Tampereen ammattikorkeakoulu

# Reitinetsintä Unity-moottorissa

Niko Matikainen

OPINNÄYTETYÖ
Marraskuu 2021

Tietojenkäsittely
Game Production

**TIIVISTELMÄ**

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Game Production

MATIKAINEN, NIKO
Reitinetsintä Unity-moottorissa

Opinnäytetyö 20 sivua, joista liitteitä 0 sivua
Marraskuu 2021

---

Tämän opinnäytetyöprojektin tavoitteena oli tehdä demonstraatiosovellus, jolla pystyisi esittelemään reitinetsinnän toimintaa visuaalisesti. Sovelluksessa pitäisi pystyä vaikuttamaan reitinetsinnän käyttäytymiseen ja näkemään miten erilaiset maastot ja heuristiikat esimerkiksi vaikuttavat reitinetsintään.

Työssä haluttiin käyttää Unity-moottoria, koska se on erittäin yleinen työkalu, jonka toiminnasta on paljon dokumentaatiota, sekä sillä pystyy tekemään helposti sovelluksia monille eri käyttöjärjestelmille. Työn alussa todettiin, että Unity-moottorissa ei ole sisäänrakennettuna minkäänlaista reitinetsintää, joka tarkoitti sitä, että se pitää rakentaa itse tai ostaa kolmannelta osapuolelta.

Opinnäytetyön tuloksena oli Unity-sovellus, joka rakentaa satunnaista maastoa erilaisilla asetuksilla, sekä visuaalisesti esittelee askel askeleelta reitinetsinnän prosessin. Lisäksi sovelluksessa oli monia asetuksia, kuten käytettävä heuristiikka ja liikkumistyyppi.

---

Asiasanat: reitinetsintä, Unity, Dijkstran algoritmi, A*-algoritmi

## ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Game Production

MATIKAINEN, NIKO
Pathfinding in the Unity Engine

Bachelor's thesis 20 pages, appendices 0 pages
November 2021

---

The goal of this thesis was to create a demo application about pathfinding, which could be used to visually demonstrate how it works. It should be possible to affect the behaviour of the pathfinding algorithm in the application and see how different terrains and heuristics affect it.

The aim was to use the Unity engine for this work due to how common it is, and how much documentation it has, but also since it is very easy to create applications for multiple operating systems with it. In the beginning of the work, we noticed that the Unity engine had no built-in pathfinding systems, which meant that such had to be either built by oneself or bought from a third party.

The result of the thesis project was a Unity application, which can build random terrain with different settings, and visually demonstrate how the pathfinding algorithm works step-by-step. In addition, the application had several settings, like the used heuristic and movement type.

---

Key words: pathfinding, Unity, Dijkstra’s algorithm, A*-algorithm

**SISÄLLYS**

## LYHENTEET JA TERMIT

| | |
|---|---|
| 3D | Kolmiulotteinen |
| 2D | Kaksiulotteinen |
| A* | ”A-star”-reitinetsintäalgoritmi |
| Heuristiikka | Arvio tai veikkaus |
| Syntaksi | Koodin tai kielen rakenne ja säännöt |
| Node | Datarakenne, joka kuvastaa pistettä kartalla |

# 1 JOHDANTO

Tämän opinnäytetyön tarkoituksena oli tehdä toimiva demonstraatio-ohjelma reitinetsinnästä erilaisissa maastoissa. Työn pääasiallinen reitinetsintäalgoritmi on A*, mutta esittelee myös Dijkstran algoritmia, jota A* on jatkanut heuristiikoilla.

Opinnäytetyö käyttää Unity-moottoria sen yleisyyden takia, sekä implementaation helpottamisen vuoksi. Unity on myös erittäin tehokas 2D-implementaatioihin, vaikka se onkin tunnetumpi sillä tehdyistä 3D-peleistä.

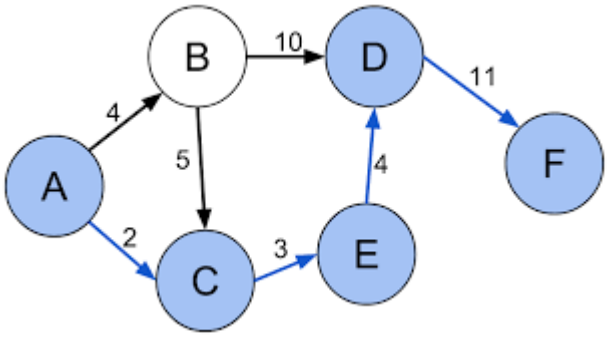
Työn päätarkoitus on esitellä toteutusta. Tarkoituksena on demonstroida reitinetsintäalgoritmien käyttäytymistä ja sen lisäksi näyttää tapoja vaikuttaa niiden toimintaan.

