Joonas Pollari

Reitinhakualgoritmien vertailu videopeliympäristöissä

Tietotekniikan kandidaatintutkielma

6. toukokuuta 2020

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Joonas Pollari

Yhteystiedot: joonas.pollari@gmail.com

Ohjaaja: Antti-Jussi Lakanen

Työn nimi: Reitinhakualgoritmien vertailu videopeliympäristöissä

Title in English: Comparison of pathfinding algorithms in video games environments

Työ: Kandidaatintutkielma

Sivumäärä: 19+0

Tiivistelmä: Reitinhaku on prosessi, jossa etsitään reittiä maaliin erilaisissa ympäristöissä. Tässsä tutkielmassa vertaillaan keskenään erilaisia reitinhakualgoritmeja, ja arvioidaan niiden käytettävyyttä videopeliympäristöissä. Algoritmien kompastuskiviä pyritään hahmottamaan ensisijaisesti tarkastelemalla algoritmien toimintaa avoimissa ympäristöissä. Tarkasteltavista algoritmeista A*-algoritmi osoittautuu selvästi muita algoritmeja ylivertaisemmaksi ja käytetymmäksi. Tutkielman havaintoja hyödyntämällä kyetään tekemään parempia ratkaisuja käytettävien reitinhakualgoritmien valinnassa. Jatkotutkimuksen kannalta todetaan A*-algoritmin optimoinnin tarjoavan hyvin mahdollisuuksia.

Avainsanat: reitinhaku, solmuverkko, syvyyshaku, leveyshaku, Dijkstran algoritmi, A*

Abstract: Pathfinding is the process of finding a route to a desired destination in different environments. This dissertation compares different pathfinding algorithms and evaluates their usability in video game environments. The stumbling blocks of algorithms are sought to be perceived primarily by looking at the operation of algorithms in open environments. Of the algorithms examined, the A * algorithm proves to be clearly superior and more used than other algorithms. By utilizing the findings of the dissertation, it is possible to make better decisions in the choice of pathfinding algorithms to be used. Considering further research, it is stated that the optimization of the A * algorithm offers great opportunities.

Keywords: pathfinding, mesh network, depth-first search, breadth-first search, Dijkstra's algorithm, A*

Kuviot

Sisältö

1 Johdanto

Reitinhakualgoritmeja käytetään useissa pelissä. Tässä tutkimuksessa vertaillaan eri reitinhakualgoritmeja sovellettuna tyypillisissä pelitilanteissa. Reitinhaku on prosessi, jossa agentti etsii reitin haluttuun pisteeseen. Tässä tutkimuksessa keskitytään vain sellaisiin tilanteisiin, joissa halutaan löytää yksinkertaisesti reitti yhdestä pisteestä toiseen ilman määrättyjä välietappeja. Tilanteita, joissa reitin on kuljettava useamman pisteen läpi, ei käsitellä lainkaan.

Reitinhakualgoritmeja on useita, ja ne poikkeavat toisistaan merkittävästi. Toiset algoritmit navigoivat tiensä lähtöpisteestä kauimmaiseen solmuun, ja haravoivat sieltä tietään takautuvasti solmuverkon läpi. Toiset taas kartoittavat solmuverkkoa edeten lähtöpisteestä ulospäin, edeten tasaisen sokeasti jokaiseen suuntaan, solmu kerrallaan. Lisäksi on olemassa heuristisia algoritmeja, jotka kykenevät tekemään arvoita esimerkiksi etsittävän pisteen etäisyydestä, ja niiden avulla parantaa reitinhakuprosessia.

Eroja algoritmien väliltä löytyy myös reitin mittaamisessa. Yksinkertaisimmat algoritmit mittaavat reittiä vain solmujen määrässä, mutta toiset algoritmit kykenevät huomioimaan lyhimmän reitin etsinnässä myös solmujen väliset etäisyydet.

Reitinhakuongelmien vaatimukset voivat myös poiketa toisistaan. Joissakin tilanteissa on ehdottoman tärkeää löytää kaikista lyhin mahdollinen reitti tutkittavan alueen halki, mutta toisinaan on myös tärkeää löytää käyttökelpoinen reitti pienessä ajassa, jotta tietokoneen laskenta-aikaa jää muillekkin prosesseille.

Videopelit asettavat reitinhaulle myös muita tavallisista reitinhakutilanteista poikkeavia haasteita. Esimerkiksi usean pelin ympäristöissä on tiekarttojen reittiverkosta poiketen laakeitakin alueita, joiden halki navigoiminen eroaa merkittävästi rajatussa solmuverkossa navigoimisesta. Toinen enimmäkseen videopeleihin rajautuva poikkeustilanne ovat äärettömät navigointialueet. Useassa pelissä kenttiä luodaan proseduraalisesti, eli peli kykenee generoimaan jopa loputtomasti lisää kenttätilaa pelin edetessä. Tämä mahdollistaa kenttien loputtoman suuren koon. Tällaisissa ympäristöissä jotkin algoritmit eivät kykene löytämään reittiä maaliin laisinkaan.

