Tampereen ammattikorkeakoulu

A*:n ja Flowfieldin vertailu Unity-pelimoottorissa

Oskari Niemelä

OPINNÄYTETYÖ
Huhtikuu 2022

Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

NIEMELÄ, OSKARI:
A*:n ja Flowfieldin vertailu Unity-pelimoottorissa

Opinnäytetyö 34 sivua, joista liitteitä 6 sivua
Huhtikuu 2022

Reitinlöytö on nimensä mukaisesti reittien etsintää. Etsintä toteutetaan matemaattisesti laskien. Tässä opinnäytetyössä esiintyvät reitinlöytöalgoritmit laskevat reitit graafien avulla. Taustatietona käydään läpi Dijkstran algoritmin toiminta. Reitinlöydöllä on monia applikaatioita. Tässä työssä vertaillaan kahta reitinlöytöalgoritmia erityisesti strategiapelien näkökulmasta. Työssä käytetään laajasti alalla käytössä olevaa Unity-pelimoottoria algoritmien implementointiin sekä testaukseen.

Reitinlöytö on tärkeä osa peliohjelmointia. Reitinlöytöä täytyy käyttää, kun pelin sisäiset hahmot navigoivat pelimaailmaa itsenäisesti, joko tekoälyn tai sitten pelaajan käskyttämänä. On erityisen tärkeää, että reitinlöytö tehdään tehokkain keinoin, jotta pelien kuvataajuus säilyy pelin ohjautuvuuden kannalta hyväksyttävällä tasolla.

Työn kaksi algoritmia, A* sekä Flowfield, valittiin niiden miltei päinvastaisten käyttötarkoitusten takia. Flowfield on tehty suurien hahmomäärien liikuttamiseen pelikentällä, minkä vuoksi se laskee koko kentän kattavan reitistön aina kun algoritmia käytetään. A* on tehty laskemaan yksi reitti pisteestä A pisteeseen B, jolloin sen täytyy laskea jokaiselle liikkuvalle yksikölle oma reittinsä päämäärään. A* sekä Flowfield käyttävät pohjanaan Dijkstran algoritmia.

Algoritmien vertailua varten määriteltiin testejä, joiden tarkoituksena oli tarkastella algoritmien skaalautuvuutta eri tilanteissa. Toteutusvaiheessa haastavimmaksi osoittautui algoritmien toteutus pelimoottorissa sekä tulosten ulossaanti.

Työn tulokset vastasivat odotuksia: algoritmien skaalautuvuus riippui eri asioista. A* skaalautui hahmojen määrän sekä laskettavan reitin pituuden mukaan. Flowfield skaalautui kartan koon sekä monimutkaisuuden mukaan. Pohdintoihin lisättiin vielä huomautus reitinlöydön muista teknisistä osista, jotka tulee ottaa huomioon reitinlöytöä implementoidessa.

Asiasanat: reitinlöytö, A*-algoritmi, flowfield, unity, Dijkstra

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering

NIEMELÄ, OSKARI:
A Comparison of the A* and Flowfield Algorithms in the Unity game engine

Bachelor's thesis 34 pages, appendices 6 pages
April 2022

This thesis examines two path finding algorithms, A* and Flowfield especially from the perspective of strategy games. Pathfinding is as the name implies, the process of finding paths on a map, this search for viable paths is done with mathematical calculation. Though this study examines the aforementioned algorithms and pathfinding in general mostly from a game programming perspective, pathfinding has many other real-world applications as well. The study uses the Unity game engine, which is widely used in the videogame industry.

Pathfinding is a key part in modern game programming. As it is performed frequently and can be quite calculation heavy it is paramount that it be done in a cost-effective way while balancing performance with results. The frames-per-second of a game must be kept high enough so as not to negatively affect the players ability to control the game and understand the state the game is in.

The two algorithms examined in the study, A* and Flowfield were chosen because of their diametrically opposed use cases. A* is used for calculating a singular path from point A to point B, in which case a separate path calculation is needed for each unit we command in a game even if their destination is the same. The Flowfield algorithm calculates routes from every point in the navigable map to the end point, this means that every unit can follow the route closest to them and there is no need to repeat calculations.

To measure the performance of these algorithms in different circumstances that are favorable to one or the other, tests were devised to check how the algorithms scale in these circumstances. The most difficult part when implementing the coding in this study was to be implementing the algorithms into the Unity game engine and determining how to extract the performance results out of Unity.

The results were as expected, the algorithms scaled with different parameters that the tests had. A* scaled according to the unit count as well as the length of the calculated route. Flowfield scaled according to the size of the map, as well as the complexity of said map.

Key words: pathfinding, A*-algorithm, flowfield, unity

SISÄLLYS

1 JOHDANTO

Videopeli-ala on miljardien eurojen arvoinen toimialue (Clement 2021), johon osallistuvat pelinkehitystiimit, jotka vaihtelevat yksittäisistä henkilöistä koostuvista tiimeistä, sadoista tai jopa tuhansista ihmisistä koostuviin suuren pelinkehitystalon tiimeihin. Suurin osa peleistä, ovat ne sitten kehittäneet yksittäinen henkilö tai isompi joukko, täytyy kuitenkin täyttää tietyt ehdot, että asiakkaat pystyvät ja haluavat pelata tuotteita, jotka laitetaan markkinoille. Yksi ehto, joka täytyy täyttää tuottojen maksimoimiseksi, on pelin sujuvuus teknisellä tasolla, jos kuvataajuus (framerate) on liian alhainen ei peliä voi pelata ollenkaan. Yksi osa-alue, johon suurin osa peleistä nykyaikana täytyy kiinnittää huomiota, on reitinlöytö, pelimaailmojen sisäiset oliot eivät osaa navigoida maailmoja ilman reitinlöytöä, ja siksi on tärkeää toteuttaa reittien löytö tehokkaasti.

Tässä työssä tullaan tutkimaan reitinlöytöalgoritmeja, erityisesti strategiapelien ja tehokkuuden kannalta. Tehokkuudella tarkoitetaan tässä työssä lähinnä suoritin aikaa eli aikaa, joka kuluu, kun suoritin ajaa tiettyä koodin osaa, eli tässä työssä reitinlöytöalgoritmia. Esimerkkinä jos koodin suoritinaika on 2 ms, suorittimella kuluu 2 ms ajaa koodin sisältö. Jos suoritinaika nousee maksimi kuvataajuus joka voidaan tavoittaa pienenee. Strategiapeleissä voi olla satoja tai jopa tuhansia yksittäisiä hahmoja, joiden täytyy navigoida kenttää, jossa on yleisesti paljon esteitä, ja kaikki täytyy tehdä reaaliajassa. Algoritmien vertailua varten työhön kuului kahden yleisen reitinhakualgoritmin implementointi Unity-pelimoottorissa, ja niiden tehokkuuden vertailu Unityn sisäisten työkalujen avulla.

2 TAUSTATIETOA

2.1 Unity-pelimoottori

Unity on 2005 vuonna julkaistu omisteinen (proprietary) pelimoottori, jota on lähiaikoina alettu käyttämään myös muihin tarkoituksiin, kuten simulaatioiden ja 3d-elokuvien tekoon. Unity-pelimoottorin on kehittänyt Tanskan Kööpenhaminassa perustettu Unity Software Inc (entinen Over the Edge Entertainment) (Haas 2014.) Unity-pelimoottori valittiin tämän työn teknistä toteutusta varten sen helppokäyttöisyyden, laajan dokumentaation sekä yhteisön tuottaman tiedon vuoksi. Unity tarjoaa myös Profiler työkalun, jolla pystytään testaamaan tehokkuutta koodia ajaessa.

Pelimoottorin sisällä käytetään C#-ohjelmointi kieltä, mutta sen lisäksi Unity käyttää runsaasti omia kirjastojaan. Tämän työn koodausosio toteutetaan kokonaisuudessaan Unityn sisällä.

2.2 Reitinlöytö yleisesti

Reitinlöydöllä (pathfinding) tarkoitetaan algoritmeja, joiden tarkoitus on löytää reitti kahden pisteen välillä. Ideaalissa tapauksessa löydetty reitti on lyhin mahdollinen, mutta varsinkin pelien tapauksessa reitinlöytö yleensä lopetetaan, kun on löydetty yksi validi reitti, tämä tehdään tehokkuussyistä.

Reitinlöydölle on kaupallisia ja henkilökohtaisia käyttöjä, joihin lukeutuu karttanavigaatio, lentokoneiden kulkureittien laskenta ja verkkojen sisäinen reititys. Tällaisille “Shortest-Path” eli lyhin reitti -ongelmille on kehitetty ratkaisuja jo vähintäänkin 1800-luvun puolesta välistä, mutta pitkään käytettiin heuristisia, ei optimaalisia menetelmiä ongelman ratkaisuun (Schrijver 2012). Näihin eri reitinlöydön applikaatioille on omintakeisia ratkaisuja, jotka eivät välttämättä ole julkista tietoa, vaan yhtiöiden sisäisiä työkaluja.

Kaikki nykyiset reitinlöytöalgoritmit perustuvat pitkälti painotettuihin graafeihin, joissa solmujen välisiin kaariin on kirjattu solmujen välisten reittien hinta numeraalisesti, tällöin voidaan sitten kaikista pisteistä laskea reitti ja reitin hinta. Halvin reitti valitaan sitten laskettujen reittien joukosta. Reitin hintaan voidaan sisällyttää periaatteessa mitä vain, jota halutaan ottaa huomioon reitinlöydössä. Yleensä halutaan ainakin mitata aika, joka kuluu reittiä kulkiessa, joten reitin pituus otetaan huomioon. Graafin rakenne käydään läpi tarkemmin kappaleessa 2.3.1.