Tässä opinnäytetyössä esitellään A*-algoritmi ja sen implementaatio 2D-maastossa. Tämän jälkeen käsitellään toteutuksen toimintaa. Lopuksi pohditaan erilaisia parannuksia sekä muutoksia, joilla voidaan vaikuttaa reitinetsintään.

# 2 REITINETSINTÄ

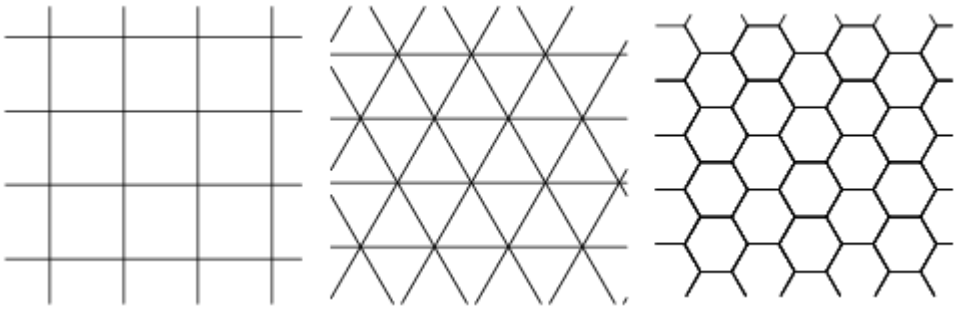
## 2.1 Mitä se on?

Reitinetsinnällä tarkoitetaan tässä tapauksessa algoritmia, joka etsii optimaalisia reittejä ruudukoissa tai pisteiden välillä. Videopeleissä yleisin tapa hyödyntää reitinetsintää on ruudukoissa, kun taas niiden ulkopuolella käytetään yleisesti karttamalleja, joissa on yhdistettyjä pisteitä. Pisteiden välillä on reittejä tai teitä, joiden pituus tai kulkemisen hinta on määritetty jollain käyttötarkoitukseen sopivalla tavalla, ks. kuva 1.




KUVA 1. Pisteitä, joiden välillä on reittejä

On myös olemassa erilaisia ruudukoita, joissa reitinetsintä algoritmeja voidaan hyödyntää. Näissä käytetään muuten samaa periaatetta kuin pisteiden välillä etsimisessä, paitsi että reittien pituus on aina sama, ja jokaisella ruudulla on reitti jokaiseen naapuriinsa. Tähän tarkoitukseen on monia erilaisia ruudukoita, ks. kuva 2. (Abd Algfoor 2015, 1.)



KUVA 2. Neli-, kolmi-, ja kuusikulmioruudukot

Tämän lisäksi voidaan myös etsiä reittejä pintojen päällä niin sanotulla ”navigation mesh” -tyylillä, joka rakentaa pisteistä vapaan pinnan 2D- tai 3D-ympäristössä. Tässä tavassa ei käytetä suoria pisteitä navigoimiseen, vaan määritetään pisteillä alueen rajat, jossa navigoida. Tämä tyyli on tosin paljon monimutkaisempi, vaatii täysin erilaisen algoritmin, ja ei sisälly tämän työn aiheeseen.

Kumpikaan algoritmi, jota tässä työssä esitellään ei toimi suoraan esimerkiksi pelimaailmassa olevan geometrian kanssa, joten siitä tehdään yksinkertaistettu approksimaatio eli graafi, jota sitten käytetään reitinetsintään (Millington 2019, 196).

## 2.2 Mihin sitä käytetään?

Reitinetsinnälle on monia erilaisia käyttötarkoituksia, sekä videopeleissä että niiden ulkopuolella. Sitä voidaan käyttää esimerkiksi karttaohjelmissa lyhyimmän ajomatkan löytämiseen kaupunkien välillä.

Peleissä yleisin käyttötarkoitus on pelaaja- tai tietokonehahmon liikkuminen peliympäristössä, mitä erittäin suuri osa peleistä tarvitsee. Tämän lisäksi myös esimerkiksi kaupunginrakennuspeleissä sitä tarvitaan teiden rakentamiseen sekä autojen navigoimiseen tieverkostossa.

# 3 REITINETSINNÄN ALGORITMIT

## 3.1 Eri algoritmien erot

Erilaisia reitinetsinnän algoritmeja on monia, mutta niistä yksinkertaisimmat ovat breadth-first- ja depth-first-algoritmit. Yksinkertaisuudessaan, breadth-first etsii ensin samalla tasolla tai lähimpänä olevat pisteet, kun taas depth-first-haku etsii yhden syvimmän reitin loppuun asti ja jatkaa kauimmaisista ensimmäisenä.