Erilaisia reitinhakualgoritmeja tutkimalla ja vertailemalla opitaan eri algoritmien suoriutumisesta erilaisissa tilanteissa, ja siten kyetään pelejä tehdessä valitsemaan käytettävät algoritmit paremmin. Huonosti toteutettu reitinhaku voi näkyä peleissä esimerkiksi pelin jäätymisenä intensiivisen reitinhaun aikana, tai pelin agenttien huonoilla reittivalinnoilla.

Erityisesti tarkastellaan algoritmien suoriutumista yksinkertaisessa avoimessa ruudukossa, sillä laajoja esteettömiä ympäristöjä esiintyy useissa eri genrejen peleissä, kuten esimerkiksi Guild Warsissa, Minecraftissa, The Simsissä ja Age of Empiresissä. Lisäksi hyvin labyrinttimaisessa ympäristössä suoriutuvat reitinhakualgoritmit voivat avoimemmassa ympäristössä törmätä ongelmiin, joita ei pelkästään solmulabyrintteja tarkastelemalla välttämättä huomattaisi.

Tämä tutkimus reitinhakualgoritmien vertailusta suoritetaan kirjallisuusanalyysiin muodossa. Erilaisia reitinhakualgoritmeja on tutkittu jo vuosikymmeniä, joten kirjallista materiaalia aiheesta löytyy riittävästi kattavaan vertailuun.

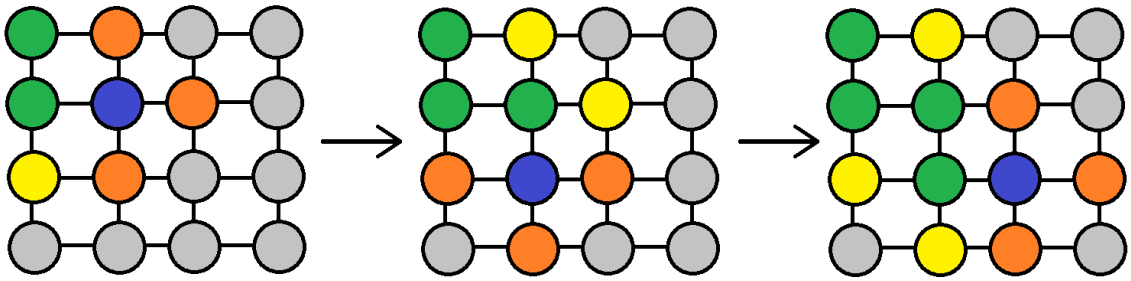
Tutkielman toisessa luvussa avataan aiheeseen liittyviä termejä, ja kolmannessa luvussa käydään läpi erilaisia reitinhakualgoritmeja, ja arvioidaan niiden käyttömahdollisuuksia erimuotoisilla navigointialueilla. Arvioinnissa algoritmien suoriutumista verrataan keskenään sitä mukaan, kun niitä on käsitelty. Lisäksi pyritään hahmottamaan algoritmien suuntaa antavaa käytettävyyttä peleissä.

2 Reitinhakualgoritmeja

Reitinhakuongelman lähtötilanne asetetaan samalla tavalla, riippumatta käytettävästä algoritmista. Ensin alue jaetaan solmuihin. Solmu on navigoitavan alueen piste, jonka läpi voidaan kulkea. Yhdessä kaikki alueen solmut muodostavat solmuverkon, jossa kuljettaessa edetään aina yhdestä solmusta toiseen. Reitinhakualgoritmin tehtävä on etsiä ne solmut, jotka yhdessä muodostavat reitin haluttuun pisteeseen (Lester 2005).