2.3 Reitinlöytö peleissä

Reitinlöytö peleissä on erittäin tärkeä osa niiden rakennetta ja pelattavuutta, reitinlöytöä tapahtuu aina kun pelin sisäiset agentit navigoivat pelimaailmaa. Agentilla tarkoitetaan yksittäistä pelinsisäistä hahmoa, joka navigoi maailmaa, ja jolle täytyy löytää reitti. Jos reitinlöytö ei ole tarpeeksi tehokas, se voi johtaa pelin kuvataajuuden tippumiseen ja tämä tekee pelin ohjaamisesta vaikeampaa, kun pelille annettavat inputit eivät rekisteröidy tai ne rekisteröityvät samaan aikaan. Vaikka reitinlöytö olisikin tehokasta, jos se on liian epätarkkaa tai jos peli ei ota huomioon mitä tapahtuu, kun agentit törmäävät toisiinsa tai muihin liikkuviin esteisiin, voivat agentit jäädä jumiin peligeometriaan. On vaikea mitata tarkalleen kuinka paljon ärsyttäviä tekijöitä pelaaja voi kohdata ilman, että hän lopettaa pelin pelaamisen. Maalaisjärjellä tehty oletus on, että mahdolliset turhauttavat tekijät tulisi minimoida. Reitinlöytö on tällöin erittäin tärkeä sekä pelin toimivuuden että pelattavuuden kannalta, joilla on vaikutus pelin rahalliseen tuottavuuteen.

Reitinlöytöalgoritmeja käytetään jokaisessa pelissä, joissa pelimaailman hahmojen täytyy navigoida maailmaa, joissain yksinkertaisissa peleissä kuten vaikka alkuperäinen Super Mario Brothers vastustajat eivät tee reitinetsintää, vaan niillä on hyvin yksinkertainen koodattu reitti mitä kulkevat, esimerkiksi ”kävele eteenpäin ja jos osut seinään, niin käänny toiseen suuntaan”. Nyrkkisääntönä jokaisessa pelissä, jossa agentit liikkuvat ympäristössä ja väistävät vastaan tulevia esteitä tarvitaan reitinlöytöä. Näistä voisi olla esimerkkeinä pelit kuten GTA V, Heroes of Might and Magic 3 ja Age of Empires 2. Mikään näistä esimerkkipeleistä ei ole koodattu Unityllä, mutta reitinlöydön periaatteet ovat samat toteutetaan se sitten minkä tahansa pelimoottorin sisällä.

Dijkstran algoritmi on kenties parhaiten tunnettu, mutta se ei ole laajalti käytössä pelien reitinlöydössä, se on liian epätehokas siihen käyttöön. Kaikista laajimmassa käytössä on eräänlainen muunnelma Dijkstran algoritmista nimeltään A*-algoritmi, joka käyttää heuristiikkaa ohjaamaan algoritmin kulkua. A*-algoritmista on myös tehty muunnelmia, jotka antavat joitain etuja verrattuna A*:en.

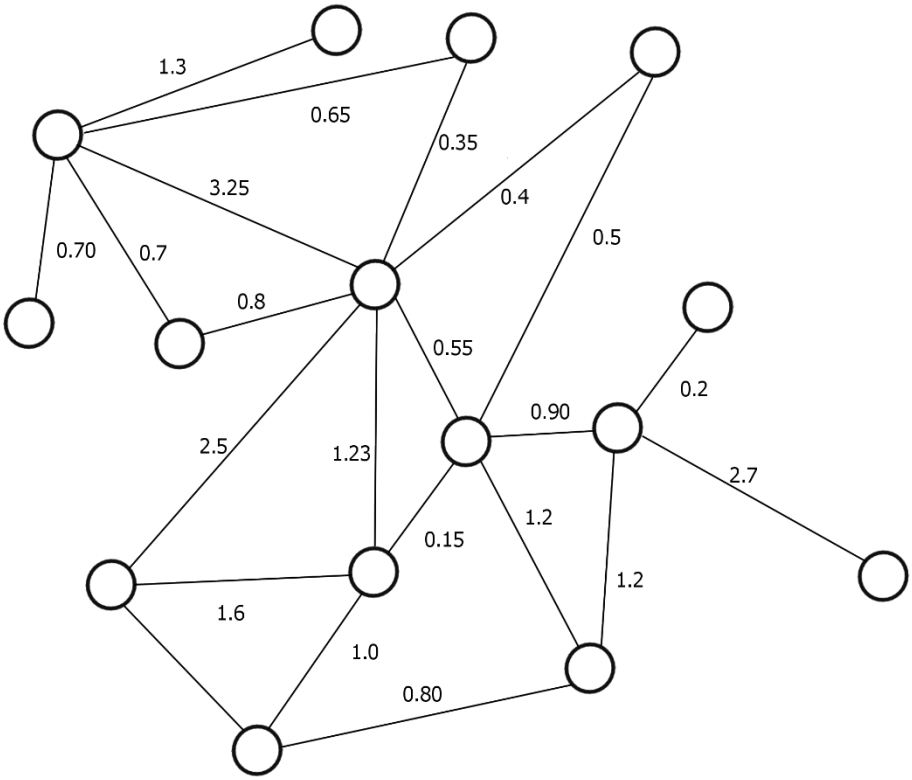
2.3.1 Reitinlöydössä käytettävän graafin rakenne

Dijkstran algoritmi ja siitä periytetyt muunnokset peleissä eivät toimi pelkällä pelikentän geometrialla, vaan pelikenttä täytyy esittää yksinkertaistetussa muodossa, jotta algoritmit pystyvät tehokkaasti löytämään hyvän reitin kentän läpi (Funge & Millington 2009, 198). Pelien reitinlöytöalgoritmit ovat graafeilla toimivia algoritmeja, edellyttäen että peli maailmalle on graafeilla toteutettu osuus, jonka kautta algoritmit toimivat. Tämä täytyy tehdä hyvin, jotta kriittistä informaatiota ei katoa (Funge & Millington 2009, 198). Graafirakenteet voidaan generoida koodilla, tai käsin.

Graafirakenteet ovat matemaattisia rakenteita, joita voidaan myös esittää diagrammeilla helposti. Graafit rakentuvat kahdesta eri osasta, solmuista (”node”), jotka ovat pisteitä diagrammilla, ja kaarista (”edge”), jotka yhdistävät solmuja. Jokainen solmu graafissa esittää pistettä pelimaailmassa, ja reitin löydyttyä hahmojen liikkumiset tapahtuvat näiden pisteiden välillä. Reitinlöytöön ei kuitenkaan riitä normaali graafirakenne, tarvitaan painotettu graafirakenne. Painotetussa graafissa kaarille on annettu numeroarvo. Kaaren numero arvo esittää hintaa, joka kaarella on, eli toisin sanoen mitä suurempi arvo kaarelle annetaan, niin sitä vaivalloisempaa kaaren yhdistämien solmujen välinen reitti on kulkea. Kaarien hinta arvoon voidaan ottaa huomioon mitä vain jonka kehittäjä katsoo oleelliseksi, se voi esimerkiksi ottaa huomioon vain puhtaasti etäisyyden solmujen välillä tai sitten ottaa huomioon muita seikkoja kuten maaston, joka voi hidastaa liikkumista solmujen välillä. Tätä työtä varten voidaan ajatella, että graafit toteutetaan kahdella luokalla, solmuilla sekä kaarilla. Solmuilla on tiedossaan lista kaarista, jotka lähtevät siitä solmusta, ja kaaret sisältävät tiedon mihin solmuun kaarella pääsee,

mistä solmusta kaari tulee ja mikä on sen kaaren hinta. Kaaren hinta ei voi olla negatiivinen. (Funge & Millington 2009, 198–200.)

Kuvio 1 esittää mielivaltaisesti rakennettua esimerkkiä graafirakenteesta, joka on painotettu.



KUVIO 1. Painotettu graafi.

Algoritmit toimisivat, vaikka käytettäisiin ainoastaan painotettuja graafeja, monesti kuitenkin käytetään painotettuja ja suunnattuja graafeja. Suunnatuissa graafeissa jokaisella kaarella on oma suuntansa. Ei-suunnatussa graafissa voi olla yhteys solmujen A ja B välillä ja tällöin A:sta pääsee B:hen ja päinvastoin. Suunnatuissa graafeissa jokaisella kaarella on suunta. Kaari, joka on solmujen A ja B välillä on joko kaari, joka vie solmusta A solmuun B tai solmusta B solmuun A. Jos halutaan että solmujen välillä päästään liikkumaan edestakaisin täytyy määrittää kaksi kaarta yksi, joka osoittaa A:sta B:hen ja toinen, joka osoittaa B:stä A:han. Suunnattuja graafeja käytetään, koska se antaa mahdollisuuden tehdä

yksisuuntaisia reittejä ja tehdä saman reitin eri suunnista eri hintaisia, tämä mahdollistaa erilaisten yksisuuntaisten esteiden ja hidasteiden mallintamisen graafeilla, esimerkiksi jos kävellään ylämäkeen. (Funge & Millington 2009, 202.) Tässä työssä pelikenttä on jaettu neliöillä koordinaatistoksi ja jokainen yksittäinen graafin solmu vastaa yhtä neliötä pelikentän koordinaateissa. Työssä käytetty graafi on painotettu, mutta ei suunnattu.

