Juha Kellokoski

REITINHAKU REAALIAIKAISISSA SIMULAATIOISSA

Opinnäytetyö
KESKI-POHJANMAAN AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma
Syyskuu 2009

SISÄLTÖ

KESKI-POHJANMAAN AMMATTIKORKEAKOULU
MELLERSTA ÖSTERBOTTENS YRKESHÖGSKOLA

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Yksikkö	**Aika**	**Tekijä/tekijät**
Tekniikka ja liiketalous	Syyskuu 2009	Juha Kellokoski
Koulutusohjelma		
Tietojenkäsittely		
Työn nimi		
Reitinhaku reaaliaikaisissa simulaatioissa		
Työn ohjaaja		**Sivumäärä**
Kauko Kolehmainen		43
Työelämäohjaaja		

Tässä opinnäytetyössä käsitellään erilaisia reitinhakumenetelmiä keskittyen erityisesti A*-algoritmiin. Toimintaperiaatteiden lisäksi esitetään potentiaalisia lajittelumenetelmiä ja tietorakenteita. Vertailussa on heuristisia funktioita, jotka vaikuttavat merkittävästi algoritmin suoritustehoon ja hakutulokseen. Työssä sivutaan myös reaaliajan ja dynaamisen ympäristön tuomia haasteita.

Lopussa esitellään reaaliaikainen sovellus, joka käyttää A*-algoritmia reitinhakuun sekä binäärikekoa tiedon varastoimiseen ja hakemiseen.

Asiasanat
algoritmit, reitinhaku, simulaatiot

CENTRAL OSTROBOTHNIA
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

CENTRAL OSTROBOTHNIA UNIVERSITY OF APPLIED SCIENCES	**Date** September 2009	**Author** Juha Kellokoski
Degree programme Business Information Technology		
Name of thesis Pathfinding in real-time simulations		
Instructor Kauko Kolehmainen		**Pages** 43
Supervisor		

The purpose of this thesis is to represent different methods of pathfinding, specifically A*-algorithm. In addition to the basic functionality, potential sorting methods and data structures are represented. Heuristics, which have significant effects on the performance and search results, are compared. Also, thesis deals with some issues introduced by real-time and dynamic environment.

In the end, the thesis demonstrates a real-time application using A*-algorithm for pathfinding and a binary heap to store and organize data.

Key words

algorithms, pathfinding, simulations

1 JOHDANTO

Reitinhaku on erityisesti peleissä yksi keskeisimpiä algoritmeja. Huolimatta siitä, että huomattavassa osassa pelejä on jonkinlainen reitinhaku, se on edelleen haastava toteuttaa. Haasteellisuus kasvaa sen mukaan, mitä monimutkaisempi ympäristö on käytössä. Yksi syy tähän on se, että ei ole niin sanottua "yleispätevää" hakualgoritmia vaan jokainen on integroitava sovelluskohtaisesti. Sama ongelma on kaikissa tekoälyä käyttävissä sovelluksissa, mikä estää yleispätevän tekoälyn kehittämisen. Tietokonegrafiikan saralla tilanne on toinen. Näytönohjaimet sisältävät 3D-mallien piirtoon käytettäviä ominaisuuksia, mikä on mahdollistanut viime vuosien huiman kehityksen matkalla kohti fotorealistisuutta. Ainakaan toistaiseksi ei ole kehitetty kotikäyttäjien koneita varten ”tekoälyohjainta”, joka parantaisi sovellusten suorituskykyä.

Graafialgoritmit ovat hyvin yleisiä peleissä, ja niistä juuri reitinhaku on ehkä kaikista tärkein. Tässä työssä käydään aluksi läpi yleisimpiä graafihakuja, jonka jälkeen luvussa 3 tutustutaan tarkemmin hyvin suositun A*-hakualgoritmin toimintaan. Esittelyssä on erilaisia lajittelumenetelmiä, joita on mahdollista käyttää tässä algoritmissa, sekä binäärikeko, joka on tietorakenne tiedon tehokasta käsittelyä ajatellen. A* käyttää heuristiikkaa matkan pituuden arviointiin. Seuraavana esitelläänkin erilaisia heuristisia funktioita, jotka vaikuttavat merkittävästi algoritmin tehokkuuteen ja tutkittavien potentiaalisten reittien määrään. Tietorakenteet ja karttatyypit -luvussa pohditaan erilaisten karttatyyppien soveltuvuutta. Käsiteltävinä karttatyyppeinä ovat ruudukko, monikulmiot ja hierarkiat. Luvussa 5 sivutaan lyhyesti geneettisiä algoritmejä. Geneettisiä algoritmeja käytetään muun muassa oppivissa järjestelmissä ja niitä voidaan käyttää myös reitinhaussa. Oikea ratkaisu ”jalostetaan” alkuperäisestä satunnaisgeneroidusta ratkaisujoukosta. Lopuksi luvussa 6 pohditaan reaaliajan tuomia haasteita eli ohjelman ajon aikana jatkuvasti muuttuvia karttoja. Esittelyssä on myös sovellus, joka käyttää A*-algoritmia reitinhaussa.

2 GRAAFIALGORITMIT

2.1 Yleisimmät menetelmät

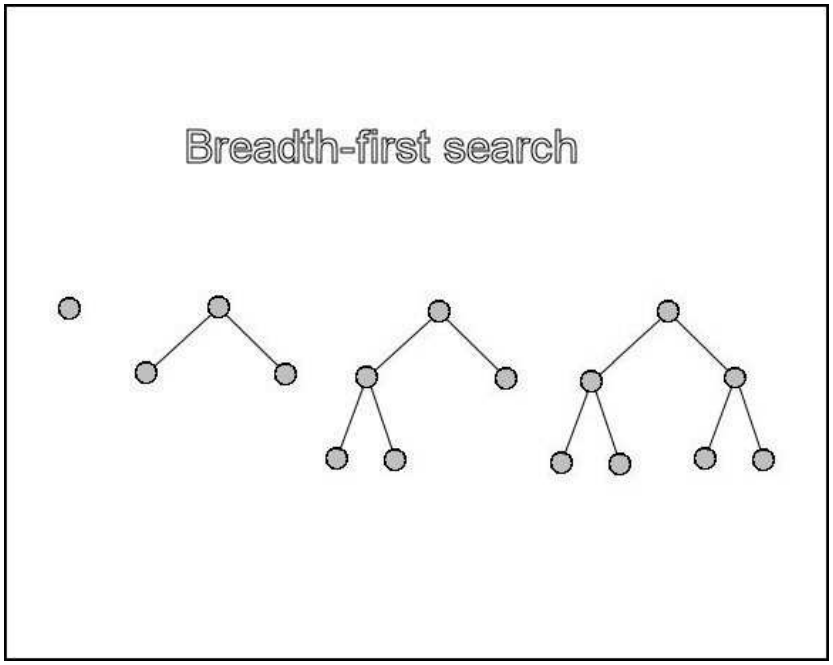
Graafit ovat tärkeitä reaalimaailmassa käytettävissä sovelluksissa. Graafi koostuu joukosta solmuja, jotka liittyvät toisiinsa kaarien avulla. Seuraavana on esiteltynä erilaisia hakumenetelmiä, joita voidaan käyttää tiedon hakemiseen graafista. (Loech 2000.)

2.2 Sokea haku

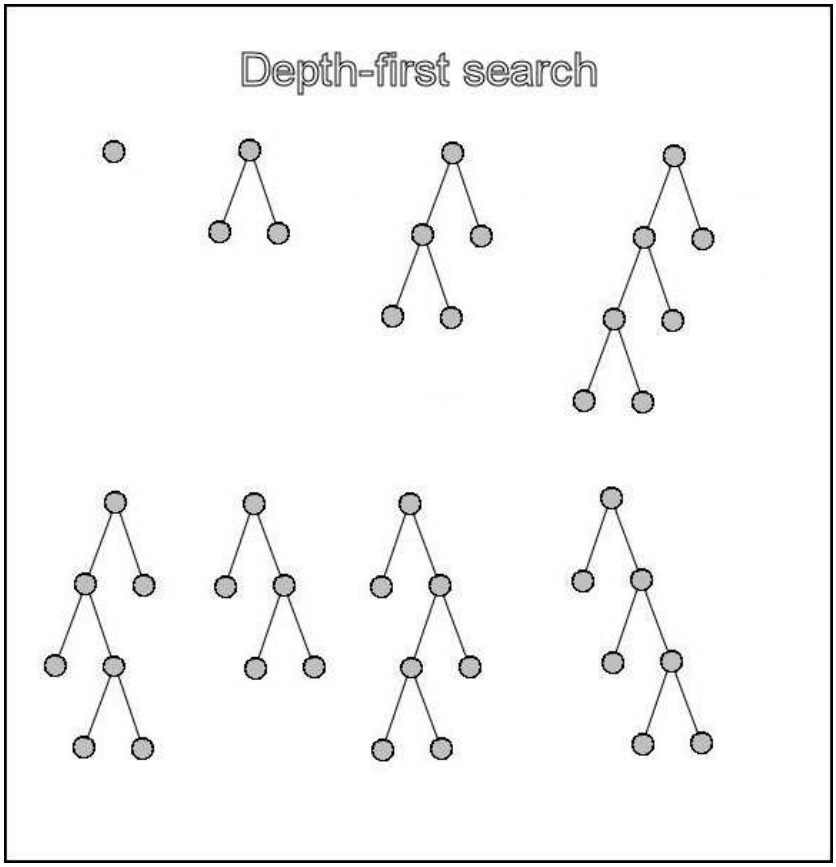
Sokeaa hakua (blind search) käytetään silloin, kun graafin tai kartan rakenteesta ei ole tarkkaa tietoa. Tällöin ei voida arvioida matkaa päätepisteeseen. Ainoa tieto saadaan naapurisolmuista, joiden määrä tiedetään, mutta siitä eteenpäin mikään ei ole varmaa, ennen kuin haku on saavuttanut seuraavan solmun. Tässä tapauksessa voidaan käyttää kahta sokeaksi hauksi luokiteltavaa algoritmia: leveys ensin (breadth first, KUVIO 1), tai syvyys ensin (depth first, KUVIO 2) -hakua. Näiden hakujen voidaan taata löytävän reitin, mikäli sellainen on olemassa. (Loech 2000.)

2.3 Leveys ensin -haku ja syvyys ensin -haku

Leveys ensin -haku tutkii kerralla jokaisen mahdollisen reitin ensimmäisen solmun. Tämän jälkeen se tutkii jokaisen reitin toisen solmun ja jatkaa samalla tavalla, kunnes löytää kohdesolmun tai on tutkinut koko kartan. Syvyys ensin -haku puolestaan valitsee ensimmäisen reitin ja tutkii sen loppuun asti, ennen kuin siirtyy seuraavaan. Molempien voidaan taata löytävän reitin kohteeseen, mutta nämä eivät kuitenkaan ole erityisen tehokkaita algoritmeja. Leveys ensin -haku on hidas, jos mahdolliset reitit haarautuvat usein, eli niillä on paljon naapurisolmuja. Syvyys ensin -haku puolestaan toimii huonosti, jos reitit ovat hyvin pitkiä. (Loech 2000.)



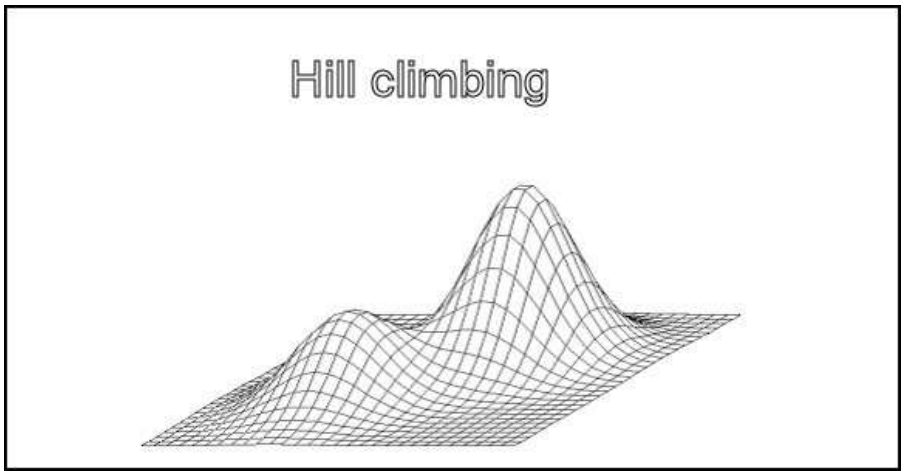
KUVIO 1. Leveys ensin -haku



KUVIO 2. Syvyys ensin -haku

2.4 Jyrkimmän nousun menetelmä

Jyrkimmän nousun menetelmällä (hill climbing, KUVIO 3) voidaan parantaa syvyys ensin -hakua. Tämän menetelmän periaatteena on lähteä liikkeelle satunnaisella vaihtoehdolla, jota aletaan muuttaa pienillä parannuksilla. Haku loppuu, kun parannuksia ei enää voi tehdä. Ongelmana on, että pyrittäessä siirtymään kohti suurempia arvoja saatetaan löytää vain pieni ”kukkula”, sen sijaan että huomattaisiin vieressä oleva ”vuori”. Tällaisen paikallisen maksimin kohdalla pitäisi pystyä hakeutumaan kohti pienempiä arvoja. Ratkaisuna voidaan käyttää esimerkiksi satunnaisuutta tai useamman solmun tutkimista kerrallaan. Yksi ongelma on myös ”tasanne”, jolla kaikki lähellä olevat arvot ovat yhtä suuria, jolloin haku alkaa edetä satunnaisesti. (Marczyk 2004.)



