A*-ALGORITMIN MUOKKAUS LEGOROBOTILLE

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Hämeenlinnan korkeakoulukeskus, Tietojenkäsittelyn koulutusohjelma

Syksy, 2020

Perttu Tähtinen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Hämeenlinnan korkeakoulukeskus

Tekijä Perttu Tähtinen **Vuosi** 2020

Työn nimi A*-algoritmin muokkaus legorobotille

Työn ohjaaja/t Tommi Lahti

TIIVISTELMÄ

Tässä opinnäytetyössä käsitellään reitinhakua ja siinä käytettäviä erilaisia algoritmeja teorian ja käytännön kannalta. Reitinhaku on tekoälyn osa-alue. Opinnäytetyön käytännönosan pohjaksi tutkittiin tekoälyn historiaa ja sen käyttöä nykyään. Teoriaosuudessa käytiin läpi reitinhakua ja tutustuttiin siinä käytettäviin erilaisiin algoritmeihin. Teoriaosan aikana haluttiin selittää tarkemmin kuinka eri algoritmit toimivat ja kuinka ne eroavat toisistaan.

Tämän opinnäytetyön tavoitteena oli jatkokehittää Hämeen ammattikorkeakoulussa ICT-projektin aikana rakennettua legorobottia. Legorobotille haluttiin rakentaa reitinratkaisija, joka selvittää sille annettavan sokkelon käyttäen A*-algoritmia. Opinnäytetyön toimeksiantajana ja ohjaajana toimi Tommi Lahti. Reitinratkaisijalle haluttiin myös asettaa erilaisia rajoitteita sen liikkumiseen. Reitinratkaisijalle oli myös rakennettava tapa, jolla sitä pystyisi visuaalisesti seuraamaan.

Opinnäytetyön lopussa saatiin rakennettua reitinratkaisija, jolle pystyttiin asettamaan erilaisia rajoitteita sen liikkumiseen. Ratkaisijalle saatiin myös tehtyä yksinkertaisia tapoja esittää sen kulkua sokkelon läpi visuaalisesti. Reitinratkaisija ohjelmaa ei opinnäytetyöprosessin aikana liitetty yhteen legorobotin kanssa. Jatkotutkimusehdotuksena esitettiinkin muun muassa, että reitinratkaisija liitetään robottiin ja ratkaisijan koodia voitaisiin tästä vielä parannella.

Avainsanat A*-algoritmi, reitinhaku, tekoäly, hakualgoritmit, Python

Sivut 28 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Program in Business Information Technology
Häme University of Applied Sciences

Author Perttu Tähtinen **Year** 2020

Subject Modification of A*-algorithm for legorobot

Supervisors Tommi Lahti

ABSTRACT

This thesis covers pathfinding and different algorithms used in it from the point of view of theory and practice. Pathfinding is a subset of artificial intelligence. The history of artificial intelligence and its use in modern days were studied as a basis for the practical part of the thesis. The theory part went through pathfinding and different algorithms that are used in pathfinding. During the theory part, the aim was to explain in more detail how the different algorithms work and how they differ from each other.

The goal for this thesis was to further develop the legorobot that was built during Häme Unversity of Applied Sciences ICT-project. Aim was to build a pathfinder for the legorobot, that could solve s maze that is given for it using A*-algorithm. Tommi Lahti was the employer and supervisor for the thesis. It was also desired that you could set restrictions for the pathfinder's movement. The pathfinder also had to be built a way that it could be followed visually. Different ways to visualize the pathfinder's path were presented by drawing the path to the console, to the png-image and drawing using turtle graphics.

At the end of the thesis, the pathfinder was built with the possibility to set different restrictions to its movement. The pathfinder also got simple way to visualize its path through the maze. The pathfinder program was not attached to the legorobot during the thesis. It was suggested that as a further research the pathfinder program could be attached to the legorobot and that the pathfinders code could be improved.

Keywords A*-algorithm, pathfinding, artificial intelligence, pathfinding algorithms, Python

Pages 28 pages

SISÄLLYS

1 JOHDANTO

Reitinhaku on tekoälyn osa-alue, jota käytetään useissa eri tarkoituksissa kuten GPS:ssä, peleissä ja logistiikassa. Reitinhaun yhtenä suurena osana on sen käyttämät erilaiset algoritmit. Algoritmeilla pyritään etsimään lyhintä ja parasta reittiä kahden pisteen väliltä. Vuosien aikana on kehitelty useita erilaisia reitinhaku algoritmeja. Tämän opinnäytetyön aikana käydään läpi muutamia näistä erilaisista reitinhakualgoritmeista.