Pelin tekijät voivat lisätä ja muokata pelikarttaan kartan maastoa simuloivia muuttujia, yleensä reitinlöydössä on mukana korkeuserot solmujen välillä, mutta esimerkiksi maastoa voidaan simuloida antamalla jokaiselle maastotyypille jonkinlainen kerroin, jolla kerrotaan solmuun tulevien reittien arvot mallintamaan eri maastotyyppien vaikutusta kulku nopeuteen. Voi olla myös erityyppisiä reitinlöytö agentteja kartalla, esimerkiksi lentävät agentit eivät välitä maastosta eikä korkeuseroista, tämä myös tietenkin vaikuttaa vahvasti niiden agenttien reitinlöytöön.

Reitinlöydössä voidaan käyttää myös klusterointialgoritmeja, jotta reitinlöytö graafeihin saadaan hierarkkisuutta (Chen, Li & Li 2011). Klusterointialgoritmit, reitinlöydön parissa, toteuttavat graafin solmujen ”rypästämistä”. Jos graafi esittää taloa, joka sisältää monia huoneita, niin eräs tapa rypästää graafi olisi valita kaikki solmut, jotka olisivat esimerkiksi olohuoneessa. Uudessa ylemmän tason graafissa olohuoneen solmut pistettäisiin yhden solmun alle. Näin tehtäisiin jokaiselle muullekin talon huoneelle, jonka jälkeen tämä korkeamman tason graafi esittäisi kaikkia talon huoneita, ei pelkästään jokaista reitinlöytöpistettä, joka on taloon asetettu. Yksityiskohtaisempi selitys klusterointi algoritmien toiminnallisuudesta sekä mahdollisista tehokkuusoptimoinneista on Pohdinnat alaotsikon alla.

2.3.2 Dijkstran algoritmi

Dijkstran algoritmi on nimetty sen keksijän, Edsger Dijkstran, mukaan. Algoritmi kehitettiin ratkaisemaan ”Shortest-Path” -ongelma matemaattisissa graafirakenteissa, jossa aloitus solmusta tulee laskea lyhin reitti jokaiseen toiseen solmuun graafissa, toisin kuin pelien reitinlöydössä, jossa halutaan etsiä reitti aloituspisteen ja loppupisteen välillä. Vaikkakin Dijkstra on huono algoritmi pelikäyttöön,

sen pohjalta on luotu kaikki muut pelialalla laajasti käytössä olevat reitinlöytömenetelmät. Dijkstran käyminen läpi tuo meille ymmärryksen miten A* ja sen muunnelmat parantavat ja nopeuttavat Dijkstraa. (Funge & Millington 2009, 204.)

Dijkstran pohjana voidaan käyttää painotettua graafia, joka joko on tai ei ole suunnattu. Se toimii leviämällä aloitus solmusta kaukaisempiin solmuihin, aina merkaten mistä mihinkä solmuun on tultu, ja saapuessaan maali solmuun se seuraa näitä merkkejä takaisin alku solmuun samalla ottaen muistiin kaikki ne kaaret, jota se käytti sitä reittiä varten. Laskettu reitti koostuu kaarista, joita täytyy seurata maaliin asti, ei solmuista.

Dijkstra lisää graafirakenteen solmuun kaksi muuttujaa. ”Cost-so-far” kertoo kuinka paljon siihen tultaessa kaaret ovat yhteensä maksaneet. Lisäksi toinen muuttuja kertoo, mistä solmun naapureista kyseiseen solmuun on tultu säilyttämällä muistissa kaaren, jota käytettiin kyseiseen solmuun pääsemiseksi. Tätä muuttujaa kutsutaan Reitti-muuttujaksi. Algoritmi pitää myös kahta eri listaa solmuista: avoimet solmut ja suljetut/prosessoidut solmut. Algoritmi toimii iteratiivisesti prosessoiden yhden solmun kerrallaan. Prosessoitavaa solmua kutsutaan tämänhetkiseksi solmuksi (current node). Kutsutaan kaaren toisessa päässä olevaa solmua määränpää solmuksi (m-solmu).

Algoritmi aloittaa asettamalla aloitussolmun ”Cost-so-far” -muuttujan olemaan yhtä suuri kuin nolla, ja alkaa sitten prosessoimaan aloitussolmua. Tämänhetkinen solmu prosessoidaan käymällä läpi yksitellen kaaret, jotka sillä ovat muistissa. Riippuen kaaren m-solmun tilasta kolme eri toimenpidettä voi tapahtua:

1. Jos m-solmu on listattuna algoritmin suljettujen solmujen listalla, ei tehdä mitään ja siirrytään tarkastelemaan seuraavaa muistissa olevaa kaarta.

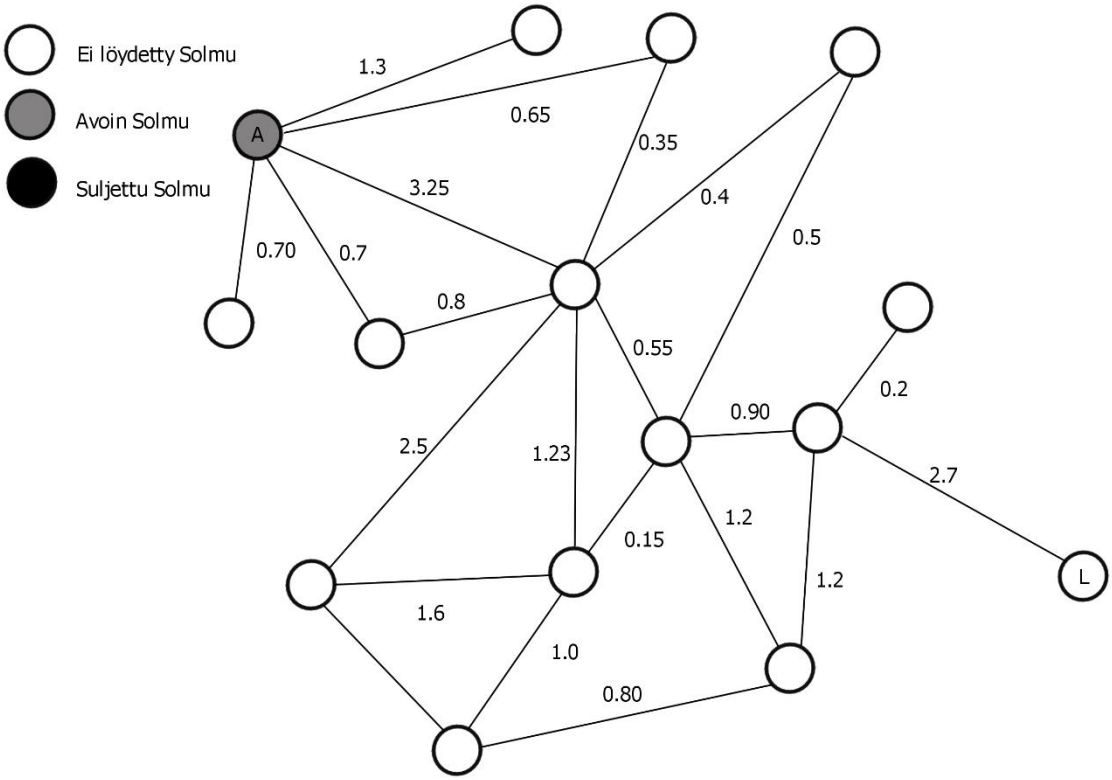
2. Solmu ei ole avoimien solmujen eikä suljettujen solmujen listalla niin m-solmun ”Cost-so-far” muuttuja asetetaan olemaan yhtä suuri kuin tämänhetkisen solmun ”Cost-so-far” + kaaren hinta, m-solmun reitti muuttuja asetetaan olemaan kaari jota tällä hetkellä tarkastellaan, sekä m-solmu lisätään avoimien solmujen listaan.

3. Jos m-solmu on avoimien solmujen listalla, eli sille on laskettu jo ”Cost-so-far”, sitten lasketaan ”Cost-so-far” tämänhetkisen solmun avulla, ja sitä verrataan m-solmun ”Cost-so-far” muuttujaan. Jos uusi laskettu ”Cost-so-far” muuttuja on alempi arvoinen kuin nykyinen, alempi arvoinen ”Cost-so-far” asetetaan vanhan tilalle ja reitti muuttuja vaihdetaan tämänhetkiseen kaareen.

Kun kaikki tämänhetkisen solmun kaaret on käyty läpi, kyseinen solmu asetetaan suljettujen solmujen listalle ja algoritmi valitsee avoimien solmujen listalta solmun, jolla on pienin ”Cost-so-far” arvo (Funge & Millington 2009). Tämä prosessi on ”ahne”, eli se valitsee aina sillä hetkellä pieni hintaisimman solmun seuraavaksi prosessoitavaksi. Tästä seuraa, että jo suljetuille solmuille ei tulla ikinä löytämään optimaalisempaa reittiä eli niitä ei tarvitse ottaa mitenkään huomioon sen jälkeen, kun ne ovat suljettu. Myöhemmin läpikäytävässä A*-algoritmissa käytetään erilaista arvojärjestystä seuraavaksi prosessoitavalle solmulle, jolloin jo suljetut solmut voivat mahdollisesti palata avoimien solmujen listalle uudelleen prosessoitavaksi (Funge & Millington 2009). Jos kohdesolmua ei ole löytynyt ja avoimien solmujen lista on tyhjä, mitään mahdollista reittiä kohteeseen ei ole, joten algoritmi lopettaa toimintansa.