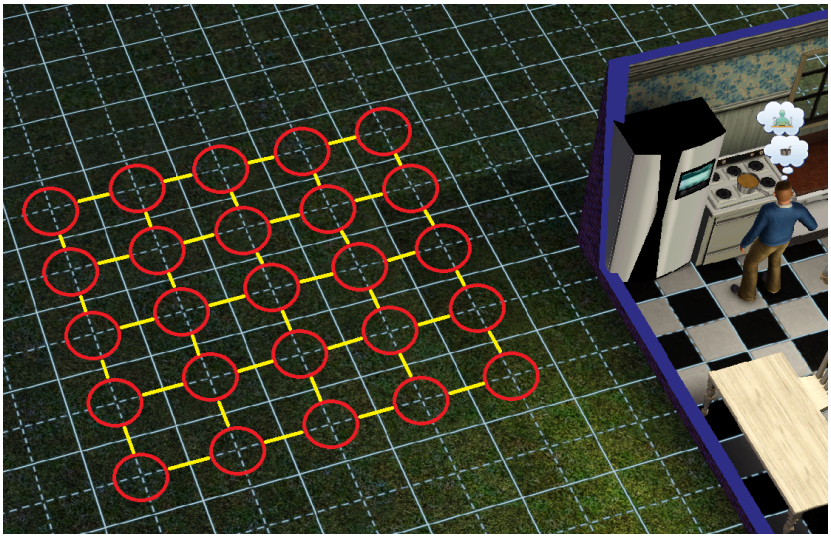
Erilaisia reitinhakualgoritmeja on olemassa useita. Tässä työssä käydään läpi yksinkertaisimmat syvyys-, ja leveyssuuntainen reitinhakualgoritmi, leveyshausta paranneltu Dijkstran algoritmi, sekä A*-algoritmi.

Tässä tutkielmassa tarkasteltavalla solmulla tarkoitetaan sitä solmua, josta reitinhakualgoritmi on jo tallentanut tarvitsemansa tiedot, ja jonka viereisten solmujen tietoja algoritmi on siirtynyt tallentamaan muistiin. Viereisillä solmuilla viitataan puolestaan niihin solmuihin, joihin tarkasteltavasta solmusta pääsee kulkemaan suoraan, kulkematta muiden solmujen läpi. Vapaiksi solmuiksi kutsutaan niitä solmuja, jotka algoritmi on tallentanut muistiin, mutta joita ei ole vielä tarkasteltu. Tämä solmujen luokittelu on visualisoitu kuviossa 1.



Kuvio 1. Tyypillisen reitinhaun solmuja. Kuvissa tarkasteltava solmu merkitty sinisellä, jo tarkastellut solmut vihreällä ja tarkasteltavan solmun viereiset solmut oranssilla. Keltaisilla merkityt solmut on aiemmissa vaiheissa tallennettu muistiin, ja seuraava tarkasteltava solmu valitaan oranssien ja keltaisten solmujen joukosta.

Tutkielmassa esteettömällä ruudukolla viitataan sellaiseen alueeseen, jossa solmut muodostavat keskenään verkkomaisen rakenteen. Esimerkiksi The Sims -pelien avoimet ympäristöt toimivat tällaisena ympäristönä, kun solmut asetetaan jokaiseen pelin ruutuun siten, että viereisillä ruuduilla sijaitsevat solmut ovat aina viereisiä solmuja. Tätä on havainnollistettu Kuviossa 2.



Kuvio 2. Esteetön ruudukko hahmotettuna The Sims 3 -pelissä.

2.1 Syvyyshaku

Navigoitavan alueen syvyyssuuntainen läpikäynti (depth-first search) toimii siten, että solmuverkossa tarkastellaan solmuja, edeten yhteen suuntaan niin kauan, kunnes saavutaan solmuverkon päähän. Tämän jälkeen palataan edelliseen ohitettuun vapaaseen solmuun, josta jatketaan uuteen tutkimattomaan suuntaan, edeten koko ajan ennalta määrätyssä suuntajärjestyksessä. Tätä jatketaan, kunnes solmuverkon etsitty piste on löytynyt (Tarjan 1972). Al-

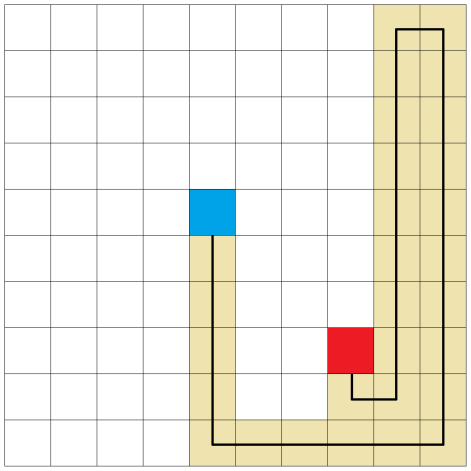
goritmin toimintaa on avattu pseudokoodin muodossa Algoritmissa 1.

```
Suuntajärjestys = [alas, oikealle, ylös, vasemmalle];
while Maalia ei saavutettu do
    if vapaa solmu vieressä then
        etene suuntajärjestyksen priorisoimassa järjestyksessä lähimpään, ennestään
          tarkastelemattomaan solmuun, jos sellainen löytyy;
    else
        Palaa edelliseen solmuun;
    end
end
```

Algoritmi 1: Syvyyshaku pseudokoodina

Labyrintissä tämä tarkoittaisi ennalta valituin käännöksin etenemistä ensimmäiseen umpikujaan asti, josta palattaisiin aina edelliseen risteykseen, ja sieltä suunnistamista jatkettaisiin uuteen suuntaan. Solmulabyrintissä reitti maaliin lopulta löytyisi, mikäli sellainen on olemassa, ja navigoitava alue on äärellinen. Syvyyshaulla löydetty reitti ei silti olisi välttämättä lyhyin mahdollinen.