Näiden lisäksi on toki muitakin algoritmeja, mutta ne kaksi, joihin keskitytään tässä työssä ovat Dijkstran algoritmi, ja sitä pohjana käyttävä laajennettu variantti, A*.

## 3.2 Dijkstran algoritmi

Dijkstran algoritmi on graafipohjainen reitinetsintäalgoritmi, jonka perusideana on käyttää aloituspistettä, ja avointa settiä pisteitä, jossa on alussa vain aloituspisteen naapurit.

Jokaisen iteraation alussa etsitään avoimesta setistä lähin piste, merkitään se suljetuksi, ja lisätään avoimeen settiin pisteen naapurit, joita ei ole vielä suljettu. Tämä yksinkertainen prosessi toistuu, kunnes algoritmi löytää loppupisteen, ja samalla myös lyhyimmän reitin. Koska lähimmät pisteet käydään läpi ensimmäisenä, on lopputuloksena myös aina nopein reitti loppupisteeseen.

Tämän käyttäytymisen voi myös kuvitella niin, että algoritmi leviäisi ruudukossa tasaisesti kaikkiin suuntiin, ja aina piirtäisi nuolen kynällä siihen suuntaan, josta se on tullut. Loppupisteen löytämisen jälkeen, lyhin reitti saadaan selville helposti vain seuraamalla nuolia taaksepäin. (Millington 2019, 204.)

Tämä algoritmi toimii myös täysin sokkona, eli meidän ei tarvitse tietää missä suunnassa loppupiste on, vaan algoritmi etsii pisteitä, kunnes ne loppuvat kesken tai loppupiste on löydetty. Mutta tästä syystä Dijkstran algoritmi voi olla myös

joissain tapauksissa erittäin hidas, koska se etsii täysin väärään suuntaan meneviä reittejä. (Millington 2019, 203, 214.)

### 3.3 A* algoritmi

Toisin kuin Dijkstran algoritmi, A* käyttää haussa apuna heuristiikkaa. Heuristiikka on nyrkkisääntö, eli karkea arvio etäisyydestä loppupisteeseen, joka saattaa toimia isossa osassa tapauksista, mutta joskus ei toimi ollenkaan (Millington 2019, 26). Tämä tarkoittaa sitä, että A* on erittäin paljon tehokkaampi löytämään reittejä silloin kun loppupisteen sijainti on tiedossa, tai osataan jollain tavalla arvioida etäisyyttä loppupisteeseen.

A*-algoritmin perusperiaate on sama kuin Dijkstran algoritmilla, eli alussa algoritmilla on aloituspiste ja avoin setti pisteitä, mutta se miten A* eroaa, on sen tapa päätellä, mitä pistettä katsotaan seuraavaksi. Dijkstran algoritmi katsoo aina seuraavaksi vain lähintä pistettä, mutta A* katsoo sitä pistettä, jolla etäisyyden ja heuristiikan summa on pienin. (Sabri 2018, 2.)

Jos heuristiikka on yhtä suuri kuin nolla, A* käyttäytyy täysin samalla tavalla kuin Dijkstran algoritmi. Mutta kun heuristiikan tarkkuutta oikean polun pituuteen verrattuna parannetaan, myös A* algoritmin tehokkuus nousee, koska algoritmi käyttää vähemmän aikaa turhien alueiden läpi käymiseen.

Jos heuristiikan arvo on täysin tarkasti oikea optimaalinen etäisyys loppupisteeseen, niin A* etsii vähiten pisteitä ja silloin on myös nopein. Tästä huolimatta, on yleisesti erittäin epäkäytännöllistä kirjoittaa täysin tarkka heuristiikka, koska se hidastaisi algoritmin toimintaa liikaa.

# 4 TOTEUTTAMINEN UNITY-YMPÄRISTÖSSÄ

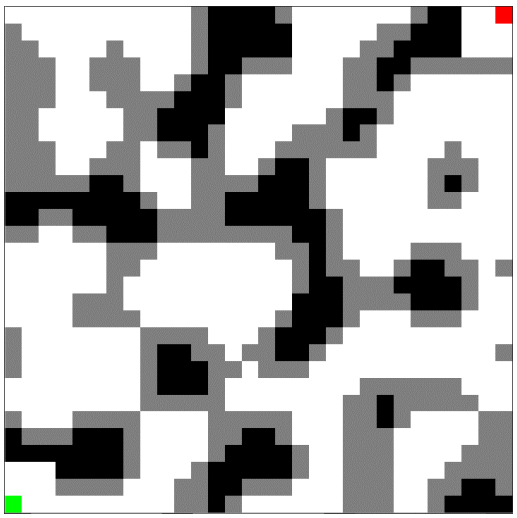
## 4.1 Maaston tuottaminen

Tässä toteutuksessa käytetään neliöruudukkoa, joka koostuu erivärisistä neliöistä.