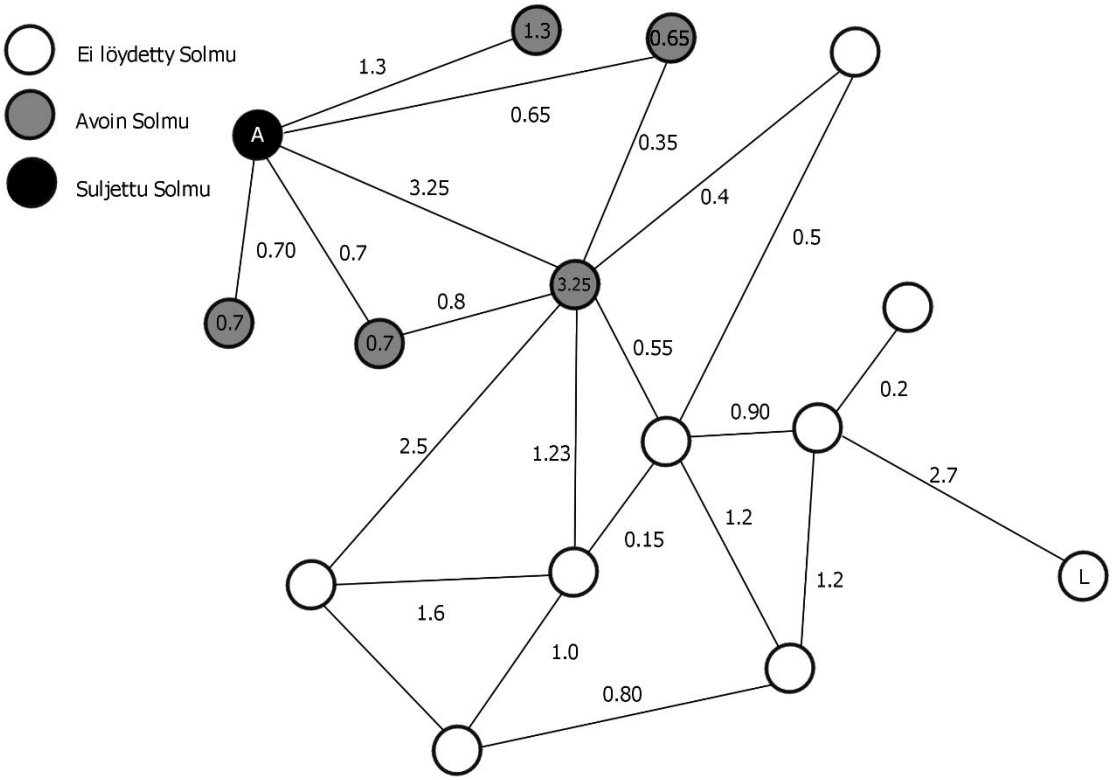
Kun kaaret on käyty läpi, algoritmi seuraa kohdesolmusta alkaen solmujen Reitti muuttujia takaisin aloitussolmuun, kooten listan seurattavista kaarista samalla. Lista kaarista on sitten reitti, jota täytyy seurata, jotta päästään aloituksesta kohteeseen.

Dijkstra on epätehokas pelikäyttöön, koska se ei ota kantaa, eikä yritä ohjautua mitenkään kohdesolmun suuntaan, vaan se yksinkertaisesti käy läpi solmuja, kunnes kaikki solmut on käyty läpi. Kuvio 2 näyttää mielivaltaisesti valitun alku (A) ja loppu (L) pisteen sekä Dijkstran algoritmin alkuasetelman näillä valituilla pisteillä aiemmin esitellyllä painotetulla graafilla.



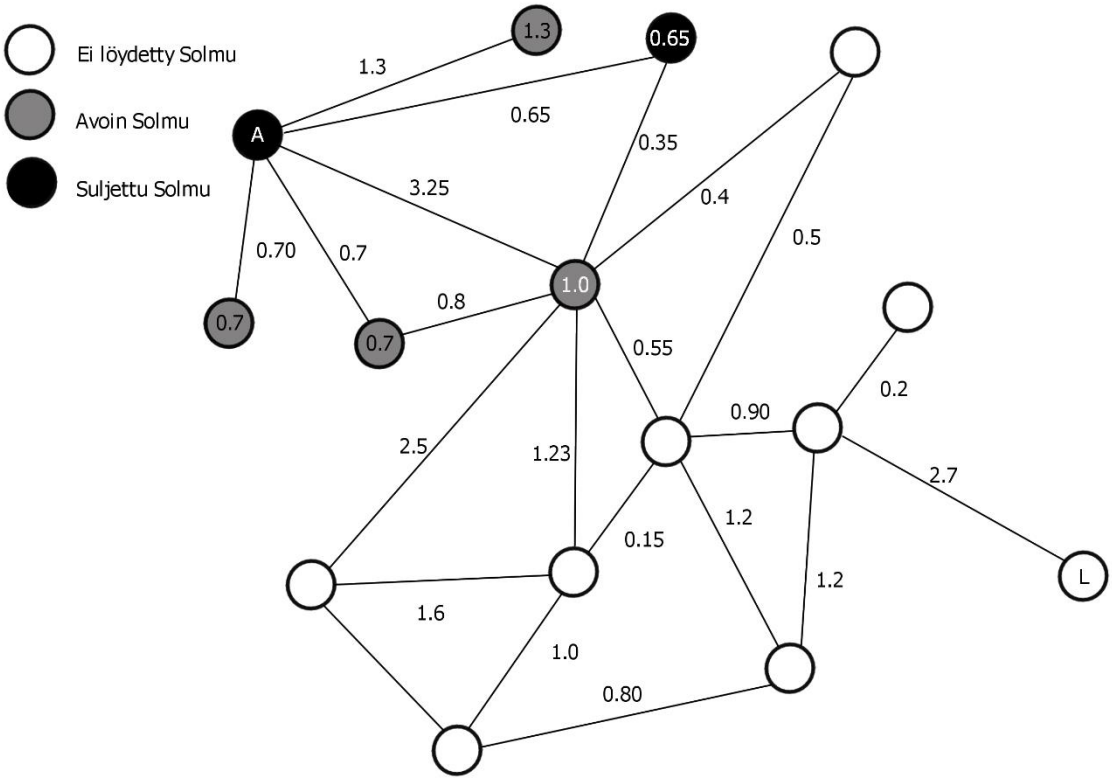
KUVIO 2. Dijkstran aloitusasetelma.

Kuviossa 3 esitetään graafin tilanne alkusolmun prosessoinnin lopuksi, jossa kyseiseen solmuun kiinnittyvien solmujen lasketut arvot on laskettu sekä ne on merkitty avoimiksi.



KUVIO 3. Dijkstra on prosessoinut alkusolmun.

Kuviossa 4 esitetään seuraava vaihe, jossa avoin solmu, jolla on pienin laskettu arvo, prosessoidaan. On syytä huomioida, että yhden avoimen solmun arvo on myös päivitetty, sillä löydettiin halvempi reitti solmuun.



KUVIO 4. Dijkstra on prosessoinut solmun ja päivittänyt sen naapurisolmun arvon.

3 SUUNNITTELU

Tässä työssä tullaan testaamaan kahta eri algoritmia, A* ja Flowfield. Nämä kaksi algoritmia ovat kehitetty ratkaisemaan eri ongelmia. A* laskee yhden reitin kahden pisteen välillä, joka tekee siitä epätehokkaan, jos agenttien määrä on suuri, koska jokaiselle agentille pitää laskea oma reittinsä. Flowfield algoritmina ei skaalaudu agenttien määrän mukaan. Flowfield-konseptia on käytetty ensin virtausdynamiikan (fluid dynamics) simulaatioon (Erkenbach 2013).

Kenttä jaetaan neliöihin ja jokainen neliö vastaa yhtä solmua graafirakenteessa. Työtä varten toteutetaan kolme eri testiskenaariota: sokkelo, 26x26 solmun avoin kenttä sekä 50x50 solmun avoin kenttä. Jokainen skenaario toteutetaan kahdella eri asetelmalla. Yhdessä asetelmassa on yksi reittiä löytävä agentti, toisessa asetelmassa lisätään agenttien määrä kahteentoista. Nämä testit valittiin hahmottamaan molempien algoritmien heikkouksia ja vahvuuksia. A* skaalautuu sekä matkan pituuden että reittiä löytävien agenttien määrän mukaan, Flowfield taas skaalautuu reitinlöytögraafin koon mukaan. Graafin koko myös voidaan olettaa korreloituvaksi pelikentän koon mukaan.