KUVIO 3. Jyrkimmän nousun menetelmä

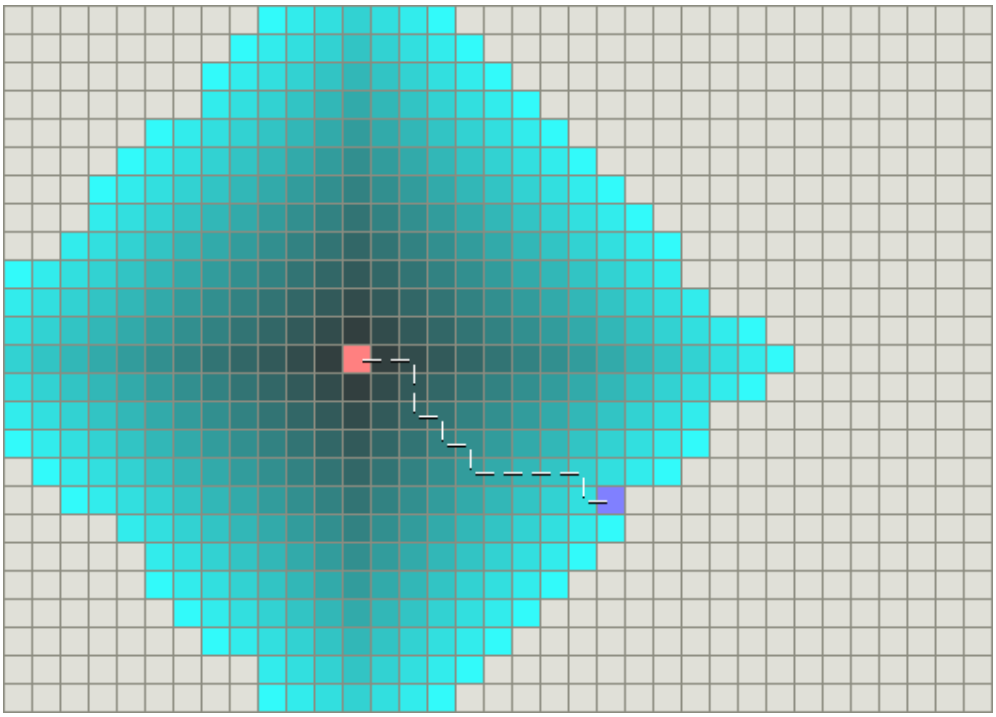
2.5 Sädehaku

Myös leveys ensin -hakua voidaan parantaa. Sädehaku (beam search) rajoittaa tutkittavaa aluetta tietyllä leveydellä. Käytännössä tämä tapahtuu vähentämällä tutkittavien naapurisolmujen määrää. Valinnan perustana on yleensä naapurin etäisyys suorasta linjasta aloitussolmun ja kohdesolmun välillä. Huonona puolena sädehaku saattaa jättää jo alussa pois potentiaalisia reittejä, ehkä jopa ainoat, jot-

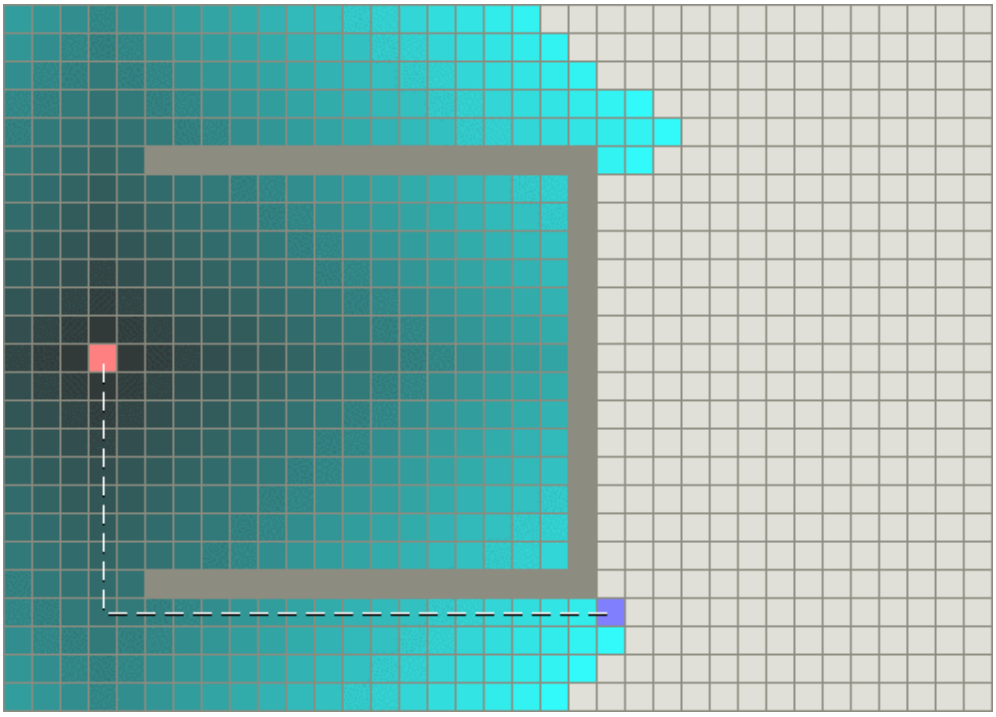
ka johtavat kohdesolmuun. Sekä jyrkimmän nousun menetelmällä että sädehaulla on ongelmana, että ne eivät välttämättä löydä yhtäkään reittiä kohdesolmuun. (Fern & Xu 2007, 1.)

2.6 Dijkstran algoritmi

Yksi tunnetuimmista reitinhaussa käytettävistä graafialgoritmeista on Dijkstran algoritmi. Nimensä mukaisesti sen toi yleisön tietoisuuteen Edsger Dijkstra vuonna 1959. Tämän algoritmin voidaan taata löytävän lyhyimmän reitin, tai mahdollisesti useammankin, koska yhtä lyhyitä reittejä saattaa olla monia. Periaatteena on laajentaa hakualuetta aina lähimpään tutkimattomaan solmuun, jolloin alue kasvaa tasaisesti joka suuntaan kuten kuvioissa 4 ja 5. (Dijkstra 1959.)



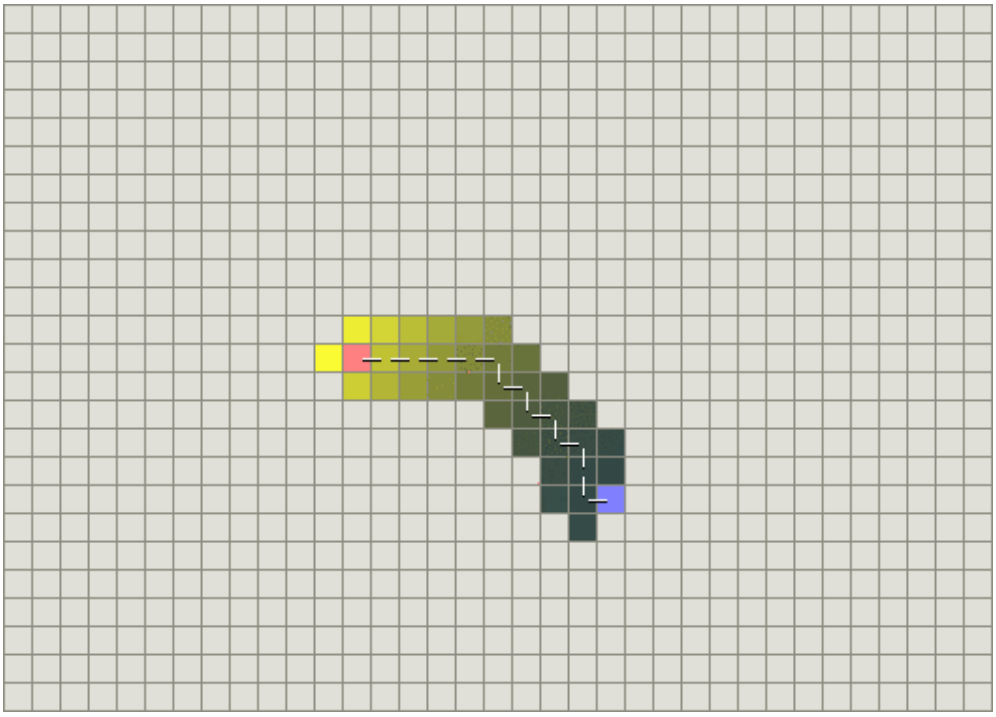
KUVIO 4. Dijkstran algoritmi (Patel 2009.)



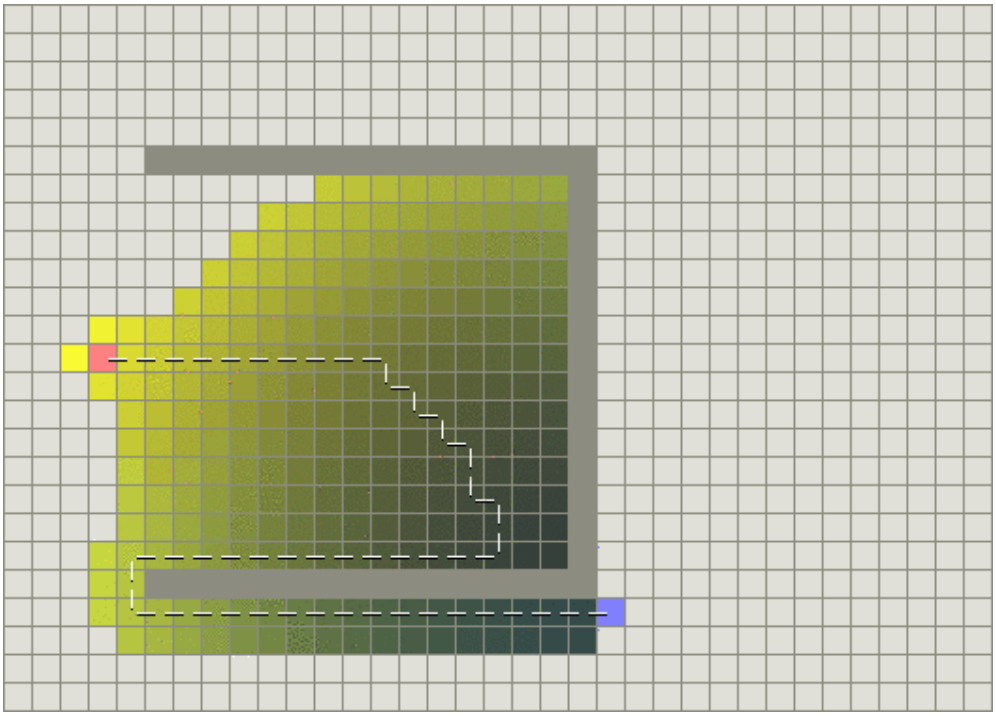
KUVIO 5. Dijkstran algoritmi ja este (Patel 2009.)

2.7 Paras ensin -haku

Paras ensin -haku (best-first-search, KUVIOT 6 ja 7) toimii lähes samalla pariaatteella, mutta seuraavaa tutkittavaa solmua valittaessa se ottaakin huomioon etäisyyden tutkittavasta solmusta päätepisteeseen. Koska todellista kuljettavaa etäisyyttä päätepisteeseen ei tiedetä, joudutaan käyttämään approksimointia eli lopullinen matka arvioidaan. Matka voidaan arvioida monella erilaisella heuristisella funktiolla, joista jokainen vaikuttaa omalla tavallaan lopulliseen reittiin. Tämän vuoksi haku on huomattavasti tehokkaampi kuin Dijkstran algoritmi; sen tutkittava alue laajenee kohti päätepistettä. Huonona puolena sen ei voida taata löytävän lyhyintä reittiä. (Drakos 1993.)