- Vihreä on aloituspiste
- Punainen on lopetuspiste
- Musta on seinä
- Harmaa on vaikeaa maastoa
- Valkoinen on normaali vapaa ruutu

Harmaa, eli vaikea maasto, tarkoittaa ruutua, johonka on kalliimpaa liikkua. Tämän voi saavuttaa helposti lisäämällä ylimääräistä hintaa liikkumiseen, samalla kuin lasketaan liikkeen hinta naapurista kyseiseen ruutuun.

Yksinkertaisin tapa tehdä maastoa on 2D ”perlin noise” kohina-algoritmilla. Tämä algoritmi antaa käyttöömme sulavasti muuttuvia satunnaisia lukuja nollan ja yhden väliltä. Näitä voidaan käyttää indikoimaan erilaisia ruututyyppejä, kuten asettaa kaikki arvoja 0,7–1,0 vastaavat ruudut seiniksi, arvoja 0,4–0,7 vastaavat ruudut vaikeaksi maastoksi, ja kaiken muun vapaiksi ruuduiksi, ks. kuva 3.



KUVA 3. 2D ”perlin noise” algoritmilla tuotettua maastoa

Maaston tuottamiseen on monia muitakin tapoja, kuten käsin piirtäminen tai erilaiset sokkelon tuottamisen algoritmit. Käyttäjän piirtämä kartta on yleisin tapa demonstroida reitinetsintää, sillä se on erittäin helppo implementoida, sekä se antaa käyttäjälle mahdollisuuden tutkia algoritmin käyttäytymistä erilaisissa erikoistapauksissa.

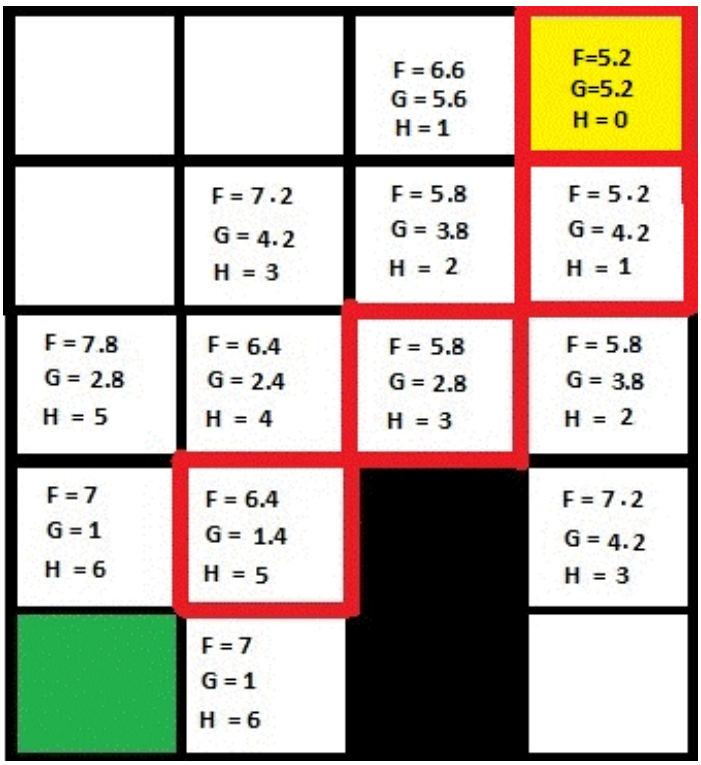
### 4.2 A*-algoritmin implementointi

Kun lähdetään implementoimaan A*-algoritmia, niin voidaan käyttää apuna niin sanottua “pseudokoodia”. Pseudokoodi kuvastaa oikean ohjelmointikielen rakennetta, mutta se on tehty helpommin ihmisen luettavaksi, ja sen tarkoituksena on vain kuvastaa mahdollisimman selkeästi algoritmin toimintaa olematta kuitenkaan syntaksiltaan rajoittava. Ei ole olemassa yhtä ainutta oikeaa tapaa kirjoittaa pseudokoodia, koska sen kirjoitustyyli vaihtelee ohjelmoijien välillä.

Algoritmin implementoimisessa lähdetään yleisesti alkuun ”node”-luokasta, joka säilyttää monenlaista dataa algoritmin toimintaa varten. Yksi node sisältää yleisesti seuraavat arvot:

- Noden sijainnin kartalla.
- H-arvon, eli heuristiikan.
- G-arvon, joka on lyhyimmän reitin pituus aloituspisteeseen.
- F-arvon, joka on G-arvon ja heuristiikan summa.
- “Parent”-viitteen nodeen josta siihen on päädytty

Näitä arvoja käyttämällä A*-algoritmi hakee lyhyimmän reitin alkupisteestä loppupisteeseen. Ks. kuva 4.