3.1 Flowfield-algoritmi

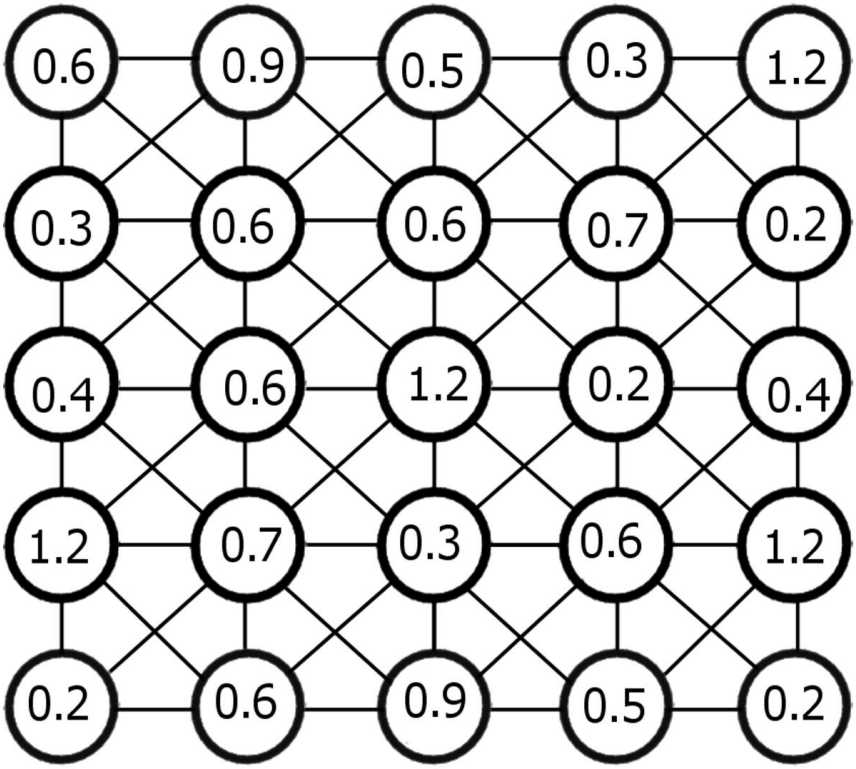
Flowfield-algoritmissa oletetaan, että kenttä on jaettu neliöihin ja jokaisella solmulla on kolme muuttujaa; hinta, lopullinen arvo, sekä suunta, johon se osoittaa. Jokainen solmu on kaarilla kytketty pää- ja väli-ilmansuunnissa sijaitseviin lähimmäisiin solmuihin. Kaarilla ei ole enää itse hintaa, sillä hinta on säilytetty solmuissa.

Flowfield-algoritmin laskennalliseen osuuteen kuuluu kolme osaa:

1. Hintakentän laskeminen
2. Integraatiokentän laskeminen
3. Flowfieldin generoiminen

Hintakentän laskussa asetetaan yksittäisen solmun hinta, solmun hinta perustuu maastoon, jossa solmu on. Jos maasto on vaikeampi kulkuista, niin solmun hinnaksi asetetaan isompi luku. Solmun hinta voi olla mitä vain 1–254 välillä, 255 hinta arvo annetaan sellaisille solmuille, joiden läpi ei päästä kulkemaan. Jos ympäristö muuttuu pelin aikana, hintakenttä pitää laskea uudelleen. Jos pelikentän ympäristö ei muutu hintakenttä täytyy laskea vain kerran. (Erkenbach 2013.)

Kuvio 5 esittää hintakenttää, joka on mielivaltaisesti generoitu.

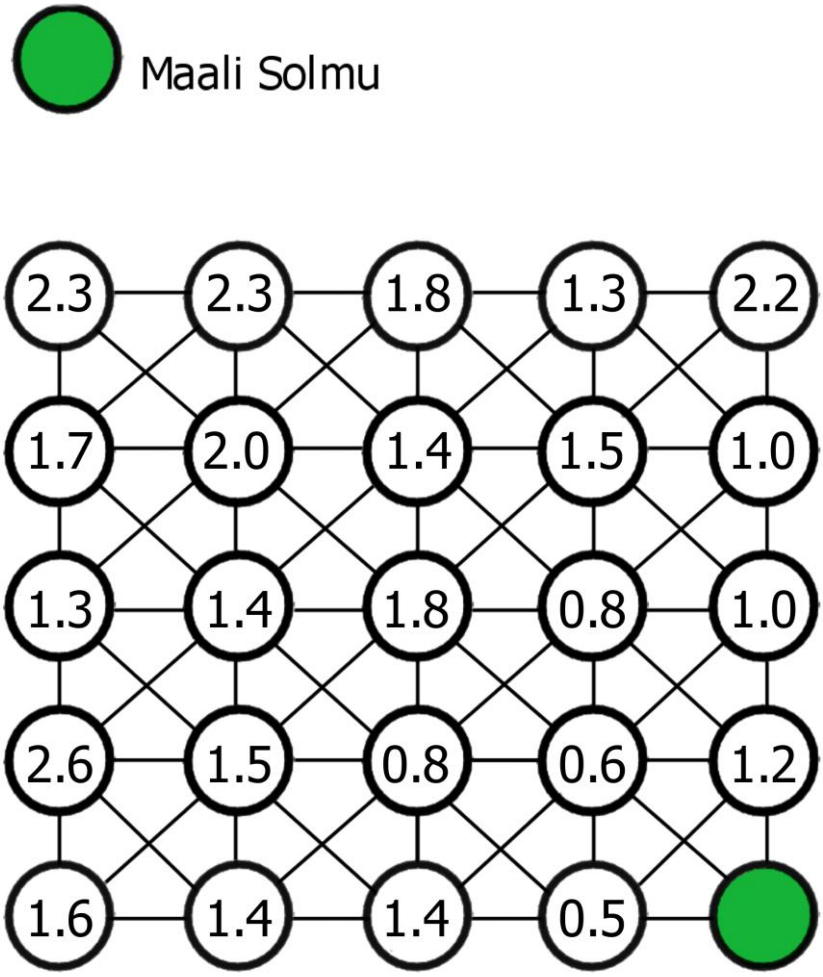


KUVIO 5. Esimerkki hintakentästä.

Seuraava vaihe, integraatiokentän laskeminen, voidaan aloittaa vasta, kun maalisolmu on valittu. Maalisolmu merkataan ja otetaan muistiin. Integraatiokenttä generoidaan käyttämällä modifioitua Dijkstran algoritmia. Algoritmi aloitetaan asettamalla jokaisen solmun lopulliseksi hinnaksi hyvin suuri arvo, paitsi maalisolmun lopullinen arvo on aina nolla. Tämän jälkeen maalisolmu asetetaan avoimien solmujen listalle. Avoimia solmuja aletaan sitten prosessoimaan samalla periaatteella kuin Dijkstrassa, eli viereiset solmut käydään läpi yksitellen. Jokaiselle tämänhetkisen solmun naapurille lasketaan tämänhetkinen hinta (t-hinta),

joka on tämänhetkisen solmun lopullinen hinta plus naapurisolmun hinta. Jos laskettu t-hinta on enemmän kuin solmulla jo oleva lopullinen hinta, niin vanha lopullinen hinta jää voimaan. Jos laskettu t-hinta on vähemmän kuin jo solmulla oleva lopullinen hinta niin uusi t-hinta laitetaan lopulliseksi hinnaksi. Jos solmun hinta on 255 niin se tarkoittaa, ettei sen läpi pääse kulkemaan, eikä sitä lisätä avoimien solmujen listalle, eikä sille lasketa uutta lopullista arvoa. Algoritmi jatkaa toimintaansa, kunnes avoimien solmujen lista on tyhjä. Solmut, joille ei päästä, ei ole laskettu uutta lopullista arvoa, joten niiden lopullinen arvo on yhtä suuri kuin sille alussa asetettu hyvin suuri arvo. (Erkenbach 2013.)

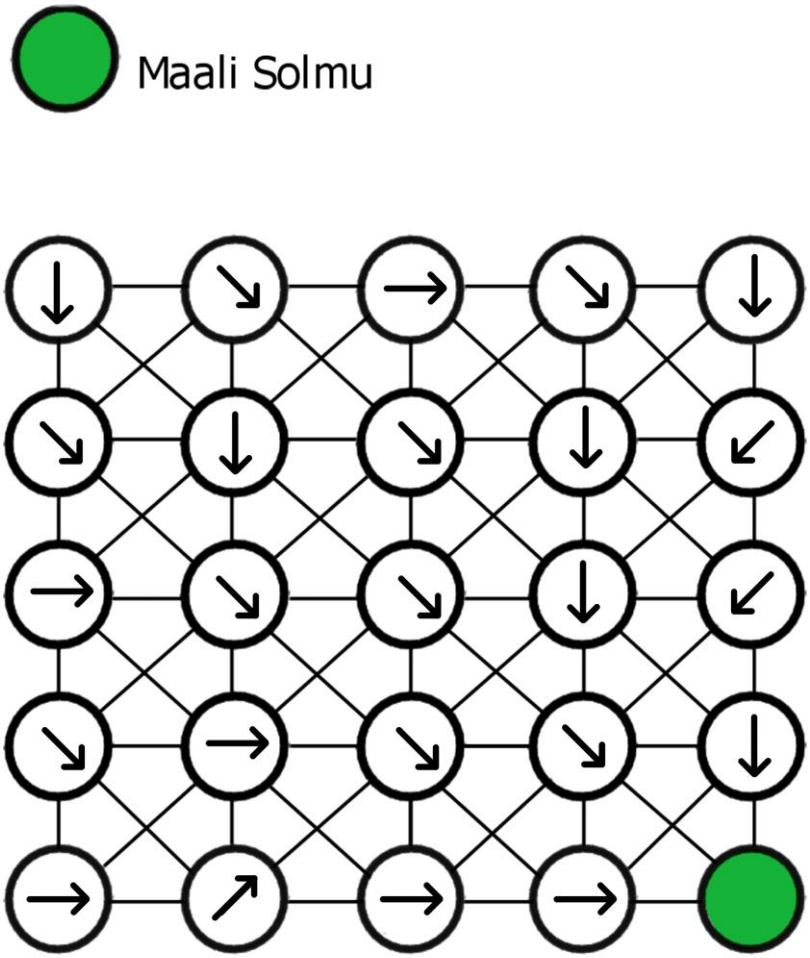
Kuviossa 6 on valittu maalisolmu, sekä on laskettu integraatiokenttä kuviossa 5 esitetyn hintakentän mukaan.