KUVIO 6. Paras ensin -haku (Patel 2009.)



KUVIO 7. Paras ensin -haku ja este (Patel 2009.)

2.8 Optimaalinen haku

Joissain tapauksissa, muttei läheskään kaikissa, riittää, että löydetään edes yksi reitti. Monesti on kuitenkin tärkeää, että löydetään nimenomaan kaikista lyhyin, nopein tai helpoin reitti. Optimaalinen haku löytää ainoastaan parhaimman reitin. Yksi keino on etsiä kaikki mahdolliset reitit ja tallettaa jokaisen pituus. Lopuksi suoritetaan yksinkertainen vertailu ja valitaan se reitti, joka on lyhyin. Tämä vaatii kuitenkin käytännössä hyvin paljon suoritintehoa ja muistia, minkä takia se soveltuu vain pienien ongelmien ratkaisemiseen. Hakualgoritmina voi toimia syvyys ensin -haku tai leveys ensin -haku. Toteutuksen erona on se, että reitin löydyttyä ei pysähdytä vaan jatketaan, kunnes kaikki mahdollisuudet on kokeiltu. Menetelmää voidaan parantaa käyttämällä heuristiikkaa, jotta ensin löydetään reittejä, jotka ovat lähempänä kohdesolmua. Jos huomataan, että tutkittava reitti on kasvanut pidemmäksi kuin jokin aikaisemmin löydetty, reitti hylätään, vaikka tutkiminen olisi kesken, ja siirrytään tutkimaan seuraavaa vaihtoehtoa. (Drakos 1993.)

2.9 A*-haku

A* on yhdistelmä Dijkstran algoritmista ja paras ensin -hausta. Se laskee niin sanotun F(n)-arvon, jossa on laskettuna yhteen etäisyys aloituspisteeseen ja arvioitu etäisyys päätepisteeseen. Arvioimalla etäisyyden se osaa laajentaa hakua päätepisteen suuntaan vähentäen tutkittavien solmujen määrää, mikä tekee siitä huomattavasti tehokkaamman. (Drakos 1993.)

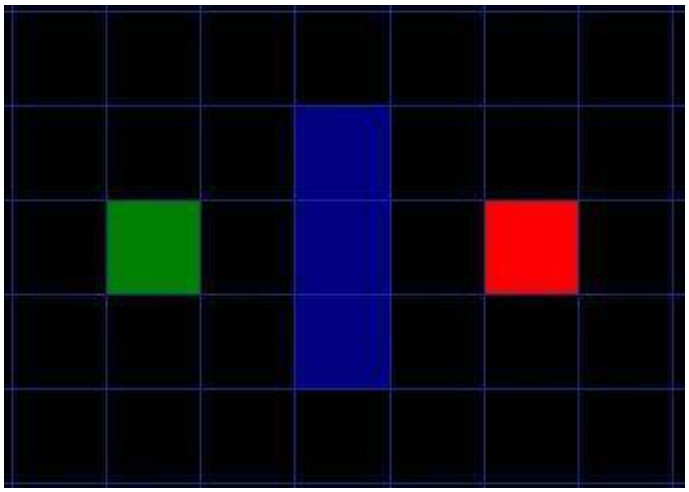
3 A*-ALGORITMI

3.1 Kartan rakenne

Reitinhaun helpottamiseksi hakualue jaetaan monesti neliön muotoisiin alueisiin. Näin saadaan käyttöön yksinkertainen kaksiulotteinen taulukko. Taulukon jokainen solu kuvastaa algoritmin käyttämiä elementtejä eli solmuja. Tämä on kaikista yksinkertaisin malli, mutta kartta voisi rakentua myös erilaisista elementeistä, esimerkiksi kuusikulmioista (heksagoneista) tai kolmioista. Solmussa voi olla este, jolloin se ei ole kulkukelpoinen. (Matthews, Pinter & Scutt 2002, 105.)

3.2 Avoin lista ja suljettu lista

A*-algoritmissa käytetään käsitteitä avoin lista (open list) ja suljettu lista (closed list). Suljettu lista sisältää jo tutkitut solmut, ja avoimessa listassa ovat solmut, jotka odottavat tutkimista. Voidaan ajatella, että avoin lista on alueen kasvava ulkoreuna ja suljettu lista sisälle jäävä osa. Aloitustilanteessa (KUVIO 8) avoimessa listassa on ainoastaan reitin aloitussolmu ja suljettu lista on tyhjä. (Matthews, Pinter & Scutt 2002, 106–107.)



KUVIO 8. A* esimerkin aloitustilanne. Reittiä haetaan vihreästä ruudusta punaiseen. Sininen on este (Lester 2005.)

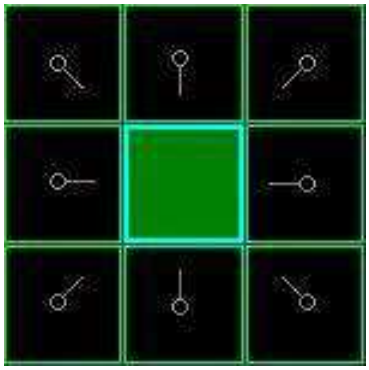
3.3 Ensimmäinen vaihe

Algoritmin pääsilmukan ensimmäisessä vaiheessa valitaan tutkittava solmu avoimesta listasta. Kuten on aikaisemmin mainittu, tässä vaiheessa avoimessa listassa on vain aloitussolmu, joten se valitaan. Tämä solmu poistetaan avoimesta listasta ja lisätään suljettuun listaan. (Lester 2005.)

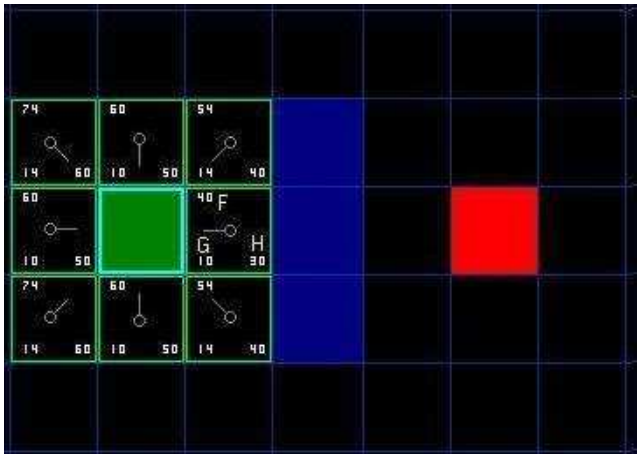
3.4 Toinen vaihe

Seuraavana kaikki tutkittavan solmun ympärillä olevan solmut lisätään avoimeen listaan, lukuun ottamatta niitä, joihin on määritetty este tai jotka ovat suljetussa tai avoimessa listassa. Jokaiselle lisätylle solmulle asetetaan isäntäsolmu (KUVIO 9), joka on sen hetkinen tutkittava solmu, sekä lasketaan F(n)-arvo (KUVIO 10). F(n):ssä on laskettuna yhteen aloituspisteestä kuljettu matka, G(n) sekä arvioitu matka päätesolmuun, H(n). Mikäli jokin ympäröivistä solmuista on jo valmiiksi avoimessa listassa, verrataan sen G(n)-arvoa tutkittavan solmun G(n)-arvoon. Jos todetaan, että nykyisen solmun arvo on pienempi eli matka aloituspisteestä on ly-

hyempi, voidaan päätellä, että sen kautta kulkeva reitti on edullisempi. Tässä tapauksessa viereiselle solmulle lasketaan uusi F(n)-arvo ja nykyinen solmu asetetaan sen isäntäsolmuksi. (Lester 2005.)



KUVIO 9. Ruutujen keskustasta lähtevät viiat osoittavat isäntäsolmuun (Lester 2005.)

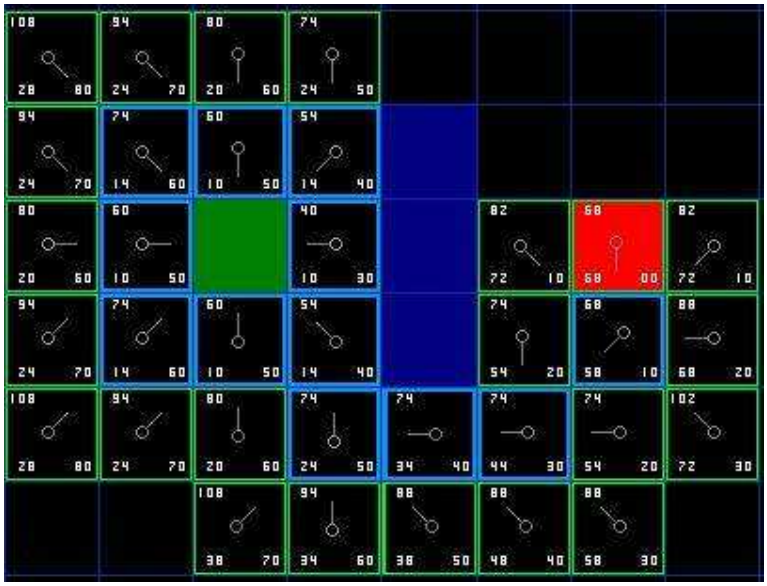


KUVIO 10. Ruutujen vasemmassa alanurkassa on G(n), oikeassa alanurkassa H(n) ja ylänurkassa yhteenlaskettuna F(n) (Lester 2005.)

Jos avoin lista on lajiteltuna F(n)-arvon perusteella, lista on hyvä lajitella uudelleen.

3.5 Kolmas vaihe

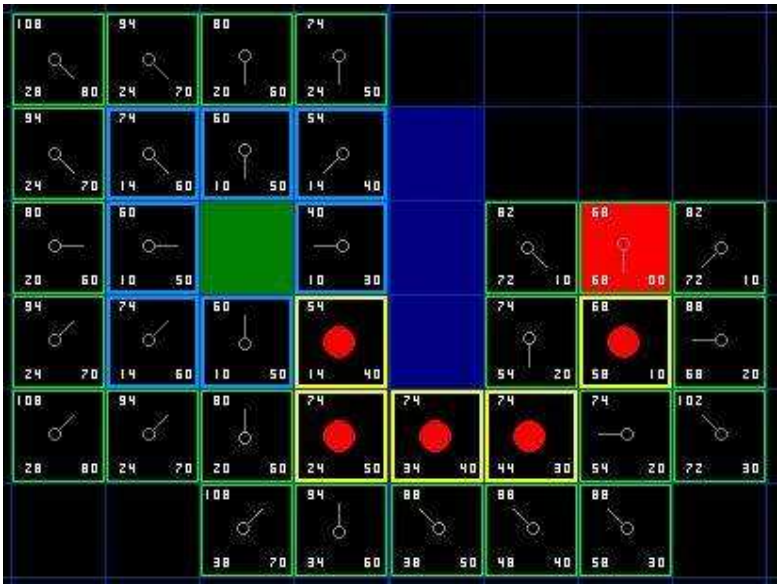
Palataan silmukan ensimmäiseen vaiheeseen. Silmukka päätetään, kun suljettuun listaan lisätty solmu on kohdesolmu, jolloin reitti on löydetty. Jos avoin lista on tyhjä eikä kohdesolmua ole löydetty, reittiä ei ole olemassa. Kun ensimmäisessä vaiheessa valitaan seuraavaa tutkittavaa solmua avoimesta listasta, valintakriteerinä on solmun F(n)-arvo, eli valitaan solmu, jonka arvo on pienin. (Lester 2005.)



KUVIO 11. Kohdesolmu on löydetty (Lester 2005.)

3.6 Lopetus

Kun reitti on löytynyt, se talletetaan (KUVIO 12). Lopullinen reitti saadaan lähtemällä kohdesolmusta ja siirtymällä aina isäntäsolmuun, kunnes ollaan aloitussolmussa. (Lester 2005.)



KUVIO 12. Lopullinen reitti muodostetaan lähtemällä kohdesolmusta ja seuraamalla isäntäsolmuja (Lester 2005.)

3.7 F(n), G(n) ja H(n)

Funktion F(n) laskemiseen käytetään siis kuljettua matkaa aloitussolmusta G(n) ja arvioitua matkaa tutkittavasta solmusta kohdesolmuun H(n). (Matthews, Pinter & Scutt 2002, 106.)

$$F(n) = G(n) + H(n)$$

G(n) kasvaa sitä mukaa, kun tutkittava reitti etenee. Mikäli viistossa oleviin solmuihin on mahdollista liikkua, lisättävän arvon tulisi olla suurempi oikeassa suhteessa. Esimerkiksi jos ylös, alas ja sivuille liikkuminen kasvattaa arvoa yhdellä, viistoon liikuttaessa vastaava arvo on √2. Lopullinen arvo saadaan tutkittavan solmun isäntäsolmun G(n) arvosta, jota kasvatetaan sen mukaan, mihin suuntaan liikutaan. Liikkuminen tapahtuu isäntäsolmusta tutkittavaan solmuun. (Lester 2005.)