KUVA 4. Esimerkki eri nodejen arvoista. Punainen on lyhyin reitti.

Yksinkertaisuudessaan, algoritmi laskee aina kaikille mahdollisille naapureille niiden arvot, jonka jälkeen se valitsee noden jolla on pienin F-arvo. Tämä toistetaan, kunnes viimeiseksi avoimesta setistä poistettu node on loppupiste.
Kun loppupiste on löydetty, voidaan lyhyin reitti löytää helposti seuraamalla taaksepäin loppupisteestä nodejen ”parent”-viitteitä. Tällä tavalla löydetään aina lyhyin reitti alkupisteestä loppupisteeseen. Algoritmin toimintaa voidaan kuvastaa myös pseudokoodina, ks. kuva 5.

1. P = aloituspiste-node
2. Asetetaan F-, G-, ja H-arvot nodelle P
3. Lisätään P avoimeen listaan. P on ainut node avoimessa listassa.
4. B = paras node avoimessa listassa. (Node jolla on pienin F-arvo.)
    a. Jos B on loppupiste, lopetetaan. Reitti on löydetty.
    b. Jos avoin lista on tyhjä, lopetetaan. Reittiä ei voi löytää.
5. C = validi node joka on yhteydessä B nodeen.
    a. Asetetaan C noden F-, G-, ja H-arvot.
    b. Tarkistetaan, löytyykö C node avoimesta tai suljetusta listasta.
        i. Jos on, tarkistetaan uusi reitti lyhyempi. (pienempi F-arvo)
        ii. Jos ei ole, lisätään C avoimeen listaan.
    c. Toistetaan askel 5 kaikille B nodeen yhdistetyille nodeille.
6. Toistetaan askeleesta 4.

KUVA 5. Yksinkertaistettu pseudokoodi A*-algoritmin toiminnasta.

### 4.3 Heuristiikat

Heuristiikoita on monenlaisia, mutta tässä kappaleessa kerrotaan neliöruudukossa toimivista vaihtoehdoista. Nämä ovat erilaisia tapoja mitata etäisyyksiä ruudukossa, ja se mitä heuristiikkaa halutaan käyttää, riippuu yleisesti siitä, miten ruudukossa liikutaan (Rabin 2002, 108).

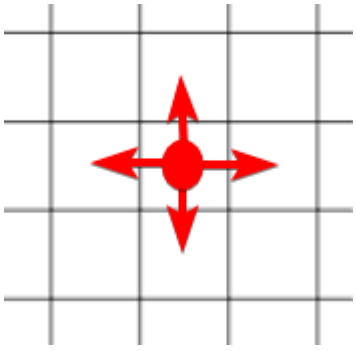
#### 4.3.1 Manhattan-etäisyys

Manhattan-metodi on tapa mitata etäisyyttä, jossa lasketaan yhteen kokonaismäärä liikuttuja ruutuja pysty- ja vaakasuunnassa, ks. kuva 6 (Rabin 2002, 108).

$$h = |x_{start} - x_{destination}| + |y_{start} - y_{destination}|$$

KUVA 6. Manhattan-etäisyyden laskeminen.

Tätä heuristiikkaa käytetään yleensä silloin, jos ruudukossa ei voida liikkua diagonaalisesti ollenkaan, ks. kuva 7. Tällöin heuristiikka kuvaa hyvin kuljettua matkaa, koska se laskee minimimäärän liikkeitä, joita tarvitaan sillä liikkumistyylillä.



KUVA 7. Liikkuminen Manhattan-etäisyyden kanssa.

### 4.3.2 Diagonaalinen etäisyys

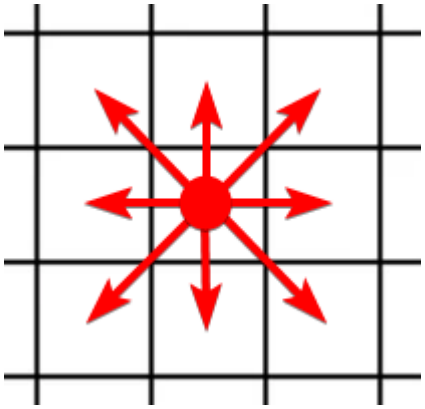
Manhattan-etäisyyden edistyneempi versio on diagonaalinen etäisyys, joka ottaa huomioon mahdolliset diagonaaliset liikkeet. Tässä esimerkissä oletetaan että pysty- ja vaakasuunnassa olevat liikkeet ovat pituudeltaan yhden yksikön pituisia, ja diagonaaliset liikkeet ovat √2 yksikön pituisia, ks. kuva 8.

$$min(x, y) = (x + y - |x - y|)/2$$
$$dx = |x_{start} - x_{destination}|$$
$$dy = |y_{start} - y_{destination}|$$

$$(dx + dy) + (\sqrt{2} - 2) * min(dx, dy)$$

KUVA 8. Diagonaalisen etäisyyden laskeminen.