KUVIO 6. Esimerkki integraatiokentästä.

Seuraavaksi integraatiokentän perusteella generoidaan Flowfield, jokainen solmu käydään läpi ja etsitään sen naapuri, jolla on pienin lopullinen arvo. Sitten solmun suunta asetetaan osoittamaan kohti sitä naapuria, jolla tämä pienin arvo on. Jos solmu on sellainen, johon ei päästä, eli sen arvo on yhä sama korkea arvo, johon se asetettiin integraatiokentän laskennan aluksi, niin sellaisten suunnaksi asetetaan ennestään valittu nolla-arvo. Maalisolmun suunnaksi asetetaan myös tämä nolla-arvo. (Erkenbach 2013.)

Kuviossa 7 on tehty Flowfield, kuviossa 6 esitetyn integraatiokentän mukaan.



KUVIO 7. Esimerkki Flowfieldistä.

Solmujen koordinaatit vastaavat tiettyä neliön muotoista maaosuutta pelikentällä. Agentit ovat pelikentällä tietyissä koordinaateissa, kun Flowfield on saatu tuotettua, niin agentit tarkastelevat heitä lähimmän solmun suuntaa ja suunnistavat sen osoittamaan ilmansuuntaan. Kun agenttia lähimpänä oleva solmu vaihtuu, niin

agentti tarkistaa suuntansa uudelleen ja suuntaa uuden lähimmän solmun osoittamaan suuntaan. Näin jokainen agentti voi löytää oman tiensä maaliin saakka. Jos agentti huomaa lähimmän solmunsa suunnan olevan nolla-arvo niin se lopettaa liikkeensä, joten jos agentti on paikassa, josta ei yksinkertaisesti pääse maaliin, tai kun agentti saavuttaa maalisolmun koordinaatit se pysähtyy.

Flowfield-algoritmin hyvä puoli on se, että reitinlöytävien agenttien määrä voi kasvaa mielivaltaisesti, eikä reitinlöytäminen ole yhtään sen raskaampaa. Se myös helpottaa kentän maaston muokkaamista pelin aikana, ei tarvitse välittää kaarista, jos halutaan esimerkiksi asettaa seiniä kenttään, vain solmujen hinnat täytyy muuttaa.

3.2 A*-algoritmi

A*-algoritmi on suunniteltu etsimään reitti kahden solmun välillä graafissa, toisin kuin Dijkstra jonka alkuperäisenä tarkoituksena oli etsiä optimaalinen reitti jokaiseen solmuun aloitus solmusta. A*-algoritmi pohjautuu Dijkstraan, mutta se käyttää heuristiikkaa ohjaamaan algoritmin kulkua, A* ei ota suoranaisesti kantaa minkälaista heuristiikkaa käytetään ja siihen on olemassa erilaisia algoritmeja, tässä työssä tullaan laskemaan lyhin suora reitti hyppien kentän neliöitä pitkin ilman, että otetaan huomioon mahdollisia esteitä.

A* toimii samalla periaatteella kuin Dijsktran-algoritmi, tärkein ero on, että sen sijaan että valitaan avoimien solmujen listalta pienin ”Cost-so-far”-listalta valitaan solmu, joka todennäköisimmin johtaa lyhimpään reittiin kohdesolmuun. A*:n teho riippuu paljolti heuristiikan toimivuudesta, miten todennäköistä on, että heuristinen algoritmi joka valittiin valitsee oikein solmun joka on lyhimmällä reitillä kohteeseen (Funge & Millington 2009)? Pitää myös ottaa huomioon, että heuristinen algoritmi ei saa itsessään olla liian raskas, sen tulisi olla hyvin nopea ja mahdollisimman tarkka ennustuksissaan. Tämän takia A* on paljon joustavampi kuin Dijkstra. Voimme valita heuristisen algoritmin, joka on raskaampi ajaa, mutta tuottaa tarkkoja tuloksia, tai sitten voimme käyttää heuristista algoritmia, joka tuottaa ei-ideaalisia tuloksia, mutta on paljon nopeampi.

Heuristiikalla tarkoitetaan menetelmiä, joilla tehdään arvioita, jotka ovat ”tarpeeksi lähellä” haluttua ratkaisua, kun optimaalisen ratkaisun löytö olisi liian työlästä. Mitä tarkoitetaan määritelmällä ”tarpeeksi lähellä”, riippuu siitä mitä koodaaja/suunnittelija haluaa tehdä. Reitinlöydön yhteydessä sen voidaan ajatella tarkoittavan reitin löytöä, joka ei ole erittäin kallis verrattuna optimaaliseen reittiin, ja että reitti löydetään nopeasti. Maalaisjärki ja asiantuntijan arvio (educated guess) ovat esimerkkejä heuristisista menetelmistä.

A* prosessoi solmuja samalla tavalla kuin Dijkstra, laskien ”Cost-so-far”-muuttujan solmulle. A* poikkeaa Dijkstrasta muutamalla tavalla:

1. Kun ”Cost-so-far” on laskettu, niin lasketaan heuristista algoritmia käyttäen arvio solmun lopullisesta hinnasta ennen kuin solmu lisätään avoimien solmujen listalle.

2. Avoimien listalta valitaan ensin solmu, jolla on pienin arvioitu lopullinen hinta.

3. Kun löydetään tämänhetkisen solmun naapurista suljettu solmu, lasketaan sille uusi ”Cost-so-far” arvo, jos uusi arvo on alempi kuin muistissa oleva arvo niin uudella arvolla korvataan vanha ja solmu asetetaan takaisin avoimien listalle.

Jos heuristisen algoritmin arviot olivat hyvin optimistisia, voi olla tarpeellista laskea kokonaisen joukon solmuja ”Cost-so-far”-muuttujat ja lopullisen hinnan arviot uudestaan. Tämä hoituu jo pelkästään sillä, että lisätään solmu, jolle on laskettu uusi ”Cost-so-far”-arvo takaisin avoimien solmujen listalle. (Funge & Millington 2009, 218.) Kohdesolmun löydyttyä löydetään reitti kohteeseen samalla tavalla kuin Dijkstrassa.

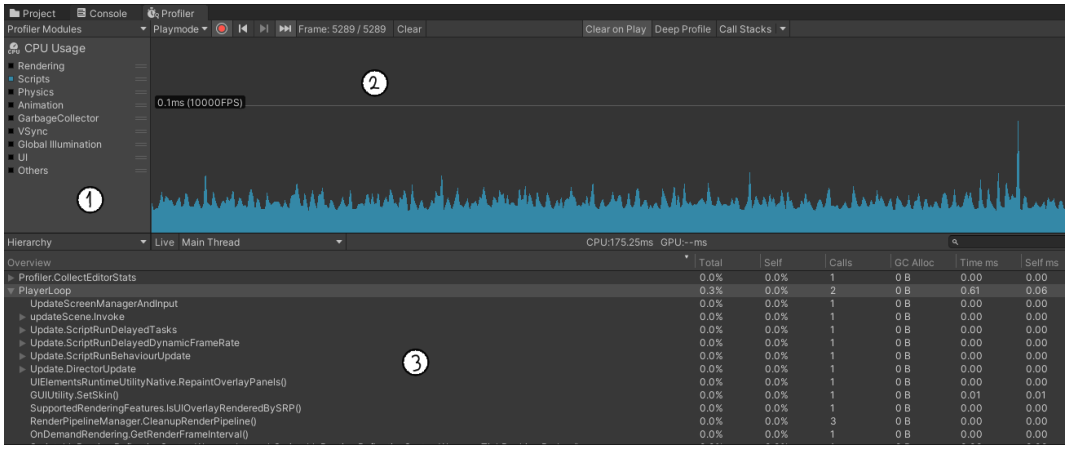
3.3 Tehokkuuden mittaaminen Unity-pelimoottorilla

Unity Profiler on työkalu Unity-pelimoottorin sisällä, jonka avulla voidaan monitoroida muun muassa CPU:n käyttöä, voidaan jopa nähdä paljonko tietyt skriptit

vievät CPU-resursseja tiettynä ajan kohtana. (Unity Software Inc. n.d) (Frame-by-frame)

Profilerilla on myös ”deepstack”-toiminto, jonka avulla voidaan nähdä tiettyjen funktioiden tasolla resurssien kulutus. Se kuitenkin hidastaa testausta ja vie enemmän resursseja. (Unity Software Inc. n.d.)

Koska reitinetsintäkoodi on rajattu yhteen komponenttiin ja komponentilla ei ole muuta tehtävää kuin reitin etsintä, niin testeissä ei tulla käyttämään ”deepstack”-toimintoa. Kuvassa 1 nähdään Unity Profilerin graafinen käyttöliittymä.



KUVA 1. Unity Profilerin graafinen käyttöliittymä.

Kuvan 1 numeroidut kohdat kuvastavat käyttöliittymän osia:

1. Valintapalkki, jolla valitaan mitä osia ajettavasta ohjelmasta monitoroidaan
2. Graafinen esitys jokaisesta raamista (frame) ja siitä miten paljon suoritin aikaa se on vienyt
3. Lista jokaisesta ajettavasta kooditiedostosta, funktioista ja niiden viemästä suoritinajasta.