H(n):n arvioimiseen voidaan käyttää erilaisia heuristisia funktioita. Funktion valinnalla on suora vaikutus algoritmin tehokkuuteen ja siihen, onko löydetty reitti todellakin lyhyin. On hyvä muistaa, että kohdesolmuun voi olla useita reittejä, jotka ovat yhtä pitkiä. Mitä lähempänä arvioitu matka on todellista jäljellä olevaa matkaa, sitä tehokkaampi algoritmi on. Jos matka yliarvioidaan, niin löydetty reitti ei välttämättä

ole lyhyin. Jotta algoritmi olisi ”luvallinen” (admissible), matka täytyy aina aliarvioida. (Lester 2005.)

3.8 Pääfunktiot

Avoimen listan käsittely vaikuttaa olennaisesti suorituskykyyn. Listan käsittelyssä on kolme pääfunktiota: alimman F(n):n omaavan solmun etsiminen ja poistaminen, tarkistus onko solmu listassa, ja uusien solmujen lisääminen. Lisäksi joissain tapauksissa solmun F(n)-arvoa joudutaan muuttamaan, jolloin solu ensin poistetaan listasta ja lisätään listaan uudestaan. Tätä toimintaa tarvitaan, kun havaitaan, että toisen solmun kautta kuljettaessa G(n) on pienempi. (Lester 2003.)

3.9 Tiedon tallentaminen

Tiedon tallentamiseen voidaan käyttää useita erilaisia tietorakenteita. Yksinkertaisin lähestymistapa on perinteiset taulukot. Ne ovat kuitenkin huono ratkaisu suorituskykyä ajatellen, sillä monet funktiot vaativat koko taulukon läpikäyntiä jokaisella kerralla. (Lester 2003.)

Tieto voidaan varastoida myös käyttämällä listoja, joko lajiteltuja tai lajittelemattomia. Lajittelematonta listaa käytettäessä joudutaan käymään läpi kaikki solmut pienintä arvoa haettaessa. Solmujen lisääminen onnistuu nopeasti, koska solmu voidaan lisätä mihin kohtaan tahansa, mutta sen poistaminen on hidasta. (Lester 2003.)

3.10 Tiedon lajittelu

Lajitellun listan käyttö parantaa tehokkuutta huomattavasti. Listaan lisääminen on jonkin verran hitaampaa, koska oikea kohta täytyy etsiä. Toisaalta pienimmän ar-

von poistaminen on nopea toimitus, sillä se suoritetaan yksinkertaisesti poistamalla listan ensimmäinen solmu. (Smith 2004, 418.)

Lajittelualgoritmit voidaan jakaa vakaisiin eli stabiileihin ja epävakaisiin. Vakaa algoritmi ei vaihda keskenään samansuuruisia alkioita. Toinen luokittelutapa on muistinkulutus. Tämä tarkoittaa sitä, että osa algoritmeista toimii niin sanotusti paikallaan, eli muistia tarvitaan vain sen verran kuin alkioille on jo valmiiksi varattuna. Muussa tapauksessa sitä voidaan joutua varaamaan väliaikaisesti lisää. (Smith 2004, 418.)

3.10.1 Kuplalajittelu

Kuplalajittelussa (bubble sort) listaa käydään yhä uudestaan läpi ja joka kerta verrataan kahta solmua, jotka asetetaan oikeaan järjestykseen. Jos vaihdettavia solmuja ei löydy, niin lista on oikeassa järjestyksessä. Kuplalajittelu on erittäin hidas, koska se vaatii listan jatkuvaa läpikäyntiä. Lajittelutapa on saanut nimensä siitä, että solmut siirtyvät eli ”kuplivat” kohti oikeaa paikkaa. Solmujen järjestyksellä on suuri vaikutus nopeuteen. Jos listan lopussa on pieniä arvoja, niiden siirtyminen alkuun vaatii suuren määrän tarkistuksia. Käytännössä kuplalajittelu on liian hidas käytettäväksi suurien joukkojen lajittelussa, elleivät alkiot ole jo valmiiksi suunnilleen oikeassa järjestyksessä. (Sherrod 2007, 93.)

3.10.2 Valintalajittelu

Toinen yksinkertainen lajittelutapa on valintalajittelu (selection sort). Lajittelun jokaisella kierroksella etsitään listasta pienin arvo. Ensimmäisellä kerralla löydetty alkio ja listan ensimmäinen alkio vaihdetaan keskenään. Toisella kierroksella vaihto tehdään toisena olevan alkion kanssa ja niin edelleen. Näin lista jakautuu kahteen osaan, alussa olevaan lajiteltuun ja lopun lajittelemattomaan osaan. Valintalajittelussa alkioiden järjestys ei vaikuta tehokkuuteen, koska koko lista joudutaan joka tapauksessa käymään läpi, jotta pienin jäljellä olevista arvoista löytyy. Loppua

kohti nopeus kasvaa, sillä tutkittavien solmujen määrä pienenee joka kierroksella. Valintalajittelu on nopeampi pienillä listoilla kuin eräät isojen listojen käsittelyyn soveltuvat algoritmit, esimerkiksi pikalajittelu. (Sherrod 2007, 98.)

3.10.3 Lisäyslajittelu

Lisäyslajittelu (insertion sort) toimii valitsemalla listan lajittelemattomasta osasta solmu ja lisäämälle se oikealle paikalle lajiteltuun osaan. Lajittelu on tehty, kun lajittelemattomassa osassa ei enää ole solmuja jäljellä. Käsiteltävän solmun valinta voidaan tehdä periaatteessa millä tahansa algoritmilla, mutta yleensä kuitenkin valitaan listan ensimmäinen alkio. Oikean paikan etsiminen lajitellussa osassa voidaan toteuttaa esimerkiksi aloittamalla viimeisestä alkiosta ja siirtämällä alkioita yksi kerralla loppua kohti, kunnes sopiva paikka löytyy. Nopeimmillaan lisäyslajittelu on pienellä listalla, joka on jo valmiiksi lajiteltu, ja hitaimmillaan, mikäli listan alkiot ovat lajiteltuina suurimmasta pienimpään. (Sherrod 2007, 101.)

3.10.4 Shell-lajittelu

Vuonna 1959 julkaistu Shell-lajittelu (shell sort) on kehitetty lisäyslajittelun pohjalta. Ideana on verrata kahta lukua, jotka ovat tietyn etäisyyden päässä toisistaan. Ensimmäisellä kierroksella verrattava luku voi olla esimerkiksi puolivälissä listaa. Kun ensimmäisiä lukuja on verrattu ja mahdollisesti vaihdettu oikeaan järjestykseen, siirrytään listassa yhdellä indeksillä eteenpäin. Näin tehdään, kunnes toinen verrattavista luvuista on listan viimeinen luku eikä eteenpäin voi enää siirtyä. Tämän jälkeen palataan takaisin listan alkuun ja tehdään sama vertailu; tällä kertaa kuitenkin pienennetään vertailtavien lukujen etäisyyttä puolella. Viimeisellä kierroksella verrataan viereisiä lukuja. Shell-lajittelun etu lisäyslajitteluun verrattuna on, että luvut pystyvät ottamaan kerralla isompia harppauksia kohti oikeaa paikkaa. Lajittelun toimintaa voidaan säätää vaihtamalla lajiteltavien solujen etäisyyttä, sillä etäisyyden ei välttämättä tarvitse olla alussa puolet taulukon pituudesta. Vii-

meisellä kierroksella etäisyys on 1 ja käytetään normaalia lisäyslajittelua. (Sherrod 2007, 255.)

3.10.5 Pikalajittelu

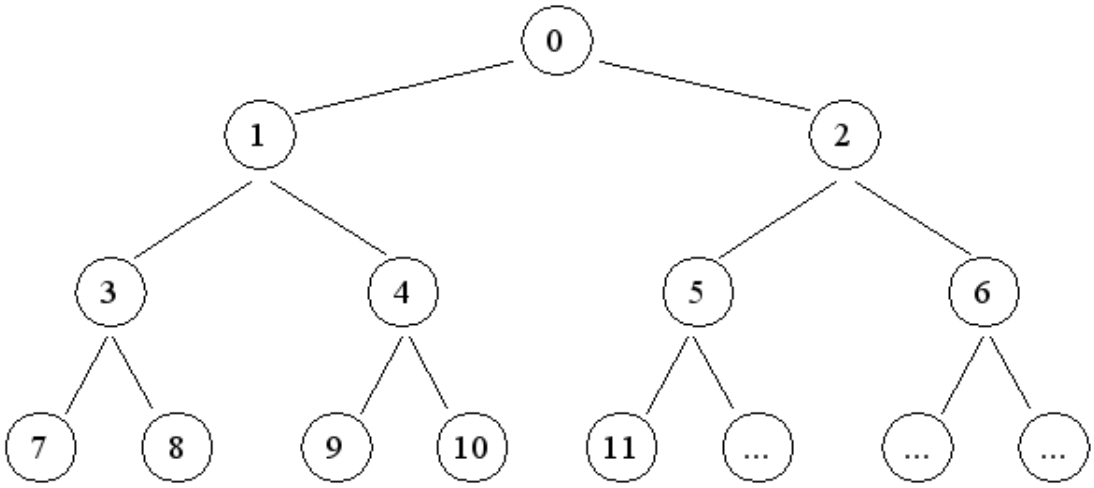
Pikalajittelu (quicksort) on nopea lajittelualgoritmi lähes kaikilla suorittimilla. Algoritmin periaatteena on jakaa lajiteltava alue joka kerralla kahteen osaan. Jakamisessa käytetään niin sanottua sarana-alkiota. Alkiot, jotka ovat sarana-alkiota pienempiä, siirretään ensimmäiseen ryhmään listan alkupäähän. Isommat alkiot siirretään loppupäähän. Tämän jälkeen tehdään sama listan molemmille osille kutsumalla samaa funktiota rekursiivisesti. Mikäli listan koko on 1 tai 0, niin listan oletetaan olevan jo valmiiksi lajiteltu. Listan jakamista useampaan osaan voidaan hyödyntää, mikäli käytössä on useampia suorittimia. Koska sarana-alkion vasemmalla puolella ovat kaikki pienemmät ja oikealla puolella suuremmat, sen tiedetään olevan oikealla paikalla. Näin listan molemmat osat pysyvät samanpituisina ja rinnakkainen käsittely on mahdollista. Moniajossa voidaan avata uusi säie jokaiselle käsiteltävälle jonon osalle. Joissain tapauksissa, esimerkiksi jos kaikki alkiot ovat samanarvoisia, pikalajittelun suoritusaika voi kasvaa huomattavasti. Tällaisia tilanteita ajatellen voi olla parempi käyttää satunnaisuutta sarana-alkion paikan valitsemisessa. (Sherrod 2007, 265.)

3.11 Binäärikeko

Edellä mainittuja lajittelualgoritmeja nopeampi tapa on kuitenkin käyttää binäärikekoa. Binäärikeon toteuttaminen on vaikeampaa kuin perinteisten hakujen, mutta ero tehokkuudessa saattaa olla huomattava erityisesti suurilla kartoilla. Binäärikeossa pienimmän alkion poisto ja uuden lisääminen tapahtuu ajassa O(log N), missä N on rakenteessa olevien alkioiden määrä. (Lester 2003.)

3.11.1 Rakenne

Binäärikeko on puurakenne (KUVIO 13), joka tallennetaan normaaliin taulukkoon, toisin kuin puurakenteet yleensä. Tavallisesti puissa käytetään osoittimia lapsisolmuihin viittaamiseen. Binäärikeossa sen sijaan käytetään indeksinumeroita. Normaaleissa lajitelluissa listoissa kaikki alkiot ovat oikeassa järjestyksessä, esimerkiksi alkaen pienimmästä ja päättyen suurimpaan. A*-algoritmia ajatellen tämä ei kuitenkaan ole välttämätöntä, sillä yleensä tarkoituksena on vain löytää solmu, jolla on pienin F(n) arvo. Binäärikeossa haettava arvo, oli se sitten pienin tai suurin, on päällimmäisenä. Tässä tapauksessa oletetaan, että arvo on pienin. Kyseisellä solmulla on kaksi lapsisolmua, jotka molemmat ovat yhtä suuria kuin isäntäsolmu tai isompia. Myös näillä solmuilla on vastaavasti omat lapsisolmunsa. (Lester 2003.)