Hämeen ammattikorkeakoulun tietojenkäsittelyn koulutusohjelman ICT-projektissa rakennettiin legorobotti, joka liikkui sille annetun sokkelon läpi. Robotti liikkui sokkelossa suoraan eteenpäin tai kääntyi oikealle. ICT-projektin aikana rakennettu robotti ei itse ratkaissut sokkeloa vaan sille kerrottiin etukäteen mistä risteyksistä se liikkuu suoraan ja mistä se voi kääntyä oikealle. Robotilla oli yksi kamera, jolla se tunnisti risteyskohdat ja toimi sille annettujen ohjeiden mukaan risteyksissä.

Tämä opinnäytetyö on jatkokehitystä tuolle ICT-projektille. Tarkoitus on rakentaa A*-algoritmia käyttäen ratkaisija, jolla saadaan ratkaistua sille annettavia sokkeloita. Ratkaisijalle pystyy antamaan erilaisia sokkeloita, joista se laskee lyhyimmän reitin kulkea annettuun kohteeseen. Ratkaisija ei kuitenkaan pysty mistä tahansa kuvasta lukemaan sokkelon pohjapiirrosta. Työn aikana on myös selvitettävä, millainen sokkelon pohjapiirroksen kuva on oltava, jotta ratkaisija ymmärtää sen ja pystyy ratkaisemaan sen ilman virheitä. Virheitä voi olla esimerkiksi, ettei ratkaisija tunnista seinää huonon kuvan vuoksi. Tämän lisäksi ratkaisijan on liikuttava samoilla säännöillä kuin ICT-projektissa tehty robotti. Ratkaisija ei siis saa reittiä etsiessään kääntyä vasemmalle vaan sen on löydettävä reitti mitä pystyy kulkemaan vain oikealle kääntymällä.

Opinnäytetyön lopussa voidaan ratkaisija liittää ICT-projektin aikana rakennettuun legorobottiin. Ratkaisijan avulla robotti pystyisi itse kulkemaan sokkelon läpi ilman että sille kerrottaisiin valmis reitti.

Opinnäytetyön tutkimuskysymykset ovat:
- Kuinka rakennetaan toimiva ratkaisija käyttäen A*-algoritmia?
- Kuinka saadaan A*-algoritmille asetettua erilaisia liikkumisen rajoitteita?
- Kuinka saadaan syötettyä ratkaisijalle kuva sokkelosta, jonka se ymmärtää?

2 TEKOÄLY

Tekoäly (artificial intelligence, AI) on tietojenkäsittelytieteen ala, jossa pyritään rakentamaan älykkäitä koneita. Tekoälyn tavoitteena on saada koneet ajattelemaan sekä toimimaan ihmisen kaltaisesti. Tekoälyllä halutaan myös, että koneet pystyvät ratkaisemaan ongelmat ihmistä paremmin ja nopeammin. Tekoälyä on käytetty vuosien aikana monissa eri tarkoituksissa. Tekoälyä pystytään hyödyntämään esimerkiksi teollisuudessa, peleissä, lääketieteessä, liikenteessä ja reitinhaussa. (Built in, 2019; Tuominen & Neittaanmäki, 2019)

2.1 Tekoälyn historiaa

Toisen maailmansodan aikana tietojenkäsittelytieteilijä Alan Turing työsti konetta, joka pystyisi ratkaisemaan Enigma-koodin. Enigma-koodin avulla saksalaiset joukot lähettivät salattuja viestejä. Alan Turing ja hänen tiiminsä rakensivat Bombe-koneen, joka pystyi ratkaisemaan Enigma-koodeja. Turingin rakentamaa konetta pidetään koneoppimisen perustana. (Copeland, 2019) Koneoppiminen on yksi tekoälyn osa-alue. Sana tekoäly nousi kuitenkin ensimmäistä kertaa ilmoille vasta vuonna 1956 kun tietojenkäsittelytieteen professori John McCarthy käytti termiä ”Artificial intelligence” Darthmouth collegen konferenssissa New Hampshiressa. John McCarthya pidetäänkin tekoälyn isänä. (Siukonen & Neittaanmäki, 2019, s. 25) McCarthy julkaisi myös 1960-luvun vaihteessa ohjelmointikielen LISP, josta tuli tärkeä osa koneoppimista (Hemmendinger, 2016).