4 TOTEUTUS

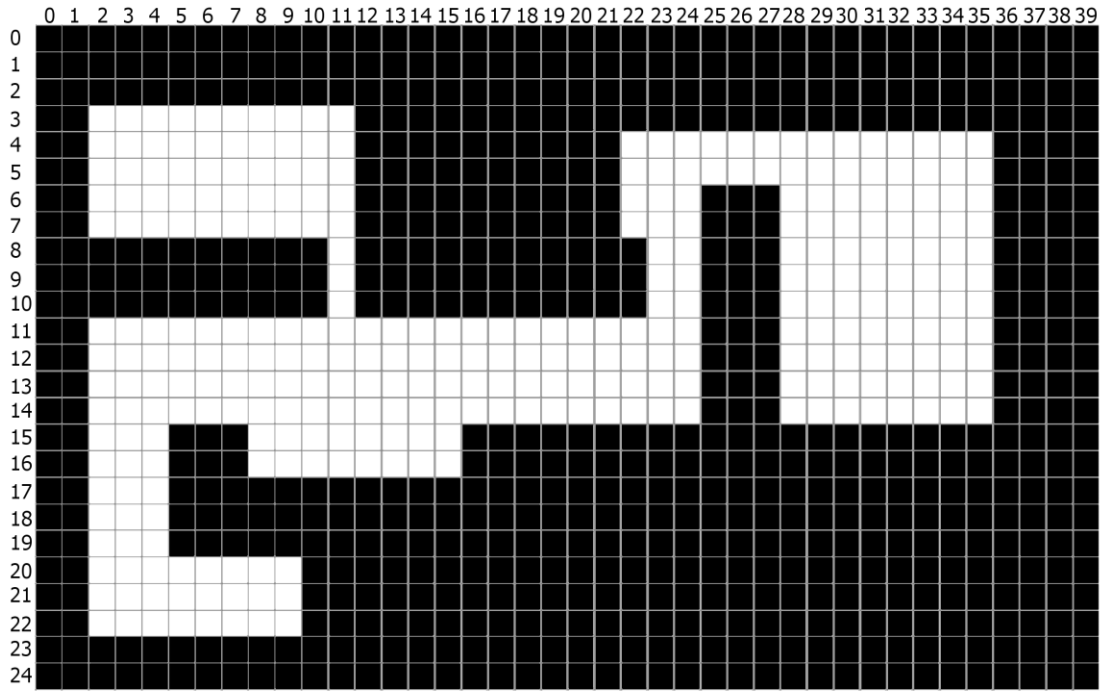
Ensimmäinen ongelma oli algoritmien koodaaminen Unity-pelimoottorin sisään. A*- ja Flowfield-algoritmeille löytyi kuitenkin Unity-pelimoottorille tutoriaalit, joita seuraamalla saatiin algoritmit koodattua Unity-pelimoottorin sisään. Tutoriaalit sisälsivät myöskin pelikentän koodauksen, joten työn yksinkertaistamiseksi seurattiin tutoriaaleja tämänkin osalta (Code Monkey 2019, Thompson 2020). Algoritmit toteutettiin oliopohjaisesti käyttämättä mitään Unity-pelimoottorin sisäisiä optimointimenetelmiä, jotta tulokset olisivat mahdollisimman geneerisiä.

Tutoriaaleista seurattua koodia modifioitiin niin, että testit olisivat yhtenäisiä. Kohdesolmu määriteltiin koodin kautta, jotta jokaista testiä ajaessa agentit olisivat menossa samaan reitinlöytökohteeseen joka kerta. Lisäksi luotiin eri testiasetelmat erillisinä Unity-kenttinä, jotta testit olisi tarpeen mukaan helppo uudelleen ajaa.

Haaste, joka tuli toteutuksen aikana vastaan, oli miten testit tarkalleen toteutetaan: käsin vai automatisoidusti. Unity Profiler pitää muistissaan vain tietyn määrän raameja (frames), ja Profilerin tulokset pitää tuoda Unitystä ulos koodin kautta, se ei tulosta tuloksiaan automaattisesti mihinkään. Jotta testit olisi voinut automatisoida, olisi täytynyt implementoida Profilerin tuloksien ulos saanti sekä jonkinlainen skripti, joka olisi etsinyt tarkalleen sen raamin, jolla reitinlöytö olisi tehty. Lisäksi skriptin olisi tullut noukkia vain kyseinen raami ja ottaa se johonkin muistiin. Todettiin että automatisaatio olisi vienyt tarpeettoman paljon aikaa. Testit toteutettiin käsin, jolloin testi laitettiin pyörimään ja kun reitinlöytö oli toteutettu, testi loppui ja Profilerista katsottiin se raami, jonka aikana reitinlöytö tehtiin. Lopuksi otettiin Excel-taulukkoon ylös, mikä tulos tuli.

Työtä varten toteutettiin kolme eri testiasetelmaa: kaksi avointa kenttää, joissa ei ole esteitä, kooltaan 26x26 solmua, sekä 50x50 solmua, sekä 40x25 solmun kokoinen sokkelo, joka toteutettiin simuloimaan mahdollisia esteitä, joita tulee vastaan pelikentällä. Korkeuseroja tai vaikeasti kuljettavia maastoja ei simuloitu testien aikana.

Sokkelotestin muoto valittiin mielivaltaisesti, mutta ottaen huomioon, että sokkelo vaikeuttaisi reitinlöytöä esimerkiksi niin, että agenttien tulee kulkea käytävää pitkin kääntyen kulmien ohi. Avoimet kentät valittiin demonstroimaan erityisesti Flowfieldin skaalautumista navigoitavan kentän koon mukaan. Lisäksi jokainen testikenttä ajettiin kahdella eri konfiguraatiolla, ensimmäisessä oli yksi agentti löytämässä reittiä ja toisessa oli 12 agenttia demonstroimaan A*:n riippuvuutta agenttimäärästä ja vastaavasti Flowfieldin riippumattomuutta siitä. Sokkelokartan lopullinen muoto esitetään kuviossa 8.



KUVIO. 8 Sokkelokartta.

Ennen testien ajoa tietokone, jolla testit tultaisiin ajamaan, kytkettiin irti internetistä ja käynnistettiin uudelleen. Näin minimoitiin muiden ajossa olevien ohjelmien kuorma testejä ajaessa. Testit toteutettiin tietokoneella, jonka spesifikaatiot ovat seuraavanlaiset. GPU = GTX 1080 CPU = Intel i7-4790k, 4-core @ 4.00GHZ.

Jokainen testi asetelma/konfiguraatio yhdistelmä ajettiin 25 kertaa ja näiden 25 tulosten keskiarvot laskettiin. Kaikki tulokset ovat työn liitteessä 1.

5 TULOKSET JA POHDINTA

5.1 Tulokset

Lukuarvot ovat suoritinaika-arvoja (CPU-time) kuvastaen sitä, kuinka kauan suorittimella kesti suorittaa reitinlöytöalgoritmille tarvittavat komennot. Testien tulokset on kirjattu taulukkoon 1.

TAULUKKO 1. Ajettujen testien keskiarvotetut tulokset

Algoritmi	Agenttien määrä	Sokkelo 40x 25 (ms)	Avoin 26x26 (ms)	Avoin 50x50 (ms)
A*	1	4,68	3,74	5,09
A*	20	17,14	9,85	22,98
Flowfield	1	68,96	11,25	21,98
Flowfield	20	69,92	10,73	22,37

Ajettujen testien tulokset menivät kuten teoriassa pitäisikin. Algoritmit oli tarkoituksellisesti tehty käyttämättä Unityn omia optimointimenetelmiä, joten tuloksia ei voida käyttää punnitsemaan Unity-pelimoottorin tehokkuutta. A* skaalautuu agenttien määrän sekä matkan pituuden mukaan, kun taas Flowfield välittää kentän koosta sekä kentän monimutkaisuudesta.

5.2 Pohdinnat

Flowfield on optimoitu suurien agenttimäärien liikuttamiseen kartalla, mutta sen tehokkuus kärsii, kun pelikentät suurenevat. Jos peli sisältää suuren määrän agentteja, on todennäköisempää, että pelikenttä on suuri, jotta kaikki agentit mahtuisivat liikkumaan siellä. Tämän takia olisi tärkeää ajatella Flowfieldin implementaatiota tarkasti. Jos Flowfieldin tehokkuus tulee esille vasta, kun agentteja on esimerkiksi satoja voi olla parempi valita jokin toinen algoritmi, jos pelin aikana ei tulla säännöllisesti ohjaamaan satoja yksikköjä.

A* laskee erillisen reitin jokaiselle agentille, jotka ovat navigoimassa pelimaailmassa ja sen tehokkuus riippuu myös reitin pituudesta. Vaikka pelikenttä olisikin iso, niin jos yksittäiset matkat, joita agenteille täytyy laskea ovat lyhyitä, esimerkiksi sodan sumun (fog of war) takia, niin A* voi olla yksinäänkin tehokas työkalu pelin reitinlöydön implementoimiseen.