Syvyyshaun suurimmat heikkoudet tulevat vastaan avoimessa ympäristössä. Mikäli algoritmi ei törmää yhteenkään umpikujaan, se jatkaa solmujen läpikäymistä, kunnes maali löytyy, palaamatta missään vaiheessa takaisin päin. Tämän ongelmallisuutta on havainnollistettu Kuviossa 3, jossa Algoritmin 1 logiikkaa on sovellettu ruudukossa. Kuten kuviosta huomataan, on algoritmin löytämä reitti huomattavasti lyhintä mahdollista reittiä pidempi.



↓ → ↑ ←

Kuvio 3. Syvyyshaku ruudukossa. Lähtöpiste sinisellä ja maali punaisella. Kuvan alla nuolilla esitettynä algoritmin käyttämä etenemissuuntajärjestys.

Syvyyshaun toinen heikkous tulisi vastaan proseduraalisesti loputtomasti generoituvassa kentässä. Tällaisessa solmuverkossa algoritmi tarkastelisi solmuja edeten lähtösuuntaan päin rajattomasti, ja siten maalin löytyminen olisi hyvin epätodennäköistä (Kallem 2012).

2.2 Leveyshaku

Leveyshaku (Breadth-first search) sen sijaan tarkastelee solmuja yksittäin lähtöpisteestä ulospäin siten, että seuraavaksi tarkasteltavaksi solmuksi valitaan aina lähtöpistettä läheisin solmu. Myös leveyshaku noudattaa suuntajärjestystä syvyyssuuntaisen tavoin, mutta tästä poikeneten leveyshaku lisää aina tarkasteltavan solmun viereiset vapaat solmut prioriteettijonon perälle, jonka mukaan solmuverkossa eteneminen tapahtuu (Beamer, Asanović ja Patterson

2013). Leveyshaun toiminta on esitetty Algoritmissä 2 pseudokoodina.

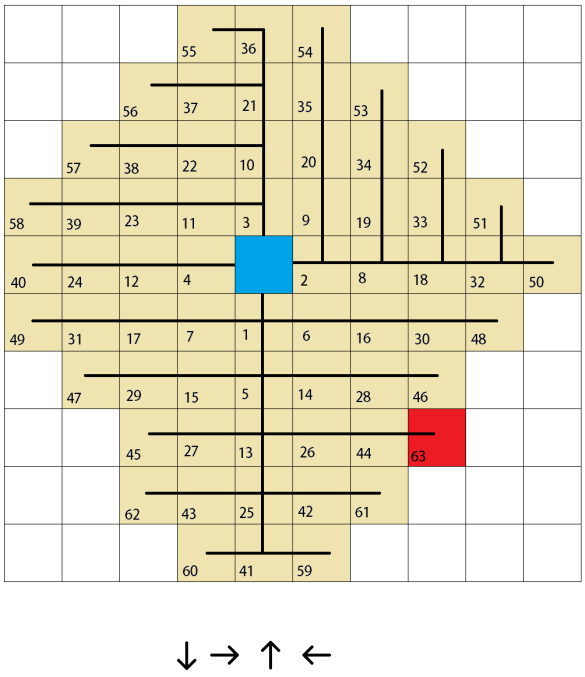
```
Suuntataulukko = [alas, oikealle, ylös, vasemmalle];
Prioriteettijono = [];
while Maalia ei saavutettu do
    if vapaa solmu tarkasteltavan vieressä then
        Lisää vapaat solmut prioriteettijonoon suuntataulukon määräämässä
         järjestyksessä;
    else
        Etene prioriteettijonon mukaisessa järjestyksessä seuraavaan solmuun;
    end
end
```

Algoritmi 2: Leveyshaku pseudokoodina

Näin edeten löydetään aina lyhyin reitti maaliin navigoitavasta alueesta riippumatta (Bundy ja Wallen 1984). Lisäksi loputtomankin suuressa alueessa navigoitaessa ei ole vaaraa, että algoritmi eksyisi etsimään mahdollista reittiä kauas ohi maalista.

Leveyshaun heikkoutena puolestaan on tarkasteltavien solmujen määrä. Kuten syvyyshaulla, ei leveyshaullakaan ole informaatiota maalin suunnasta, ja siten se käy läpi jokaisen navigoitavan alueen solmun, joka sijaitsee maalia lähempänä lähtöpistettä. Tämä heikkous osoittautuu sitä ongelmallisemmaksi, mitä kauempana maali sijaitsee lähtöpisteestä. Tätä havainnollistetaan Kuviossa 4, jossa leveyshakua on sovellettu avoimessa ruudukossa.