Tekoälyn ensimmäinen kukoistuskausi oli 1950-luvun lopusta 1970-luvun alkupuolelle. Tänä aikana tietokoneista tuli nopeampia ja helppokäyttöisempiä. Koneoppimisessa käytettävistä algoritmeista tuli parempia ja ihmiset oppivat käyttämään niitä paremmin. Tekoälyn edistyminen sai myös eri maiden hallitusten, kuten esimerkiksi Amerikan yhdysvaltojen, huomion ja sai näin niiltä rahoituksia tekoälyn kehittämiseen. Kone, joka pystyisi kirjoittamaan tekstiä sekä kääntämään puhuttua kieltä, oli suurena kiinnostuksen kohteena. Koneilla haluttiin myös olevan suuri datan prosessointiteho. Tekoälyn kehitys ei kuitenkaan edennyt odotetusti ja suuri osa rahoituksista lopetettiin 1970-luvun puolivälistä 1980-luvun alkuun. Tätä aikaväliä kutsutaan yleisesti ”tekoälytalveksi”. (Anyoha, 2017; Lewis, 2014)

Tekoälyn kehitys heräsi henkiin 1980-luvun alussa, kun tekoälyn kehittäminen sai lisää rahoituksia ja algoritmit kehittyivät. Tänä aikana kehitettiin syväoppimistekniikoita, joilla tietokoneita pystyttiin opettamaan kokemuksien avulla. Edward Feigenbaum esitteli asiantuntijajärjestelmän, joka pystyi matkimaan ihmisen päätöksentekoa. Asiantuntijajärjestelmä on tietokoneohjelma, joka käyttää tekoälymetodeja ratkaistakseen hyvin rajatussa ympäristössä olevia monimutkaisia ongelmia. Näitä järjestelmiä

käytettiin paljon teollisuudessa, jossa ne tuottivat yhtiöille taloudellisia säästöjä. (Anyoha, 2017; Zwass, 2016)

Tekoälyn ala koki toisen tekoälytalven 1980-luvun ja 1990-luvun vaihteessa. 1990-luvun lopulla tekoäly alkoi jälleen saamaan rahoituksia sekä uusia kiinnostuneita katseita. Amerikkalaiset yhtiöt alkoivat kiinnostumaan tekoälystä ja Japani alkoi kehittämään viidennen sukupolven tietokonetta edistääkseen koneoppimista. Vuonna 1997 IBM:n *Deep Blue* shakkia pelaava tietokone voitti hallitsevan shakkimestarin Garry Kasparovin. Tämä voitto oli suuri askel tekoälyn päätöksenteko-ohjelmille. (Anyoha, 2017; Lewis, 2014)

2.2 **Tekoäly nykyään**

Tekoälyn kehitys on mahdollistanut automatisoinnin tehtävissä, joissa aikaisemmin on vaadittu ihmisten ammattiosaamista. Tämän kehityksen avulla monet eri koneet, järjestelmät ja palvelut pystyvät tekemään tehtävänsä tilanteen vaatimalla tavalla. Tekoälyn kehittyessä sen avulla pystytään päivittämään työtehtäviä nykyaikaisemmiksi ja automatisoiduiksi. Koska tekoäly pystyy tekemään päätöksiä sille saatavien tietojen perusteella, pystyy se myös auttamaan ihmisiä työpaikoilla ja arjessa. (Tuominen & Neittaanmäki, 2019)

Terveydenhuollossa tekoälyä käyttämällä saadaan muun muassa kustannussäästöjä. Tekoälyn avulla diagnosoinnista tulee helpompaa lääkäreille ja diagnooseista tulee tarkempia ja nopeampia. Sairaaloissa tuotettua elintoimintojen monitoroimaa dataa pystytään tekoälyn avulla käsitellä paremmin henkilökunnan tueksi. (Neittaanmäki, Ojalainen, Tuominen, Vähäkainu & Äyrämö, 2019)

Terveydenhuollon alalla on tekoälyn ja robotiikan myötä kehitetty erilaisia robottikirurgeja ja sosiaalisia robotteja. Sosiaaliset robotit ovat yleensä ihmisen tai eläimen kaltaisia, isoja tai pieniä koneita. Sosiaaliset robotit pystyvät tekemään erilaisia tehtäviä sekä kommunikoimaan ihmisten kanssa. Nämä robotit pystyisivät tulevaisuudessa esimerkiksi auttamaan kotitöissä. Terveydenhuollossa sosiaaliset robotit pystyvät esimerkiksi pitämään vanhemmille henkilöille seuraa ja muistuttamaan heitä ottamaan tarvittavat lääkkeet. (The Medical Futurist, 2019)