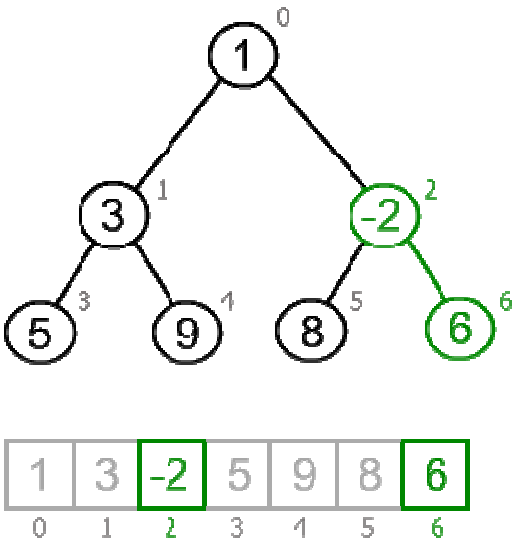


KUVIO 13. Binäärikeko

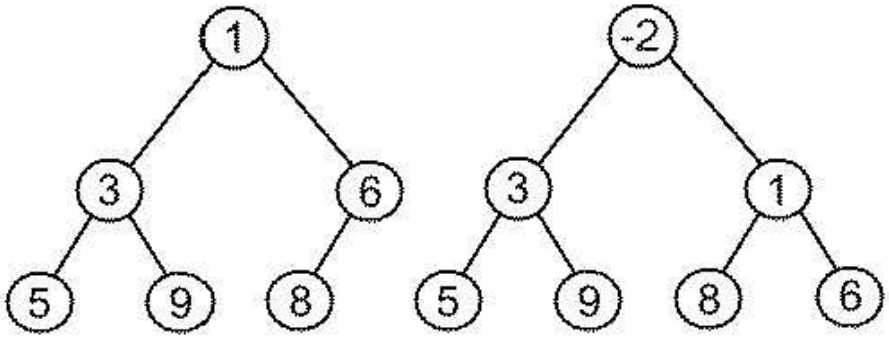
Yksi binäärikeon ominaisuus on, että se voidaan tallettaa normaaliin kaksiulotteiseen taulukkoon. Päällimmäinen solmu talletetaan taulukon ensimmäiseen alkioon, tämän lapsisolmut kahteen seuraavaan ja niin edelleen. Taulukosta minkä tahansa solmun lapsisolmut löytyvät helposti kertomalla isäntäsolmun binäärikeon indeksinumero kahdella. Tästä saatava luku osoittaa solmuparin ensimmäiseen solmuun, toinen saadaan lisäämällä lukuun 1. (Lester 2003.)

3.11.2 Solmun lisääminen

Lisättäessä solmua binäärikekoon se asetetaan taulukon loppuun (KUVIO 15). Uuden solmun isäntäsolmun paikka löytyy jakamalla solmujen kokonaismäärä kahdella (KUVIO 14). Uutta solmua verrataan isäntäsolmuun. Mikäli uudella solmulla on pienempi F(n)-arvo, se vaihtaa paikkaa isäntäsolmunsa kanssa. Tämän jälkeen tehdään sama vertailu uuden isäntäsolmun kanssa, joka löytyy jakamalla solun sen hetkinen indeksinumero kahdella. Jos F(n)-arvo on pienempi, tehdään uusi vaihto. Tätä toistetaan, kunnes löytyy paikka, jossa isäntäsolmun F(n) ei ole pienempi tai kunnes keon huippu on saavutettu. (Lester 2003.)



KUVIO 14. Binäärikeko ja arvojen sijainti taulukossa



KUVIO 15. Solmun lisääminen binäärikekoon

3.11.3 Solmun poistaminen

Kun päällimmäinen solmu halutaan poistaa, listan viimeinen solmu siirretään sen tilalle. Päällimmäiseksi siirrettyä solmua verrataan sen uusiin lapsisolmuihin. Ensimmäinen solmu verrattavasta lapsiparista löytyy kertomalla viimeisen solun nykyinen paikka kahdella, toinen saadaan kasvattamalla lukua vielä yhdellä. Jos siirretyn luvun F(n) on pienempi kuin kummankin lapsisolmun, se pysyy paikoillaan. Muussa tapauksessa se vaihtaa paikkaa pienemmän lapsisolmun kanssa. Samaa toistetaan, kunnes löytyy paikka, jossa molemmat lapsisolmut ovat suurempia tai saavutetaan keon pohja. (Lester 2003.)

3.11.4 Hyöty

A*-algoritmissa käytettävä avoin lista voi kasvaa hyvin suureksi jopa pienillä kartoilla. Mitä suurempi kartta on, sitä enemmän binäärikeolla on vaikutusta tehokkuuteen. On kuitenkin huomattava, että pienillä kartoilla saavutettava hyöty on pienempi, etenkin kun yksinkertaisemmat hakutavat ovat huomattavasti helpompia toteuttaa. Binäärikekoa käytettäessä solmujen hakeminen, lisääminen ja poistaminen voivat nopeutua moninkertaisesti. Listasta alimman F(n):n omaava solmu löytyy helposti, koska se on keossa päällimmäisenä. Tätä funktiota käytetään jatkuvasti, kun tutkittavia solmuja poistetaan avoimesta listasta ja lisätään suljettuun listaan.

Kekoa varten luotavan taulukon tarvitsee olla maksimipituudeltaan kartan leveys x korkeus, eli pahimmassa tapauksessa voidaan joutua käymään läpi koko kartta. Tämä pätee ainakin kaksiulotteisella kartalla. Taulukko kuvastaa algoritmin avointa listaa.

3.11.5 Tunnistenumerot

A*-algoritmia ajatellen ei kuitenkaan riitä, että keosta löytyvät ainoastaan solmujen F(n)-arvot, koska ei tiedetä, mitä kartan koordinaattia ne vastaavat. Tästä syystä kekoon voidaan tallettaa varsinaisten F(n)-arvojen sijasta tunnistenumerot, joiden perusteella oikeat arvot löytyvät. Itse arvot voidaan tallettaa normaaliin yksiulotteiseen taulukkoon. Tunnisteen lisäksi tallennetaan solmujen koordinaatit omaan tauluunsa. (Lester 2003.)

3.12 Heuristiset funktiot

A*-algoritmissa heuristisia funktioita käytetään H(n) laskemiseen. Tämä tarkoittaa arvioitua jäljellä olevaa matkaa kohdesolmuun. Käytettävä funktio vaikuttaa olennaisesti lopullisen reittiin. Arviointitekniikoita on tarjolla monenlaisia, ja siksi onkin tärkeää miettiä tarkoin, mikä sopii omaan käyttötarkoitukseen. (Patel 2009; Pramudya 2008, 2.)

3.12.1 Vaikutus

Esimerkiksi jos joka kerta approksimoidaan, että jäljellä oleva matka on 0, A* muuttuu Dijkstran algoritmiksi. Voidaan taata, että lyhyin reitti löytyy mutta hakualue laajenee tasaisesti joka suuntaan ja tarkistuksien määrä kasvaa huomattavasti. (Patel 2009.)

Jos matka arvioidaan oikein joka kerta, A* löytää aina lyhyimmän reitin ja on hyvin nopea. Oikean matkan arviointi on kuitenkin yleensä mahdollista vain erityistapauksissa. (Patel 2009.)

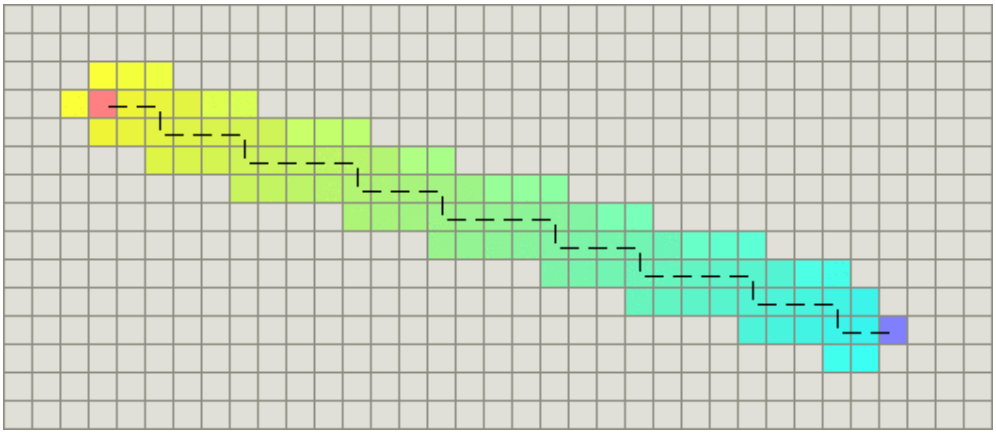
Jos matka arvioidaan joskus liian suureksi, lyhyintä reittiä ei aina löydetä. Tällöin A* ei enää ole ”luvallinen”. Tämä voidaan joskus ohjelmiston käyttötarkoituksen mukaan hyväksyä. A* on tiettyyn rajaan asti sitä nopeampi, mitä suuremmaksi kuljettava matka arvioidaan. Jos jäljellä oleva matka arvioidaan aina hyvin suureksi suhteessa kuljettuun matkaan, A* muuttuu paras ensin -hauksi, eli se pyrkii aina liikkumaan suoraan kohti kohdesolmua. (Patel 2009.)

3.12.2 Hyöty

Peleissä parhaimman reitin löytäminen ei aina ole välttämätöntä, jolloin riittää, kun löydetään ”tarpeeksi hyvä”. Esimerkiksi jos kartalla on erilaisia maastotyyppejä, joiden liikkumiskustannus vaihtelee, voidaan harkita kompromissia nopeimman ja lyhyimmän reitin välillä. Tässä tapauksessa ”lyhyimmällä” tarkoitetaan suoraa etäisyyttä, kun taas ”nopein” on se reitti, jolla on pienin kustannusarvo. Lyhyimmän reitin suosiminen nopeimman edelle on hyväksi myös laskentanopeutta ajatellen. Kuten aiemmin mainittiin, A* voidaan säätää suosimaan lyhyempää reittiä yliarvioimalla jäljellä oleva matka. Heikommilla suorittimilla laskentatehoa voidaan säästää nostamalla kustannusarvoja dynaamisesti. Sovelluksessa on hyvä olla säätömahdollisuus reitinhaun tarkkuuden ja nopeuden välillä. Yksi vaihtoehto on asettaa jokaiselle hakijalle prioriteetti hakijoiden kokonaismäärän ja tärkeyden mukaan. Jos H(n) arvioidaan aina oikein, A*:n voidaan olettaa käyttäytyvän lähes täydellisesti. Periaatteessa on mahdollista laskea etukäteen reittien pituudet jokaisesta solmusta kaikkiin muihin solmuihin. Tällöin A* laajentaa hakuaan ainoastaan oikeaan suuntaan ja algoritmista tulee hyvin nopea. Käytännössä tämä ei yleensä kuitenkaan ole mahdollista, pelkästään jo muistinkulutusta ajatellen. (Patel 2009.)

3.12.3 Manhattan-etäisyys

Yleinen tapa on käyttää Manhattan-etäisyyttä (KUVIO 16). Manhattan-etäisyys eli korttelietäisyys saadaan laskemalla yhteen nykyisen solmun ja kohdesolmun vaaka- ja pystykoordinaattien erotuksien itseisarvot. (Patel 2009.)



KUVIO 16. Manhattan-heuristiikka (Patel 2009.)