Käytännössä tämä ongelma ilmenee tietokoneen muistin kanssa. Koska jokainen tarkasteltu reitti täytyy tallentaa reitinhaun aikana muistiin, kasvaa tallennettavan datan määrä reitinhaun aikana nopeasti. Jos solmuverkko on laaja, ja maali sijaitsee liian kaukana lähtöpistettä, on algoritmin mahdollista kaatua muistin täyttymiseen ennen maalin löytymistä (Korf 1985).



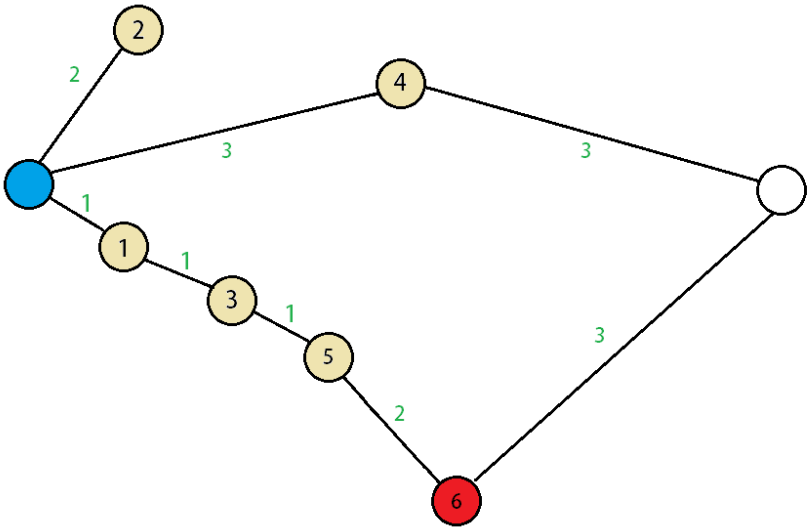
Kuvio 4. Leveyshaku ruudukossa. Ruutujen läpikäyntijärjestys on numeroituna ruutuihin. Löytynyt reitti on lyhyin mahdollinen, mutta etsittävien ruutujen määrä on suuri.

2.3 Dijkstran algoritmi

Kuten leveyshaku, myös Dijkstran algoritmi etsii reitin maaliin lähtöpisteestä sokeasti jokaiseen suuntaan edeten. Erona näillä on Dijkstran algoritmin kyky ottaa huomioon solmujen väliset etäisyydet, pelkän solmujen määrän sijaan. Kuvion 4 tilanteessa, jossa jokaisen solmun välinen etäisyys on yhtä pitkä, Dijkstra toimisi lähes identtisesti leveyshaun kanssa.

Sen sijaan esimerkiksi kartalla kaupungit sijaitsevat yleensä hyvin vaihtelevin etäisyyksin. Tällöin lyhyin reitti kaupungista toiseen ei ole välttämättä se, jonka varrella joudutaan ohittamaan pienin määrä muita kaupunkeja, vaan kaupunkien väliset etäisyydet vaikuttavat merkittävästi parhaimman reitin etsimisessä. Tällaisessa tilanteessa edellinen leveyshaku löytäisi juuri risteysten määrässä lyhimmän reitin, mutta Dijkstran algoritmi kykenisi palauttamaan lyhimmän reitin matkassa mitattuna, kuten kuviossa 5.

Dijkstran alogoritmi toimii siten, että se merkitsee muistiin etäisyydet tarkasteltavan solmun



Kuvio 5. Dijkstran algoritmi osaa etsiä lyhyimmän reitin solmujen väliset etäisyydet huomioiden. Vastaavassa tilanteessa leveyshaku antaisi pienemmän solmumäärän läpikäyvän, mutta matkaltaan pidemmän reitin. Solmujen tarkastelujärjestys numeroituna solmuihin ja solmujen väliset etäisyydet merkitty vihreällä.

naapurisolmuihin, ja etenee aina siihen tutkimattomaan solmuun, johon pääsee lyhintä reittiä lähtöpisteestä. Mikäli tarkasteltavan solmun naapurille on jo tallennettu reitti muistiin, verrataan tätä tarkasteltavan solmun kautta kuljettavaan reittiin, ja parempi reitti jätetään muistiin. Algoritmi laajentaa tutkittua aluetta tasaisesti jokaiseen suuntaan, kunnes löytää lyhyimmän reitin maaliin (Reddy 2013). Dijkstran algoritmin toimintaa on hahmotettu pseudokoodilla