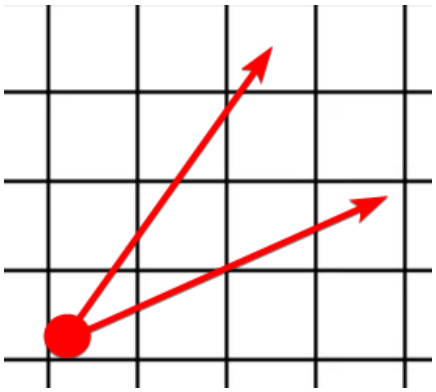
Tämä heuristiikka on hyvä silloin, jos ruudukossa voidaan liikkua pysty- ja vaakasuuntien lisäksi myös diagonaalisesti, ks. kuva 9. Diagonaalinen etäisyys on aina yhtä pitkä tai lyhyempi kuin Manhattan-etäisyys.



KUVA 9. Liikkuminen diagonaalisen etäisyyden kanssa.

### 4.3.3 Euklidinen etäisyys

Euklidista etäisyyttä käytetään enimmäkseen silloin, jos ruudukossa voidaan liikkua missä tahansa kulmassa, ks. kuva 10. Tätä voisi myös kutsua suoraviivaiseksi etäisyydeksi, koska se on aina täysin suora viiva nodesta loppupisteeseen, riippumatta ruudukon rajoituksista (Patel 2021). Euklidinen etäisyys on laskettu Pythagoraan lauseen mukaan, ks. kuva 11.



KUVA 10. Liikkuminen euklidisen etäisyyden kanssa.

$$h = \sqrt{(x_{start} - x_{destination})^2 + (y_{start} - y_{destination})^2}$$

KUVA 11. Euklidisen etäisyyden laskeminen.

A*-algoritmin kanssa tämän etäisyyden käyttäminen on tosin ongelmallista, koska tällöin G- ja H-arvot eivät täsmää. Koska euklidinen etäisyys on lähes aina lyhyempi kuin Manhattan tai diagonaalinen etäisyys, tulos on silti lyhin reitti loppupisteeseen, mutta A* algoritmilla kestää pidempään laskea koko reitti (Millington 2019, 231).

### 4.3.4 Monta loppupistettä

Joskus on tarpeellista käyttää montaa loppupistettä, kuten esimerkiksi jos etsitään lyhyintä reittiä ulos talosta, jossa on monta ulko-ovea. Näissä tapauksissa tarvitsee yleensä vain muuttaa heuristiikan laskentaa ottamaan huomioon kaikki loppupisteet. (Patel 2021)

Heuristiikka itsessään lasketaan täysin samalla tavalla, mutta tässä tapauksessa se lasketaan erikseen jokaiseen loppupisteeseen, ja nodeen kirjataan H-arvoksi pelkästään niistä matalin. (Patel 2021)

### 4.4 Tasatulosten ratkaiseminen

Joissain kartoissa on mahdollista olla monta eri reittiä, jotka ovat saman pituisia, esimerkiksi tasaisilla alueilla ilman seiniä tai vaihtelua. Tässä tapauksessa, A* yleisesti käy läpi kaikki mahdolliset reitit, joilla on sama F-arvo, eikä pelkästään yhtä. (Patel 2021)

Yksinkertaisin tapa ratkaista tämä ongelma on nostaa heuristiikan arvoa erittäin pienellä määrällä, ks. kuva 12. Tämä ei ole paras tapa, ja se tekee heuristiikasta vähemmän hyväksyttävämmän, mutta sen sivuvaikutukset suurimmassa osassa käyttötarkoituksista ovat erittäin minimaaliset.

$$p = (one\ step\ minimum\ cost)/(expected\ maximum\ path\ length)$$
$$heuristic = heuristic * (1 + p)$$