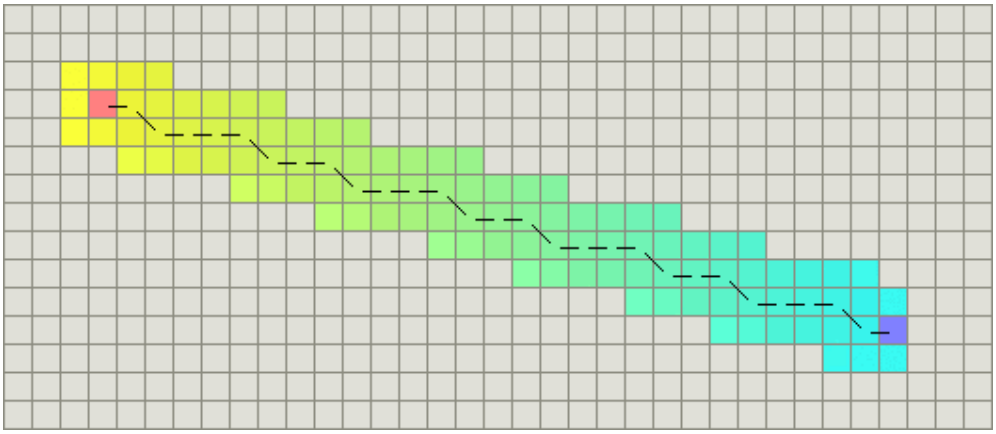
Ruudukkopohjaisella kartalla ruutujen välillä käytetään yleensä minimikustannusta. Joka tapauksessa käytettävä kustannus täytyy olla skaalattuna kartan muihin kustannuksiin, jotta ei synny vääristymiä. Manhattan-etäisyys on yksinkertainen tapa arvioida etäisyys, mutta ei kuitenkaan niin sanottu ”luvallinen” tapa, koska sitä käytettäessä matka yleensä yliarvioidaan. Jotta A* olisi luvallinen, matkaa ei saa koskaan yliarvioida. Manhattan-etäisyyttä tulisi käyttää vain, jos solmujen välillä ei voi liikkua viistottain. Tämä heuristiikka on kevyt, erityisesti jos kustannusarvoina käytetään vain kokonaislukuja. (Patel 2009.)

3.12.4 Viistoon liikkuminen

Jos kartalla voi liikkua viistoon, on hyvä käyttää jotain muuta heuristiikkaa kuin Manhattan-etäisyyttä, muuten tietyissä tapauksissa syntyy vääristyneitä reittejä. Esimerkiksi jos lähtösolmun ja kohdesolmun välinen vaakaetäisyys on suurempi

kuin pystyetäisyys, A* pyrkii ensin liikkumaan viistoon samalle vaakatasolle, josta se jatkaa suoraan vaakasuorasti kohteeseen. Mikäli suoraan ja viistoon liikkumisen kustannusarvot ovat samat, voi riittää, että käytetään yhteenlaskussa vain suurempaa etäisyyttä. (Patel 2009.)

Sen sijaan jos viistoon liikkumisen kustannus on suurempi kuin suoraan liikkumisen, mikä on välttämätöntä, jos halutaan saada realistisia reittejä, tarvitaan hieman enemmän laskemista (KUVIO 17). Ensin verrataan pisteiden vaaka- ja pystyetäisyyksiä ja valitaan niistä pienempi, josta saadaan arvio siitä, montako ruutua täytyy vähintään liikkua viistoon. Toisena lasketaan normaali Manhattan-etäisyys. Jokaista suoraan liikuttua solmua kohti vähennetään kaksi viistoon liikuttua solmua. Lopputulos saadaan laskemalla nämä yhteen. (Patel 2009.)

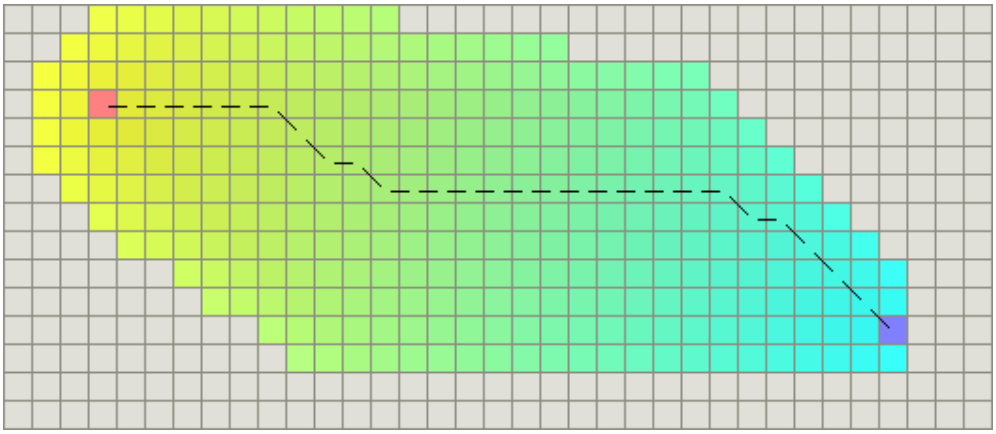


KUVIO 17. Viistoon liikkuminen (Patel 2009.)

3.12.5 Euklidinen etäisyys

Euklidinen etäisyys on niin sanottu ”tavallinen” etäisyys kahden koordinaatin välillä (KUVIO 18). Tätä voidaan käyttää, jos kartalla pystyy liikkumaan mihin suuntaan tahansa eikä pelkästään ruutujen keskustojen kautta. Ongelmana on kuitenkin se, että kuljettu etäisyys G(n) on saatu laskemalla etäisyydet ruutujen keskipisteiden kautta. Periaatteessa F(n) vääristyy, koska se saadaan laskemalla G(n) + H(n). Toisaalta euklidinen etäisyys on aina lyhyempi kuin todellinen kuljettava etäisyys,

joten heuristiikka on luvallinen. Etäisyyden laskemiseen tarvitaan neliöjuurta, mikä tekee tästä heuristiikasta melko raskaan. (Patel 2009.)



KUVIO 18. Euklidinen etäisyys (Patel 2009.)

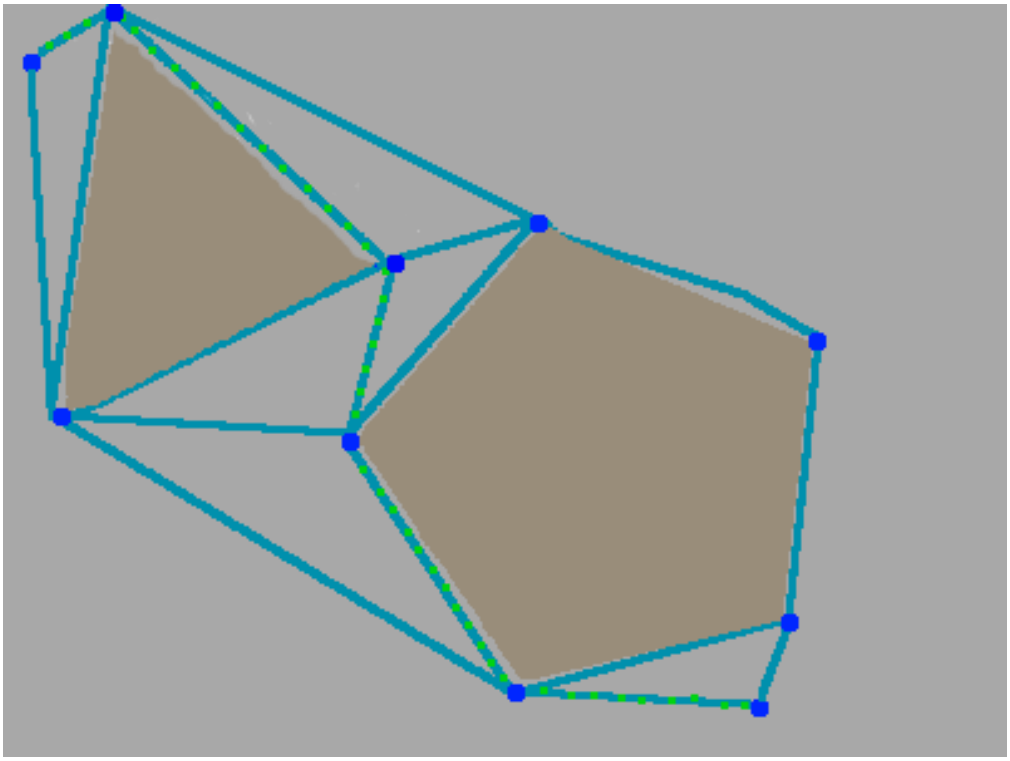
4 TIETORAKENTEET JA KARTTATYYPIT

4.1 Ruudukko

Kaksiulotteinen ruudukkopohjainen kartta on yksi yleisimmin käytetyistä A*-karttatyypeistä. A* on kuitenkin suunniteltu käytettäväksi hyvinkin erilaisilla tietorakenteilla. Ruutupohjaisella kartalla jokainen ruutu kuvastaa solmua. Yleensä ruutujen oletetaan olevan samankokoisia ja symmetrisiä. Ruudukkopohjaisen kartan solut voivat olla paitsi neliöitä myös heksagoneja tai kolmioita. Symmetristen ruudukkojen etuna on niiden selkeys ja käsittelyn helppous. (Patel 2009.)

4.2 Monikulmiot

Jos oletetaan, että kartalla ei ole erilaisia maastotyyppejä, eli kustannukset ovat vakioita sama missä liikuttaessa, pohjana voidaan käyttää monikulmioita (KUVIO 19). Reitillä olevat esteet muodostetaan omilla monikulmioillaan. Tällaisella kartalla reitit muodostetaan ennalta määriteltyjen reittipisteiden välille. Reittipisteet ovat algoritmin solmuja. Reittipisteiden lisäksi täytyy määrittää, mistä solmusta voi kulkea mihin ja mikä on niiden välinen kustannus. Koska kartalla ei ole muuttuvia maastokustannuksia, kustannuksena voidaan käyttää etäisyyttä. Reittipisteiden käyttäminen on huomattavasti nopeampaa kuin normaalin ruudukkopohjaisen kartan. (Patel 2009.)



KUVIO 19. Monikulmiot esteinä (Patel 2009.)

4.3 Hierarkiat

Yksi reitinhaun ongelma on se, että kun etäisyys kohteeseen kasvaa, niin reitin laskemiseen käytettävä aika kasvaa moninkertaisesti. Suhde ei siis ole lineaarinen. Tutkittavan alueen pienentämiseksi se voidaan jakaa hierarkkisiin tasoihin. Sen sijaan, että reitti tutkittaisiin matalalla tasolla solmu solmulta, käytetään suurempia kokonaisuuksia. Esimerkiksi liikuttaessa kaupunkitasolla reitti haetaan liikkumalla katuja pitkin. Kadulta voidaan siirtyä talotasolla, jossa liikutaan huoneiden välillä. Kaupunkien välillä voi olla määriteltynä ”moottoriteitä”, joita käyttämällä löydetään helposti nopein reitti. (Patel 2009.)

Kartta jaetaan mahdollisesti pienempiin alueisiin ja ylemmällä tasolla reitti haetaan ensin niiden väliltä. Alemmalla tasolla käytetään normaalia ruudukkopohjaista hakua. Kun saavutaan uudelle alueelle, haetaan uusi reitti seuraavan alueen reunaan. (Patel 2009.)

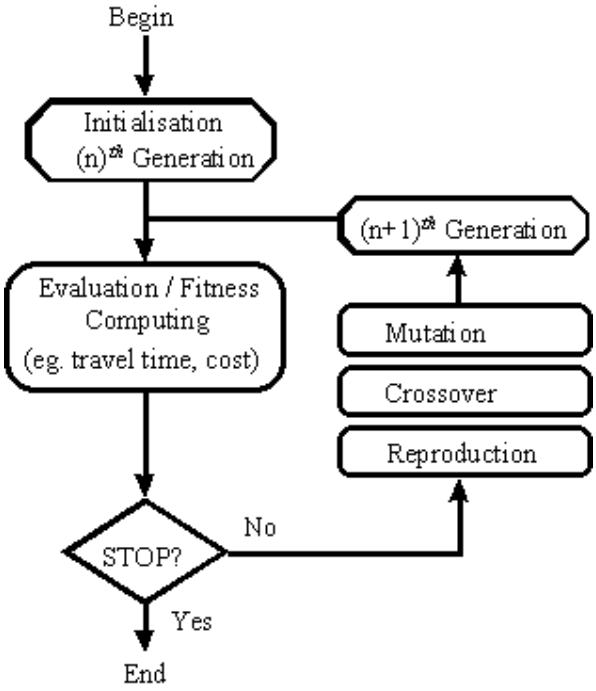
5 GENEETTISET ALGORITMIT

5.1 Evoluutio

Geneettinen algoritmi (KUVIO 20) perustuu luonnossa tapahtuvaan evoluutioon. Sillä voidaan kehittää ratkaisuja monenlaisiin ongelmiin ja sitä käytetään muun muassa oppivissa järjestelmissä. Algoritmilla simuloidaan populaation kehittymistä, jossa parhaiden geenien yhdistelmät selviytyvät ja säilyvät eteenpäin. (Tyni & Ylinen 1999, 1.)

5.2 Satunnainen populaatio

Lähtökohtana on satunnainen populaatio. Tästä satunnaisesta populaatiosta valitaan parhaat kromosomiparit, jotka yhdistetään ja joista muodostetaan uusi populaatio. Pienellä todennäköisyydellä tapahtuu myös mutaatioita, joilla vältetään väestön liiallinen samankaltaistuminen. Mutaatioita tarvitaan, jotta saadaan uusia ja mahdollisesti entistä parempia ratkaisuvaihtoehtoja. Algoritmi saattaa olla pitkiäkin aikoja juuttuneena paikalliseen maksimiin tai muuhun esteeseen, kunnes mutaation avulla se pääsee taas jatkamaan. (Tyni & Ylinen 1999, 1.)