Algoritmissa 3

```
G = solmun etäisyys lähtöpisteestä kuljettuja solmuja pitkin;
while Maalia ei saavutettu do
    if tarkistamaton solmu vieressä then
        Laske viereisten solmujen G;
        Jos viereisen solmun vanha G suurempi, kuin uusi, korvaa vanha uudella;
    else
        Siirry tarkastelemattomaan solmuun, jolla on pienin G;
    end
end
```

Algoritmi 3: Dijksrtan algoritmi pseudokoodina

Koska Dijkstran algoritmi etenee aina lähtöpisteestä lähimpään solmuun, se käy leveyshaun tavoin välttämättä läpi pisteitä täysin riippumatta maalin sijainnista. Siten senkin joutuu, maalin ollessa kaukana lähtöpisteestä, tarkastamaan suuressa ja esteettömässä alueessa navigoitaessa massiivisen määrän solmuja, kuten leveyshakukin.

2.4 A*-algoritmi

Edellisistä algoritmeista poiketen A* on heuristinen reitinhakualgoritmi, joka ilmenee siten, että se kykenee hyödyntämään tietoa maalin sijainnista reitinhakuprosessin aikana. Algoritmi tarkastelee solmuja Dijkstran algoritmin ja leveyshaun tavoin, kartoittaen kaikki tarkasteltavaa solmua ympäröivät solmut, mutta tallentaen niistä enemmän dataa muistiin.

Jokaisesta solmusta tallennetaan lyhin reitti kyseiseen solmuun, sekä heuristinen arvio pisteen ja maalin välisestä etäisyydestä. A* priorisoi etenemisjärjestyksessä ensisijaisesti niitä solmuja, joissa näiden etäisyyksien summa on pienin, ja toissijaisesti niitä solmuja, joiden etäisyys maalista on piemenpi (Reddy 2013). Näin edeten algoritmi tulee lopulta aina löytämään reitin maaliin, mikäli sellainen on olemassa (Hart, Nilsson ja Raphael 1968). Algorit-

min toimintaa on hahmotettu pseudokoodilla Algoritmissa 4.

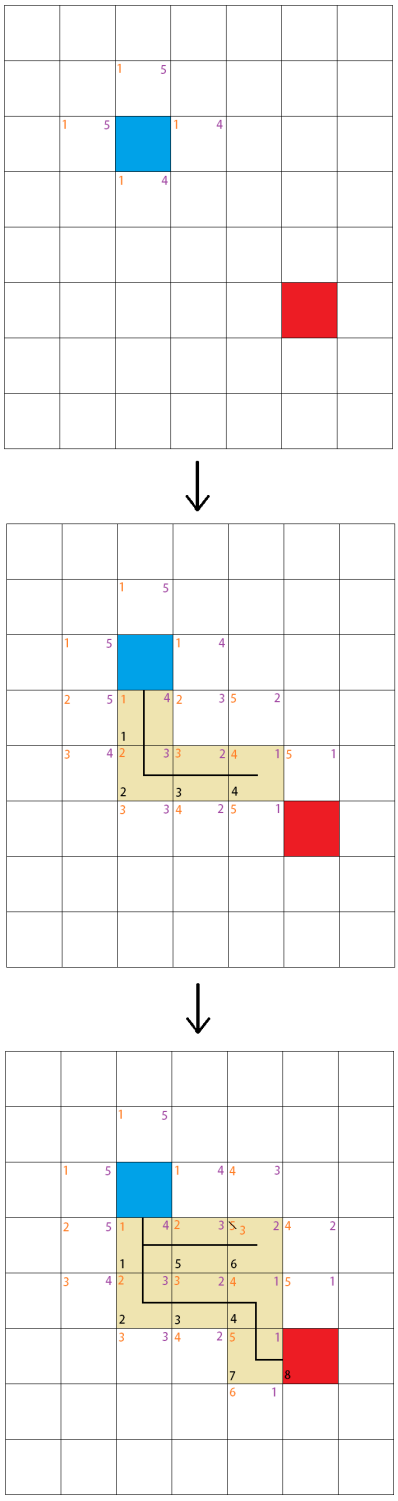
```
G = solmun etäisyys lähtöpisteestä;
H = solmun arvioitu etäisyys maalista;
F = G + H;
while Maalia ei saavutettu do
    if tarkistamaton solmu vieressä then
        Laske viereisten solmujen G, H ja F;
        Jos viereisen solmun vanha G suurempi, kuin uusi, korvaa vanha uudella;
    else
        Siirry solmuun, jolla pienin F, tasatilanteessa valiten näistä solmu, jolla
         pienempi H;
    end
end
```

Algoritmi 4: A*-algoritmi pseudokoodina

A*-algoritmi korjaa leveyshaun, ja Dijkstran algoritmin heikkoutta tarkastella läpi jokainen solmu sattumanvaraisesti hyödyntämällä tietoa maalin sijainnista. Reitin mittaaminen A*-algoritmissa toimii Dijkstran algoritmin tavoin. Etäisyys lähtöpisteestä jokaiseen solmuun mitataan yhä etäisyydellä, pelkän solmujen lukumäärän sijaan, joten myös A* kykenee etsimään matkallisesti lyhintä reittiä alueessa, joissa solmujen väliset etäisyydet poikkeavat toisistaan.