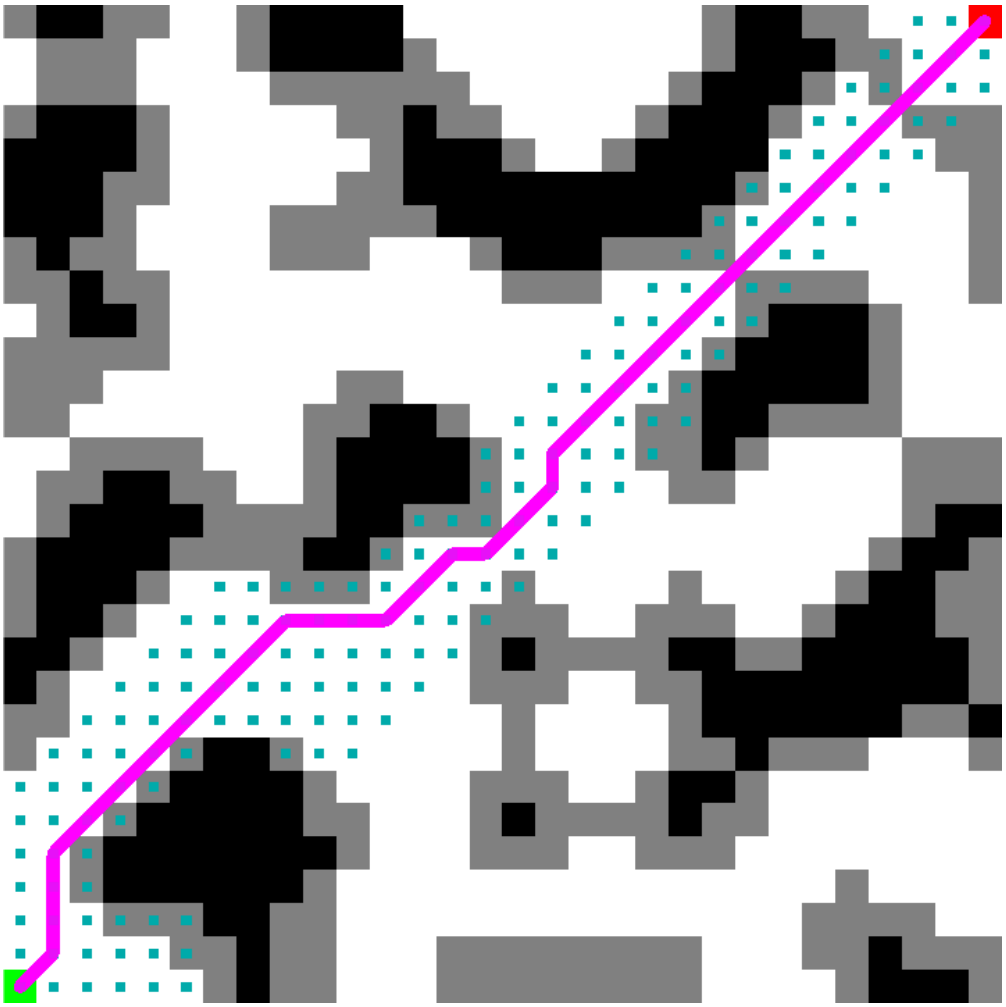
KUVA 12. Esimerkki heuristiikan nostamisesta. (Patel 2021)

Toinen tapa ratkaista ongelma on antaa heuristiikka vertausfunktioon, jolloin jos F-arvot ovat identtiset, niin vertausfunktio valitsee sen noden seuraavaksi, jolla on pienempi H-arvo. (Patel 2021)