KUVIO 20. Geneettinen algoritmi (Algers, Bernauer & Boero 1997.)

5.3 Soveltuvuus

Algoritmin mahdollisia ratkaisuja kuvataan kromosomeilla. Jalostamalla saadaan parhaat mahdolliset ratkaisut. Reaaliaikaisissa ohjelmistoissa ongelmana on tarvittava laskentojen määrä, joka kasvaa samassa suhteessa kuin populaation koko. Tämän takia suurin osa geneettisiä algoritmeja käyttävistä sovelluksista ei ole reaaliaikaisia. Myös ohjausjärjestelmissä varsinainen geneettinen algoritmi on yleensä itse järjestelmän ulkopuolella. Tyypillisiä käyttötarkoituksia ovat muun muassa erilaiset optimointi- ja suunnittelutehtävät. Ollessaan osana ohjausta algoritmi saattaa tuottaa välillä mahdottomia tai epäkäytännöllisiä ratkaisuja. Näitä voidaan karsia rajoittamalla hyväksyttävien ratkaisujen määrää. Toinen tapa on antaa ”sakkoja” niille ratkaisuvaihtoehdoille, jotka rikkovat rajoituksia, jolloin ne tulkitaankin huonoiksi ratkaisuiksi. (Tyni & Ylinen 1999, 1.)

5.4 Stabiilius ja toimivuus

Geneettisellä algoritmilla toimivan järjestelmän stabiilius voi olla epävarmaa. Perinteisissä järjestelmissä saadaan yleensä selkeämpiä arvoja, joita analysoidaan tasaisin välein ja muutetaan ohjauskäskyiksi. Geneettisessä menetelmässä taas voi esiintyä sen monimutkaisuuden takia paikallisia ääriarvoja, jotka vääristävät tuloksia, koska selkeää globaalia ääriarvoa ei välttämättä ole. Samoihin tuloksiin ei päästä joka kerta, mikä täytyy ottaa huomioon algoritmin suunnittelussa. Muuten järjestelmässä voi esiintyä epävakautta ja toimivuus voi olla ennalta arvaamatonta. Geneettisen algoritmin tehokkuutta voidaan parantaa rinnakkaisajolla. Algoritmia on myös kehitetty eteenpäin, mutta monesti esitetyt tavat lisäävät sovellusriippuvuutta. Reitinhaussa geneettisen algoritmin tehtävä on löytää reitti, jonka kustannus on mahdollisimman pieni. (Tyni & Ylinen 1999, 2.)

6 REAALIAIKAISUUS

6.1 Dynaamiset kartat

Realistinen reitinhaku dynaamisilla kartoilla voi olla hyvin haastava toteuttaa. Esimerkiksi monissa peleissä käytetään monimutkaisia kolmiulotteisia karttoja, jotka muuttuvat reaaliajassa. Tässä tulee mukaan esteiden väistely. Kiinteiden esteiden lisäksi on dynaamisia esteitä, joiden sijainti vaihtelee. Yleisimmin käytetyt yksinkertaistetut karttamallit ovat epäkäytännöllisiä tähän tarkoitukseen, mikä johtuu lisääntyvästä laskennasta. Tästä syystä nämä kartat esikäsitellään ja ladataan muistiin, jotta niitä olisi helpompi käsitellä ajon aikana. Reitinhaku käyttää staattista versiota, joka ladattiin ohjelman käynnistyessä. Tämä menettely ei kuitenkaan toimi nopeasti muuttuvilla dynaamisilla kartoilla. (Koenig & Likhachev 2002, 1–2.)

6.2 Liike

Realistista liikkumista ajatellen liikkeen täytyy olla myös sulavaa. Ei välttämättä riitä, että liikutaan pelkästään kohtisuoraan tai viistoon solmujen välillä, vaan liikkeen tulisi olla pehmeää. (Matthews, Pinter & Scutt 2002, 186.)

6.3 Taktinen reitinhaku

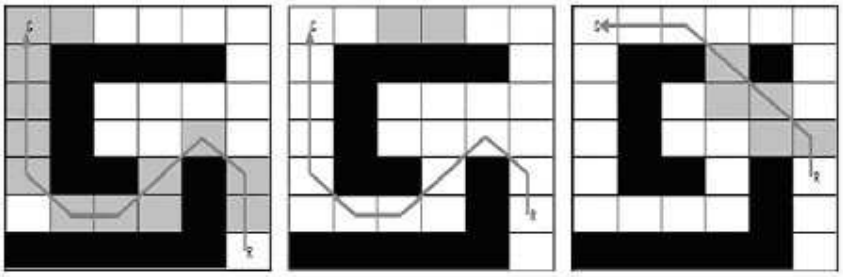
Taktisella reitinhaulla tarkoitetaan reitin etsimistä, jossa otetaan huomioon etäisyyksien ja esteiden lisäksi muita ympäristön tekijöitä, kuten esimerkiksi maaston tarjoama suoja. Lisäksi voi olla ”vaarallisia” alueita, joita tulisi välttää mahdollisuuksien mukaan. Nämä tekijät voidaan lisätä reitinhakuun muokkaamalla kartan kustannuksia tai käytettävää heuristista funktiota. Näidenkin ongelmana on se, että tilanne ei välttämättä pysy staattisena koko ohjelman ajon ajan. (Matthews, Pinter & Scutt 2002, 211.)

6.4 Pääperiaatteet

Reaaliaikaisen reitinhaun pääperiaatteet ovat päämäärää kohti liikkuminen ja staattisten sekä dynaamisten esteiden väistely. Tätä varten haun täytyy pystyä aistimaan ympäristön esteet ja niissä tapahtuvat muutokset. Sen täytyy myös osata reagoida oikealla tavalla. Reaaliaikainen A* pystyy käsittelemään monimutkaisiakin karttoja, sillä sen voidaan taata palauttavan tulokset ennalta määrätyssä ajassa. Toimintaperiaate on sama kuin normaalissa A*-algoritmissa, mutta aikamuuttuja saa sen niin sanotusti katsomaan muutaman askeleen eteenpäin. Jos reitti löytyy aikarajan puitteissa, niin algoritmi toimii normaalisti. Toisaalta jos reittiä ei löydykään, niin se muodostaa reitin sen perusteella, mitä tietoa on saatavilla. Muodostuvaa kantaa päivitetään jatkuvasti. Tämä reaaliaikainen versio on käytännöllinen sovelluksissa, joissa reittejä hakevia objekteja voi olla hyvin paljon. Hakuun käytettävä aika ja tarkkuus voidaan säätää sen mukaan, paljonko objekteja on sillä hetkellä olemassa. Näin estetään reitinhakua muodostamasta pullonkaulaa ohjelman ajossa. Huonona puolena hakemiseen voidaan antaa liian vähän resursseja, jolloin reittien laatu kärsii. Lisäksi tämäkin A*-versio käyttää staattisia karttoja, joten se ei sovellu täysin dynaamiseen ympäristöön. (Koenig & Likhachev 2002, 1–2.)

6.5 D*-algoritmi

A*-algoritmin dynaamista versiota kutsutaan nimellä D* (KUVIO 21). Se soveltuu hyvin käytettäväksi esimerkiksi tietokonepeleissä. Se toimii alustavasti samalla tavalla kuin A* eli laskee reitin aloituspisteestä kohteeseen. Erona on se, miten se reagoi uusiin esteisiin, joihin törmätään liikuttaessa reittiä pitkin. Tässä tapauksessa lasketaan uusi reitti. Tämä algoritmi ei välttämättä ole kovin käytännöllinen reaaliaikaisissa sovelluksissa, sillä se on melko raskas. Pelien lisäksi algoritmia käytetään robotiikassa ja siitä on tehty huomattava määrä tutkimusta. (Ferguson, Gordon & Likhachev 2005, 1–3.)



KUVIO 21. Robotti lähtee liikkeelle oikeasta alanurkasta ja laskee nopeasti suboptimaalisen reitin vasemmassa ylänurkassa olevaan kohteeseen. Laskemisen jälkeen robotti lähtee etenemään ja huomaa ylimmässä seinässä (mustat alueet) läpimentävän aukon. Tämän jälkeen lasketaan uusi reitti päivitetyn tiedon perusteella ja lähdetään seuraamaan sitä (Ferguson, Gordon & Likhachev 2005, 2.)

6.6 Ohjausalgoritmit

Ohjausalgoritmeja (steering algorithm) voidaan sanoa lyhyen matkan reitinhakualgoritmeiksi. Niiden tehtävänä on yleensä esteiden väistely ja reitin seuraaminen. Ohjausalgoritmit antavat reitinhaulle kyvyn toimia ja tehdä päätöksiä reaaliajassa käyttämällä sensoreita tiedon keräämiseen ympäristöstä. Esteiden mallintamisessa niiden voidaan ajatella työntävän ohjattavia objekteja kauemmaksi esteistä, jolloin objektit saadaan varomaan kyseisiä esteitä. Liikkuvan objektin liikerataa muokataan kaikkien esteiden yhteisvaikutuksen mukaan eli kiihtyvyyttä ja nopeutta säädetään vastaavasti. Toinen mahdollisuus on lähettää reitillä liikkujasta ”säteitä” ja liikkua suuntaan, jossa säteet pääsevät pisimmälle. Sensoreiden varassa toimivat algoritmit pystyvät reaaliaikaiseen toimintaan ja toimivat hyvin dynaamisia esteitä ajatellen. Ne voivat reagoida nopeasti ympäristössä tapahtuviin muutoksiin. (Reynolds 1999.)

6.7 Parveilualgoritmit

Parveilualgoritmeja (flocking algorithm) käytetään joukkojen hallintaan. Näissä algoritmeissa käytetään usein myös ohjausalgoritmeja. Perusideana oletetaan, että

parven käyttäytyminen kokonainaisuutena on älykkäämpää kuin yksittäisten jäsenten. Usein yritetään mallintaa eläinparvien käyttäytymistä, mutta niitä voidaan käyttää myös reaaliaikaisissa peleissä. Pelien tapauksessa yksi hyöty on, että peliobjekteille saadaan käyttäytymismalliin ennalta määriteltyjä reittejä. Lisäksi parveilun mallintaminen on laskennan kannalta kevyempää kuin normaali esteiden väistely, kun kyseessä on suuri joukko objekteja. Reaaliaikaisissa strategiapeleissä, kuten myös kolmiulotteisissa toimintapeleissä, käytetään hyvin usein parveilualgoritmeja. Näissä peleissä on monesti hyvin paljon objekteja pelaajan näkökentässä. (Matthews, Pinter & Scutt 2002, 202.)

6.8 Oppiminen

Reitinhakuun voidaan sisällyttää oppimista. Oppivalla reitinhaulla voidaan korvata kokonaan valmiiksi tehdyt ja perinteisillä menetelmillä saadut reitit. Oppivalla reitinhaulla täytyy olla kaksi ominaisuutta: kyky analysoida lähiympäristö reaaliajassa ja muuttaa tämä tieto ohjauskäskyiksi. Oppivissa järjestelmissä voidaan käyttää hermoverkkoa. Hermoverkko (neural network) on oppimisalgoritmi, joka on ”opetettu” suodattamaan halutut arvot annetusta syötteestä. Jos hermoverkko on opetettu oikein, se pystyy yleistämään tilanteet ja toimimaan myös erikoistilanteissa. Tämä on hyödyllistä dynaamisissa ympäristöissä. Häiriönsietokyvyn täytyy olla erittäin hyvä. (Graham, McCabe & Sheridan, 1–2.)

7 ESIMERKKISOVELLUS

Esimerkkinä toimii tekemäni reaaliaikainen taktisen tason sotasimulaatio Armored Brigade. Ajon aikana kartalla voi olla yhtäaikaisesti satoja yksiköitä. Kartan koko on maksimissaan noin 15 x 15 km ja se koostuu 512 x 512 -ruudusta eli solmusta. Jokainen ruutu kuvastaa 30 x 30 m:n kokoista aluetta.