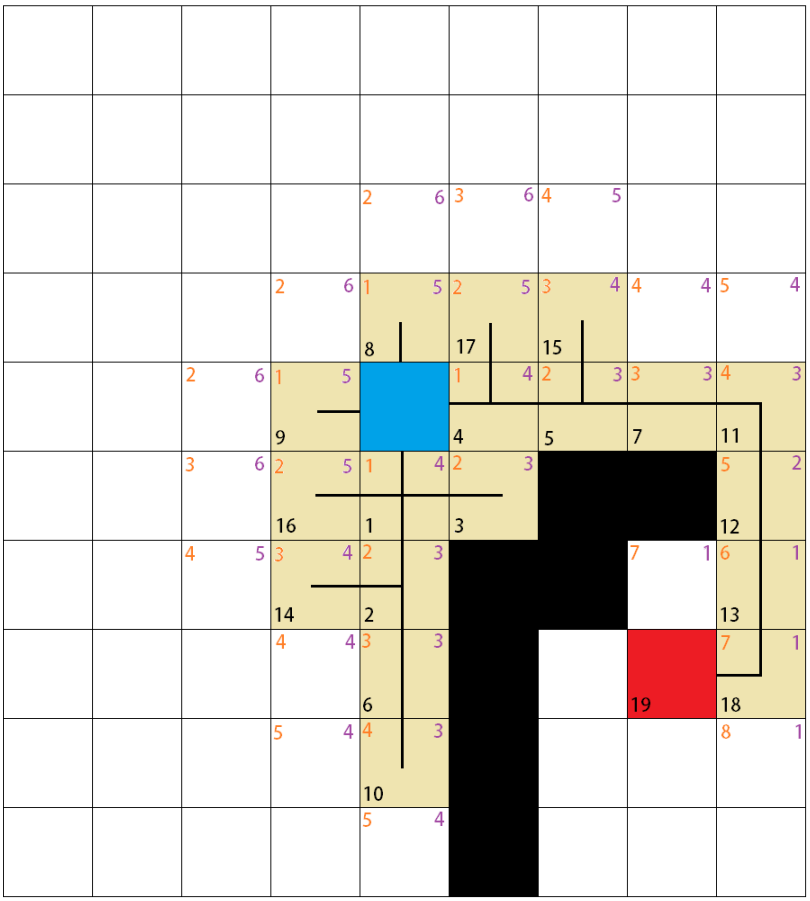
Näiden algoritmien tavoin runsas muistin kulutus on silti merkittävä heikkous A*-algoritmissäkin (Cui ja Shi 2011). Vaikka algoritmi ei tarkastelekaan jokaista mahdollista reittiä maaliin asti, tallentaa se yksittäisistä solmuista enemmän dataa, kuin mikään aiemmista.

Huolimatta tästä, A* on yleisin peleissä käytetty algoritmi (Cui ja Shi 2011). Aiempiin algoritmeihin verrattuna se suoriutuu reitinhausta esteettömissä ruudukossa huomattavasti leveyshakua pienemmällä tarkasteltavien solmujen määrällä, ja löytäen syvyyshakua paremman reitin, kuten voidaan huomata Kuviosta 6, jossa A*-algoritmin reitinhakua käydään läpi vaiheittain.



Kuvio 6. Koska A* kykenee hyödyntämään tietoa maalin sijainnista hakuprosessistaan, se suoritutuu avoimessakin ympäristössä navigoimisesta hyvin. Kuvassa violeteilla numeroilla esitettynä tarkasteltujen ruutujen etäisyys punaisella merkittyyn maaliin linnuntietä pitkin ja oranssilla tarkasteltujen ruutujen etäisyys lähtöruudusta tunnettua reittiä pitkin.

Vaikka tässä tutkielmassa vertaillaan pitkälti esteettömiä ruudukoita, täytyy reitinhakualgoritmien kyetä navigoimaan myös vaikeammassakin ympäristössä. Koska leveyshaku ja Dijkstran algoritmi tarkastelevat solmuja sokaeasti jokaiseen suuntaan, tuottavat ympäristön esteet näille hyvin vähän ongelmia. Myös A* kykenee aina löytämään optimaalisen reitin maaliin, jos sen heuristinen funktio on konsistentti (Russell ja Norvig 2010). A*-algoritmin reitinhakua esteen ympäri on hahmotettu kuviossa 7.