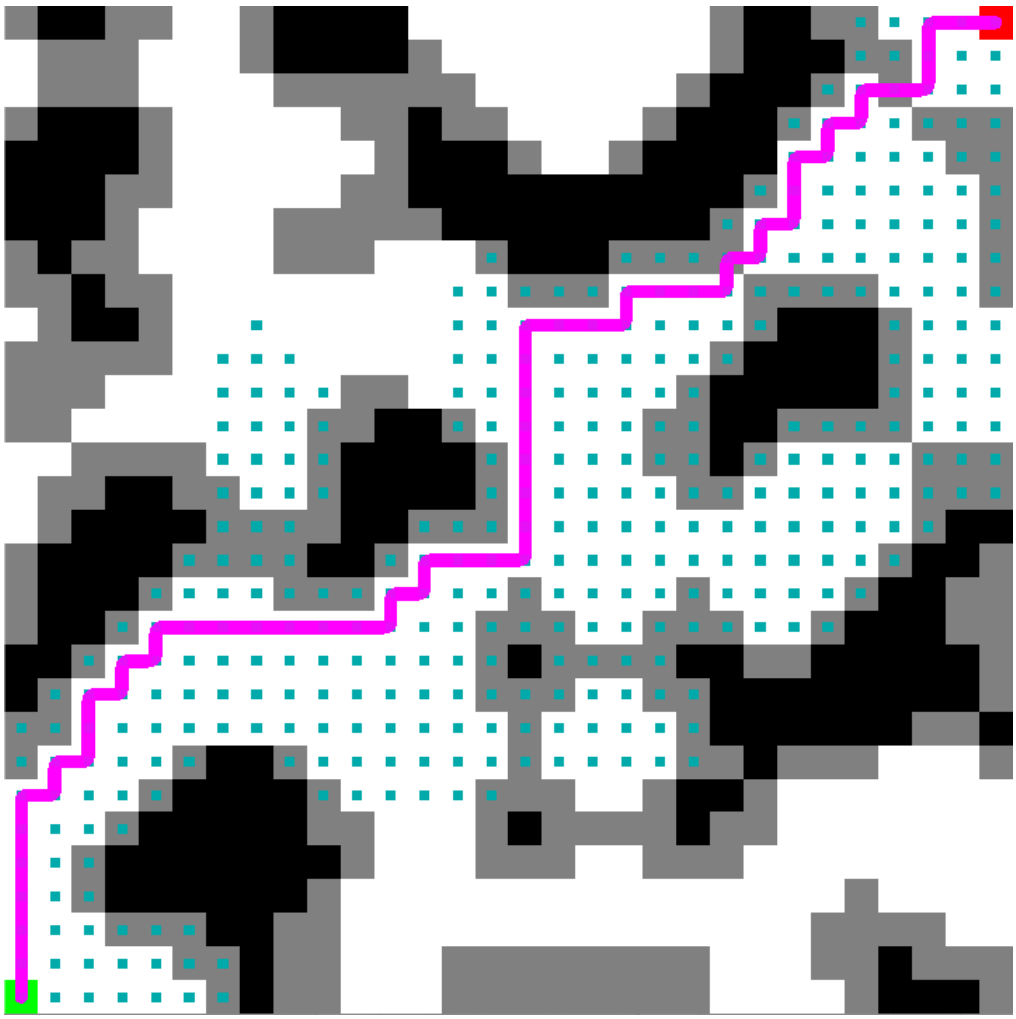
# 5 Unity-toteutuksen esimerkit

Tämän opinnäytetyön Unity-toteutuksessa käyttäjä voi valita monista erilaisista asetuksista, jotka vaikuttavat reitinetsintään. Nämä esimerkit näyttävät, miten eri tavoilla algoritmi lähestyy reitinetsintää, jos se ei esimerkiksi pysty liikkumaan diagonaalisesti ollenkaan, ks. kuvat 13 ja 14.

Kuvissa syaanit pisteet ovat nodeja, jotka algoritmi on tarkistanut, eli suljetun ja avoimen setin sisältö.



KUVA 13. Etsitty reitti diagonaalisella etäisyydellä ja liikkumisella.



KUVA 14. Etsitty reitti Manhattan-etäisyydellä ja -liikkumisella.

# 6 POHDINTA

Opinnäytetyön tekemisen aikana tuli perehdyttyä moniin eri tapoihin ratkaista samoja ongelmia reitinetsinnässä, kuten esimerkiksi tasatulosten ratkonnassa. Lisäksi luin erittäin paljon kirjallisuutta siitä, miten A* toimii ja miten sen toimintaan voi vaikuttaa pienillä ja suurilla muutoksilla.

Myös Unity-moottorin käytön opettelu oli erittäin keskeinen osa toteutusta, koska reitinetsinnän prosessin piti näkyä askel askeleelta käyttäjälle. Tämän lisäksi algoritmin käyttäytymiseen täytyi pystyä vaikuttamaan monilla asetuksilla käyttöliittymästä.

Koin opinnäytetyön erittäin hyvänä motivaattorina ruveta tutkimaan sekä Unity-moottorin omia järjestelmiä, että muita reitinetsintäalgoritmeja kuin vain opinnäytetyössä mainitut A* ja Dijkstran algoritmi. Sovelluksen käyttäjälle annettujen asetusten toteuttamiseen vaadittiin paljon muokkauksia A*-algoritmin normaalisti yksinkertaiseen toimintaan.

Toteutetussa sovelluksessa on vielä tilaa jatkokehitykselle monilla tavoilla, kuten pitkien matkojen optimoinnissa sekä uusien asetusten lisäämisessä. Myös maaston muokkaamista varten voi lisätä työkaluja satunnaisen rakentamisen lisäksi.

**LÄHTEET**

Abd Algfoor, Sunar. “A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games.” International journal of computer games technology 2015 (2015): 1–11. Web.

Millington, Ian. “Pathfinding.” AI for Games. 3rd ed. CRC Press, 2019. 195–295. Web.

Rabin, Steve. *AI Game Programming Wisdom*. Charles River Media, 2002.

Sabri, Radzi. “A Study on Bee Algorithm and A∗ Algorithm for Pathfinding in Games.” 2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE). IEEE, 2018. 224–229. Web.

Patel Amit, Heuristics, luettu 17.11.2021. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html