Yksi mahdollinen tapa, jolla voitaisiin parantaa molempien algoritmien tehokkuutta, on hierarkkinen reitinlöytö, jossa sen sijaan että kartta jaotellaan yhteen suureen graafiin, sen eri osat yhdistetään korkeatasoisemmalla graafilla. Esimerkiksi pelikentässä, jossa on yksi kolmekerroksinen rakennus, luodaan korkeatasoisempi graafi, joka esittää jokaista huonetta rakennuksessa. Yhdessä huoneessa voi olla kymmeniä tai jopa satoja solmuja, mutta korkeampitasoisella graafilla ne esitetään yhdellä solmulla. Korkeampitasoisen graafin solmujen yhteydet voidaan sitten määrittää alempitasoisten solmujen yhteyksillä. Esimerkiksi jos makuuhuoneesta pääsee keittiöön ja keittiöstä olohuoneeseen, mutta makuuhuoneesta ei pääse olohuoneeseen suoraan, niin huonetason graafilla makuuhuoneen ja keittiön välillä on kaari, ja keittiöstä kaari olohuoneeseen, mutta olohuoneesta ei ole kaarta makuuhuoneeseen. Tätä abstraktointia voitaisiin jatkaa korkeammalle, esimerkiksi jokaisen kerroksen huoneet niputetaan yhden kerrostason graafin solmun alle (Funge & Millington 2009, 255).

Kuvatun kaltainen hierarkia auttaa tietämään, mitä kartan osia tulee käyttää reitinlöydössä, ja voidaan selvittää reitti pienemmissä osissa. Jos tahdotaan löytää reitti pohjakerroksen aulasta toisen kerroksen varastohuoneeseen, voidaan ensin kerrostasolla katsoa, että reitti kulkee pohjakerroksesta suoraan ensimmäiseen kerrokseen ilman välietappeja. Huonetasolla voitaisiin nähdä, että täytyy kulkea aulasta portaikkoon, portaikosta käytävälle ja käytävältä varastohuoneeseen. Tämän jälkeen voitaisiin ensimmäiseksi laskea reitti aulasta portaikkoon. Kun portaikko on saavutettu, lasketaan reitti käytävälle ja niin edelleen. (Funge & Millington 2009, 256). Ensimmäinen reitti olisi vain aulasta portaikkoon ja vasta sitten kun agentit ovat saavuttaneet portaikon seuraava reitti laskettaisiin. Tämä auttaisi pitämään laskettavat reitit mahdollisimman lyhyinä, jolloin laskenta aikaa ei kuluisi hukkaan, jos agentin reitti muuttuisikin kesken kaiken, toisin kuin jos laskettaisiin heti koko reitin aulasta varastoon. Tämä toimisi Flowfield-algoritmin

kanssa myös, täytyisi vain laskea niiden alueiden Flowfieldit, mistä löytyy yksiköitä, eikä koko karttaa tarvitsisi laskea joka kerta.

Tämäkään ei vielä kata kaikkia ratkaistavia ongelmia reitinlöydön kanssa. Jos halutaan esimerkiksi koodata pelikentällä liikkuville agenteille fysiikkatörmäykset, tulisi reitinlöydössä ottaa huomioon agenttien mahdollinen törmäily toisiinsa, ja pyrkiä minimoimaan se. Tähän ongelmaan auttaa ohjauskäyttäytyminen (steering behaviour) jossa ideana on käyttää yksinkertaisia parametrejä, jotta saataisiin agentit liikkumaan yhdessä kitkattomasti. (Fray & Pentheny 2013.)

LÄHTEET

Chen, C., Li, Y. & Li, T. 2011 KM-A* Pathfinding Algorithm Based on Hierarchical Clustering and Strengthened DB Index Criteria. Teoksessa 2011 International Conference on Machine Learning and Cybernetics. Manhattan:IEEE 1571-1576

Clement, J. Video Gaming market in the United States. Nettisivu. Statista. Julkaistu 19.11.2021 Viitattu 9.03.2022 https://www.statista.com/topics/868/video-games/#dossier-chapter1

Code Monkey. 2019. A* Pathfinding in Unity. Youtube-video Julkaistu 20.10.2019. Viitattu 30.02.2021 https://youtu.be/alU04hvz6L4

Erkenbranch, L. Flow Field Pathfinding. Verkkosivun Blogi kirjoitus. Julkaistu 5.12.2013 Viitattu 07.05.2020 https://leifnode.com/2013/12/flow-field-pathfinding/

Fray,A., Pentheny, 2013. G. The Next Vector:Improvements in AI Steering Behaviours. Konferenssiesitys 25-29.3.2013 videokuvattu. Informa. San Francisco

Funge, J. & Millington, I. 2009. Artificial Intelligence for Games Second Edition. Amsterdam: Elsevier.

Haas, J. 2014 A History of the Unity Game Engine. Tietotekniikka. Worcester Polytechnic Institute. Interactive Qualifying Project. Viitattu 20/03/2022 https://digital.wpi.edu/concern/student_works/tx31qh96p?locale=en

Schrijver, A. 2012. On The History of the Shortest Path Problem. Teoksessa Documenta Mathematica. Extra Volume: Optimization Stories. Washington DC: SPARC, 155-167.

Thompson, J. 2020. Tutorial - Flow Field Pathfinding in Unity. Youtube-video Julkaistu 14.08.2020. Viitattu 01/03/2021 https://youtu.be/tSe6ZqDKB0Y

Unity Software Inc. Unity Documentation Profiler Overview. Netti Dokumentaatio. Viitattu 30.02.2021 https://docs.unity3d.com/Manual/Profiler.html viitattu 24/02/2021

LIITTEET

Liite 1. Testitulokset

Avoin kenttä, 50x50 solmua, A*-algoritmi

Algoritmin suoritusaika, 1 agentti (ms)	Algoritmin suoritusaika, 12 agenttia (ms)
4,85	21,84
4,59	24,93
4,85	26,76
4,86	24,25
4,94	23,55
4,84	24,32
7,05	23,89
4,89	24,04
5,18	23,66
4,89	21,63
4,90	24,17
5,31	24,42
5,27	23,85
5,02	27,03
5,10	21,97
4,90	24,30
5,26	21,78
4,92	25,11
4,90	21,40
5,02	24,60
5,23	27,91
5,47	25,02
5,03	22,20
5,04	21,79
5,02	24,23

Avoin kenttä, 26x26 solmua, A*-algoritmi

Algoritmin suoritusaika, 1 agentti (ms)	Algoritmin suoritusaika, 12 agenttia (ms)
3,70	8,57
3,60	10,65
3,56	9,70
4,22	11,14
3,68	9,78
3,92	9,65
3,64	9,58
3,73	8,68
3,56	8,94
3,75	8,91
3,81	8,69
3,66	9,30
3,73	11,20
3,88	9,59
3,85	11,38
3,64	9,52
3,75	10,74
3,69	8,67
3,66	9,06
3,63	9,23
3,75	11,80
3,57	9,13
3,79	9,09
3,56	11,54
4,07	11,69

Sokkelo kenttä, A*-algoritmi

Algoritmin suoritusaika, 1 agentti (ms)	Algoritmin suoritusaika, 12 agenttia (ms)
4,34	15,88
4,45	15,83
4,59	16,07
4,33	16,15
4,28	15,65
4,45	15,52
4,81	15,95
4,20	16,45
4,49	18,26
4,30	18,46
4,33	18,96
4,85	18,85
4,47	16,16
4,26	16,55
4,66	16,37
4,38	16,35
4,46	18,99
4,29	19,25
6,70	16,39
4,88	16,36
5,09	19,67
4,36	18,66
4,35	16,24
6,82	19,08
4,78	16,45

Avoin kenttä, 50x50 solmua, Flowfield-algoritmi

Algoritmin suoritusaika, 1 agentti (ms)	Algoritmin suoritusaika, 12 agenttia (ms)
20,23	23,89
20,81	20,78
21,25	20,53
19,55	22,03
24,07	22,42
23,34	23,68
19,10	21,51
20,29	21,53
23,57	21,56
23,73	21,54
22,59	22,01
20,96	21,17
23,93	23,28
21,68	21,85
22,89	19,68
24,59	21,62
22,44	23,15
21,44	23,09
21,64	23,48
20,21	23,53
23,81	24,19
22,64	23,69
19,57	22,33
23,88	22,11
19,99	24,56

5(6)

Avoin kenttä, 26x26 solmua, Flowfield-algoritmi

Algoritmin suoritusaika, 1 agentti (ms)	Algoritmin suoritusaika, 12 agenttia (ms)
9,39	8,60
9,51	11,19
12,17	11,23
12,03	8,34
9,40	11,25
12,07	11,80
12,10	10,96
9,48	14,22
12,31	11,72
9,79	11,75
12,04	11,21
11,93	11,01
12,31	9,18
11,98	11,56
9,89	11,52
12,74	9,23
12,36	11,66
9,85	9,07
12,44	9,39
13,14	8,84
11,92	11,97
9,89	12,10
9,91	8,73
12,88	12,03
9,79	9,67

Algoritmin suoritusaika, 1 agentti (ms)	Algoritmin suoritusaika, 12 agenttia (ms)
67,67	67,53
67,25	69,06
69,79	69,97
69,65	69,49
67,90	70,21
73,08	67,95
70,61	68,42
66,60	70,08
71,65	68,26
65,90	70,64
67,82	70,06
71,76	70,84
72,19	70,47
66,59	69,36
67,72	70,17
69,14	71,82
69,40	70,15
68,66	70,13
67,79	70,80
66,78	67,86
69,38	68,85
67,24	67,96
69,73	71,77
69,79	73,03
69,87	73,22

6(6)

Sokkelo kenttä, Flowfield-algoritmi