Kuvio 7. Konsistentin heuristisen funktion avulla A* löytää optimaalisen reitin esteidenkin ympäri. Kuvassa violeteilla numeroilla esitettynä tarkasteltujen ruutujen etäisyys punaisella merkittyyn maaliin linnuntietä pitkin ja oranssilla tarkasteltujen ruutujen etäisyys lähtöruudusta tunnettua reittiä pitkin.

3 Yhteenveto

Reitinhakualgoritmeja on kehitetty jo vuosikymmeniä, ja tämän seurauksena erilaisia algoritmeja, sekä niitä käsitteleviä tutkimuksia, on runsaasti.

A*-algoritmi näyttäisi olevan laajalti vakiintunut ratkaisu reitinhakuongelmiin videopeleissä, ja hyvästä syystä. Tässä tutkimuksessa keskityttiin vertailemaan algoritmien suoriutumista erityisesti esteettömissä ruudukoissa, koska tämän kaltaiset alueet ovat peleissä yleisiä, pelin genreen katsomatta. Tällaisessa ymäristössä A*-algoritmi suoriutui reitinhaussa erityisen hyvin muihin algoritmeihin verrattuna.

Tässä tutkimuksessa tarkasteltu leveyshaku, sekä Dijkstran algoritmi löytävät aina lopulta lyhimmän reitin haluttuun pisteeseen, mutta solmuverkon muodosta riippuen nämä algoritmit saattavat joutua tarkastelemaan reitinhaun aikana maalin etäisyyteen nähden eksponentiaalisesti kasvavan määrän solmuja.

Myös sillä, kuinka solmut sijaitsevat navigoitavalla alueella, vaikuttaa reitinhakualgoritmin valintaan. Mikäli solmujen väliset etäisyydet vaihtelevat, kykenevät tässä tarkastelluista algoritmeista ainoastaan Dijkstran algoritmi, sekä A*-algoritmi ottamaan tämän huomioon lyhintä reittiä etsiessään.

Tarkastelluista algoritmeista heikoimmin suoriutui syvyyshaku. Mikäli reitinhakua joudutaan suorittamaan esteettömässä ruudukossa, voivat algoritmin löytämät reitit olla pituutensa puolesta merkittävästi pidempiä, kuin muiden algoritmien vastaavissa tilanteissa löytämät.

Vaikka A* ei oletettavasti löydä aina lyhintä mahdollista reittiä, niin sen suuren käyttöasteen perusteella voidaan olettaa, että sen löytämät reitit ovat hyvin usein käyttökelpoisia. Uudemmat reitinhakualgoritmeihin liittyvät tutkimukset näyttäisivät keskittyvän A*-algoritmin optimointiin ja muihin heuristisiin algoritmeihin. Tämänkin tutkielman perusteella tämä vaikuttaisi järkevältä kohteelta lisätutkimukselle.

Lähteet

Beamer, Scott, Krste Asanović ja David Patterson. 2013. “Direction-optimizing breadth-first search”. *Scientific Programming* 21 (3-4): 137–148.

Bundy, Alan, ja Lincoln Wallen. 1984. “Breadth-first search”. Teoksessa *Catalogue of artificial intelligence tools,* 13–13. Springer.

Cui, Xiao, ja Hao Shi. 2011. “A*-based pathfinding in modern computer games”. *International Journal of Computer Science and Network Security* 11 (1): 125–130.

Hart, Peter E, Nils J Nilsson ja Bertram Raphael. 1968. “A formal basis for the heuristic determination of minimum cost paths”. *IEEE transactions on Systems Science and Cybernetics* 4 (2): 100–107.

Kallem, Sreekanth Reddy. 2012. “Artificial Intelligence Algorithms”. *IOSR Journal Of Computer* 6 (3): 1–8.

Korf, Richard E. 1985. “Depth-first iterative-deepening: An optimal admissible tree search”. *Artificial intelligence* 27 (1): 97–109.

Lester, Patrick. 2005. “A* pathfinding for beginners”. *online]. GameDev WebSite. http://www. gamedev. net/reference/articles/article2003. asp (Acesso em 08/02/2009).*

Reddy, Harika. 2013. “PATH FINDING-Dijkstra’s and A* Algorithm’s”. *International Journal in IT and Engineering:* 1–15.

Russell, Stuart J, ja Peter Norvig. 2010. *Artificial intelligence: a modern approach.*

Tarjan, Robert. 1972. “Depth-first search and linear graph algorithms”. *SIAM journal on computing* 1 (2): 146–160.