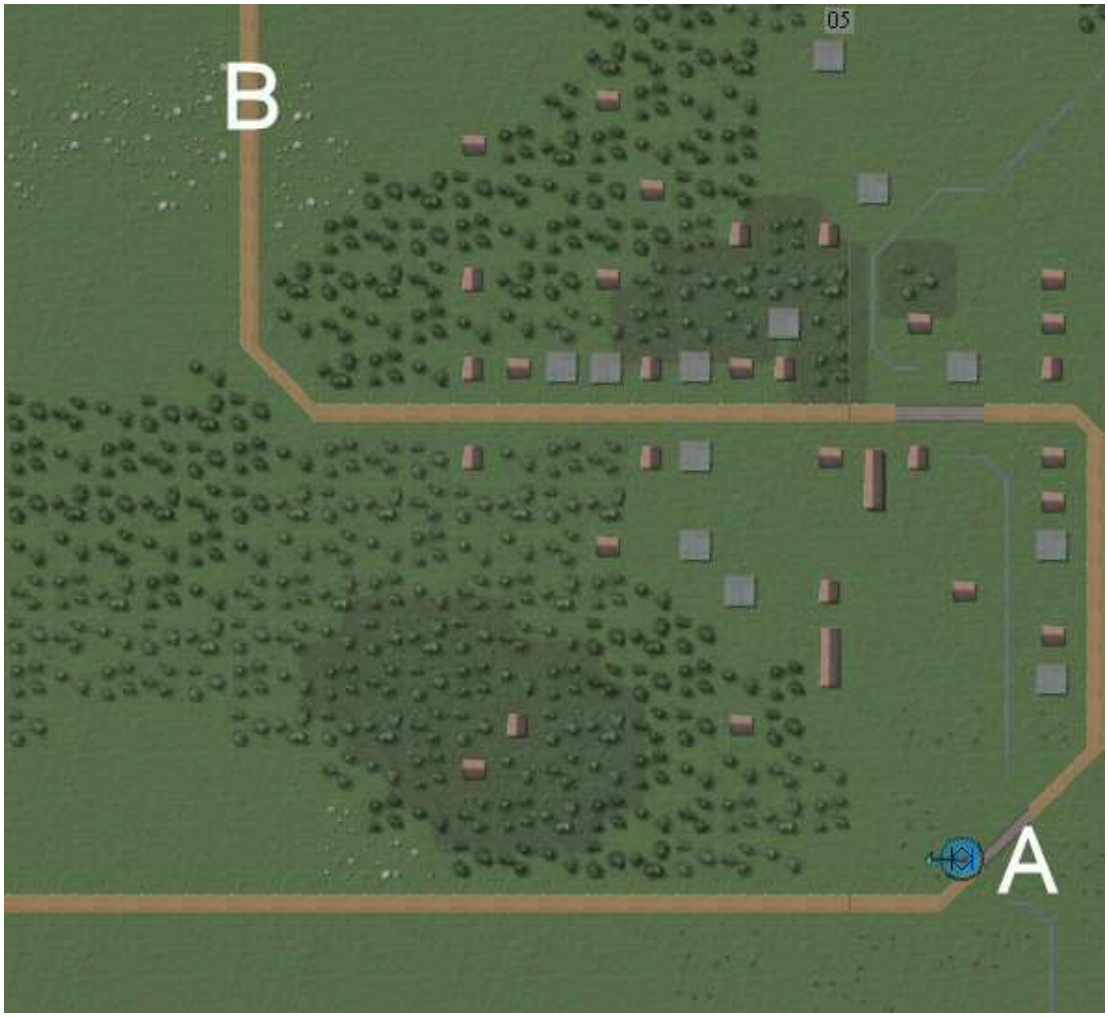
Yksikkö (unit) voi olla esimerkiksi yksittäinen ajoneuvo tai 2–10 hengen ryhmä. Maayksiköt ovat linkitettyinä joko joukkueeseen tai komppaniaan, mikä mahdollistaa liikkeen synkronoinnin ja muodostelmassa liikkumisen. Komppanian voi esimerkiksi asettaa liikkumaan tietä pitkin, jolloin hitaimman yksikön nopeus asettaa muiden yksiköiden nopeuden. Yksiköiden ja muodostelmien ohjaaminen tapahtuu asettamalla reittipisteitä (waypoint), joiden väliltä yksiköt etsivät reitin sen mukaan, millaisia kriteerejä reitin suhteen on aikaisemmin annettu. Pääpiirteissään reittejä on kolmea tyyppiä: lyhyin (quickest), nopein (fastest) ja suojaisin (covered). Etäisyyden lisäksi tärkeimmät maaston ominaisuudet tässä tapauksessa ovat kuljettavuus (trafficability) ja suoja (cover). Jotkin maastotyypit saattavat estää liikkumisen kokonaan, esimerkiksi ajoneuvot eivät pysty liikkumaan solmuun, jossa on rakennus. Siltojen reunoilla oleviin solmuihin ei pysty liikkumaan, jotta siltoja ei voi ylittää sivusuunnassa.

Reitinhakualgoritmina toimii A*. Esiteltävässä versiossa (0.496) yksiköt käyttävät liikkumisessa kiintopisteinä solujen keskikohtia, joten liikeradat eivät ole täysin sulavia. Lisäksi käytössä on Manhattan-heuristiikka, minkä takia yksikkö pyrkii hakeutumaan välittömästi samalle pysty- ja vaakatasolle kuin kohde. Reitinhaku ei toimi omassa prosessissaan, mikä saattaa pahimmassa tapauksessa aiheuttaa sovelluksen hetkellisen ”jäätymisen”. Tämä tapahtuu yleensä vain, jos suuri määrä yksiköitä hakee samanaikaisesti reittiä toisella puolella karttaa olevaan pisteeseen, käytännössä kuitenkin hyvin harvoin.

Molemmilla osapuolilla eli pelaajalla ja vastustajalla on käytössä omat taulukot kartan vapaista solmuista. Ajoneuvojen ja jalkaväen liikkumiseen on myös erilliset taulut, mikä mahdollistaa omanlaisen estekartan tekemisen molemmalle yksikkö-

tyypille. Estekartan pohjana toimii staattinen maasto, joka on alusta lähtien molempien tiedossa. Simulaation ajon aikana estekarttoja päivitetään, kun saadaan näköhavaintoja vastustajan liikkeistä.

Seuraavissa esimerkeissä ajoneuvo hakee reitin kohteesta ”A” kohteeseen ”B” käyttäen kolmea erilaista hakua (KUVIO 22).



KUVIO 22. Esimerkin aloitustilanne

7.1 Shortest path

"Shortest" eli lyhyimmän reitin haku etsii suoraviivaisimman reitin (KUVIO 23). Nykyisessä versiossa yksiköt pyrkivät liikkumaan kaikkien solmujen keskipisteiden

kautta, ja käytössä on Manhattan-heuristiikka. Tämän vuoksi reitti ei käytä "linnuntietä", mikä olisikin mahdotonta nykyisellä pelimekaniikalla.

KUVIO 23. Shortest path

7.2 Quickest path

"Quickest"-haku pyrkii hakeutumaan maastoon jolla, on pienin kustannusarvo. Kuviosta 24 havaitaan, että vaikka kohteeseen voisi kulkea koko matkan tietä (ruskea alue) pitkin, yksikkö päättää ottaa suoremman aloituksen. Loppumatka kuljetaan kuitenkin tiellä.

KUVIO 24. Quickest path

7.3 Covered path

“Covered”-haku pyrkii käyttämään maastoa, jossa on suuri Cover-arvo (KUVIO 25). Koska tässä tapauksessa kyseessä on ajoneuvo, se ei pysty käyttämään rakennuksia hyväkseen. Se lähtee seuraamaan metsän laitaa ja lopussa jopa valitsee kivikon (harmaat täplät) tien sijaan.

KUVIO 25. Covered path

8 YHTEENVETO

Työn tarkoituksena oli perehdyttää lukija reitinhakuun ja erityisesti A*-algoritmiin. Reitinhaun toteuttaminen on yleensä haastavaa, sillä vaatimukset vaihtelevat hyvin paljon sovelluskohtaisesti. Työssä pyrittiin antamaan perustieto oman reitinhaun toteuttamista varten.

Käytännön sovelluksena esiteltiin A*-algoritmia ja Manhattan-heuristiikkaa käyttävä simulaatio. Algoritmin parissa työskentely osoitti, että valmiina tarjolla olevasta materiaalista ja koodista huolimatta toteuttaminen voi olla hyvinkin haastavaa. Heuristiikan valinnalla on hyvin suuri merkitys algoritmin toimintaan. Joskus algoritmin tarkkuudesta voidaan tinkiä, jotta suorituskykyä saadaan paremmaksi. Tämä onnistuu heuristiikkaa säätämällä. Tehokkuus on tärkeää esimerkiksi peleissä, koska pelattavuuden säilyttämiseksi ohjelman suorituksen on oltava täysin sulavaa.

Perinteisten lajittelumenetelmien käyttö on melko tehotonta suurilla kartoilla. Tätä ajatellen esiteltiin binäärikeko, joka on tehokas keino tiedon varastointiin ja lajitteluun. Suurilla kartoilla tosin ongelmana on binäärikeon kasvaminen ja muistin kulutus.

LÄHTEET

Algers, S. & Bernauer, E. & Boero, M. 1997. Review of Micro-Simulations Models. Www-dokumentti. Saatavissa: http://www.its.leeds.ac.uk/projects/smartest/deliv3.html. Luettu 10.8.2009.

Dijkstra, E. 1959. A Note on Two Problems in Connexion with Graphs. Www-dokumentti. Saatavissa: http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf. Luettu 10.8.2009.

Drakos, N. 1993. Www-dokumentti. Saatavissa: http://www.macs.hw.ac.uk/~alison/ai3notes/tableofcontents2_1.html, http://www.macs.hw.ac.uk/~alison/ai3notes/subsubsection2_6_2_3_2.html. Luettu 10.8.2009.

Ferguson, D. & Gordon, G. & Likhachev, M. 2002. Anytime Dynamic A*: An Anytime, Replanning Algorithm. 1–3. Www-dokumentti. Saatavissa: http://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf. Luettu 10.8.2009.

Fern, A. & Xu, Y. 2007. On Learning Linear Ranking Functions for Beam Search. 1. Www-dokumentti. Saatavissa: http://www.machinelearning.org/proceedings/icml2007/papers/168.pdf. Luettu 10.8.2009.

Graham, R. & McCabe, H. & Sheridan, S. Neural Networks for Real-time Pathfinding in Computer Games. 1–2. Www-dokumentti. Saatavissa: http://www.gamesitb.com/nnpathgraham.pdf. Luettu 10.8.2009.

Koenig, S. & Likhachev, M. 2002. Fast Replanning for Navigation in Unknown Terrain. 1–2. Www-dokumentti. Saatavissa: http://www.cs.cmu.edu/~maxim/docs/dlite_tro05.pdf. Luettu 10.8.2009.

Lester, P. 2003. Using Binary Heaps in A* Pathfinding. Www-dokumentti. Saatavissa: http://www.policyalmanac.org/games/binaryHeaps.htm. Luettu 10.8.2009.

Lester, P. 2005. A* Pathfinding for Beginners. Www-dokumentti. Saatavissa: http://www.policyalmanac.org/games/aStarTutorial.htm. Luettu 10.9.2009.

Loerch, U. 2000. An Introduction to Graph Algorithms. Www-dokumentti. Saatavissa: http://www.cs.auckland.ac.nz/~ute/220ft/graphalg/graphalg.html. Luettu 10.8.2009.

Marczyk, A. 2004. Genetic Algorithms and Evolutionary Computation. Www-dokumentti. Saatavissa: http://www.talkorigins.org/faqs/genalg/genalg.html. Luettu 10.8.2009.

Matthews, J. & Pinter, M. & Scutt, T. 2002. AI Game Programming Wisdom. Charles River Media Inc, 105 – 107, 186, 202, 211.

Patel, A. 2009. Www-dokumentti. Saatavissa: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html, http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html. Luettu 10.8.2009.

Pramudya, P. 2008. Heuristically Informed Search Methods for Solving Path-finding. 2. Www-dokumentti. Saatavissa: http://www.informatika.org/~rinaldi/Stmik/2007-2008/Makalah2008/MakalahIF2251-2008-022.pdf. Luettu 10.8.2009.

Reynolds, C. 1999. Steering Behaviors For Autonomous Characters. Www-dokumentti. Saatavissa: http://www.red3d.com/cwr/steer/gdc99/. Luettu 10.8.2009.

Sherrod, A. 2007. Data Structures and Algorithms for Game Developers. Course Technology, 93, 98, 101, 255, 265.

Smith, P. 2004. Applied data structures with C++. Jones & Bartlett Publishers, 418.

Tyni, T. & Ylinen, J. 1999. Geneettiset algoritmit ohjausjärjestelmissä. 1. Www-dokumentti. Saatavissa: http://www.pcuf.fi/sytyke/lehti/kirj/st19992/06.pdf. Luettu 10.8.2009.