parantaa algoritmien nopeutta kun taulukoita ei tarvitse kopioida niin usein.

Työn graafisen käyttöliittymän teossa on käytetty surutta javan valmiita tietorakenteita ja muita luokkia, koska graafinen käyttöliittymä ei ollut varsinainen vaatimus. Käyttöliittymä koodi on myös muokattu internetistä löytyneestä koodista (linkki window luokassa) eikä koodin selkeyteen (käyttöliittymäpuolella) tai käyttöliittymän käyttäjäystävällisyyteen ole juuri kiinnitetty huomiota.

Yksi huomioitava seikka on että tekemäni hajautustaulu kaatuu ja kaataa ohjelman jos sitä yritetään tehdä liian isolla alkukapasiteetilla. Omalla koneellani raja oli 2^28. Tällöin loppui muisti. Sama ongelma tuli tosin vastaan kokeiltaessa tehdä samaa javan valmiin kaluston hajautustaululla, joten tätä ei varmaan pysty edes järkevästi kiertämään.