Tekoälyä käytetään myös vahvasti kyberturvallisuuden tukena. Kyberturvallisuudella tarkoitetaan verkossa olevien järjestelmien ja niiden informaation kontrollointia. Tekoälyllä pyritään tuottamaan parempaa tietoturvaa estämään kehittyviä ja taitavampia hyökkääjiä. Tekoälyn avulla pystytään nopeuttamaan ja automatisoimaan prosesseja, joilla tunnistetaan kyberhyökkäyksiä. Hyökkäyksiä vastaan voidaan reagoida nopeammin ja

niiden alkuperää voidaan alkaa selvittämään helpommin. (Neittaanmäki, Ojalainen, Tuominen, Vähäkainu & Äyrämö, 2019)

Rakennusten suunnittelussa sekä rakennusvaiheiden aikana käytetään myös tekoälyä. Valmiissa kiinteistöissä ja teollisuudessa tekoälyä pystytään käyttämään kulujen tarkkailussa ja osassa rakennusten sähkö- ja automaatiojärjestelmissä. Tekoälyä on myös käytössä kouluissa oppimisen tueksi. Opiskelun avuksi on esimerkiksi tekoälyn avulla kehitetty mukautuvia koulutusjärjestelmiä. Mukautuvan koulutusjärjestelmän tarkoituksena on luoda joustava oppimisympäristö opiskelijoille, joilla on eri taustoja, erilaisia kiinnostuksen kohteita, vammoja tai muita ominaisuuksia. Edellä olevien esimerkkien lisäksi tekoälyä käytetään apuna kaupoissa, varastoissa, lakialalla, peleissä, eri bisneksissä sekä monilla muilla aloilla ja palveluissa. (Neittaanmäki, Ojalainen, Tuominen, Vähäkainu & Äyrämö, 2019)

3 REITINHAKU

Reitinhaku on yleisesti tietokoneohjelma, joka hakee lyhintä reittiä kahden pisteen välille. Reitinhaku yhdistetään yleisesti tekoälyn kanssa, koska useimmat reitinhakualgoritmit kehitettiin tekoälytutkijoiden toimesta. Reitinhakuohjelmat ovat tärkeä osa monissa eri tarkoituksissa kuten peleissä, GPS:ssä, logistiikassa ja robotiikassa. Reitinhaku on erittäin käytännöllinen työkalu erilaisten sokkeloiden ratkaisuun. Reitinhaussa käytetään graafeja, joilla kuvataan sokkeloita tai alueita, joista halutaan löytää reitti kahden pisteen väliltä.

Reitinhaussa käytetään erilaisia algoritmeja, joilla pystytään löytämään järjestelmällisesti reitti pisteiden väliltä. Erilaisia reitinhakualgoritmeja on useita, mutta tunnetuimmat ovat Dijkstran algoritmi ja A*. Osa reitinhaku algoritmeista käyttää myös heuristiikkoja, joilla ne pystyvät arvioimaan mistä kannattaa lähteä etsimään lyhintä reittiä. Reitinhaulla pyritään siis suunnittelemaan reittiä jo ennen kuin reitiltä löytyykin jokin este, jota pitää väistää.

Reitinhakualgoritmeja pystytään räätälöimään eri tarkoituksiin. Esimerkiksi GPS-laitteissa voidaan haluta löytää useita eri reittejä haluttuun määränpäähän. Algoritmeja voidaan tällöin muokata käymään kaikki mahdolliset reitit läpi graafilla ja näyttää erilaisia reittejä määränpäähän. Eri reiteistä voidaan myös algoritmien avulla muun muassa laskea mitä reittiä pitkin olisi lyhintä kulkea ja mikä olisi taas ajallisesti nopein reitti.

3.1 Heuristiikka

Heuristinen haku on hakustrategia, jolla yritetään ratkaista ongelmaa heuristisen funktion avulla. Heuristinen haku ei aina takaa parasta ratkaisua ongelmalle, vaan se löytää melko hyvän ratkaisun kohtuullisessa ajassa. Heuristiikka tutkii reitinhakualgoritmeja ja graafin pisteiden haarautuessa eri suuntiin se laskee sille käytettävissä olevat tiedot ja tekee niiden perusteella päätöksen mitä reittiä olisi paras kulkea. (Dataflair, 2018)

Heuristisen haun menetelmät voidaan jakaa kahteen ryhmään, suoriin ja heikkoihin heuristisiin hakutekniikoihin. Suorat hakutekniikat käyvät koko niille annettua aluetta sokkona läpi. Tämän vuoksi ne vaativat usein paljon aika ja muistitilaa. Suoria hakutekniikoita ovat esimerkiksi Leveys ensin- ja Syvyys ensin -hakualgoritmit. (Dataflair, 2018)

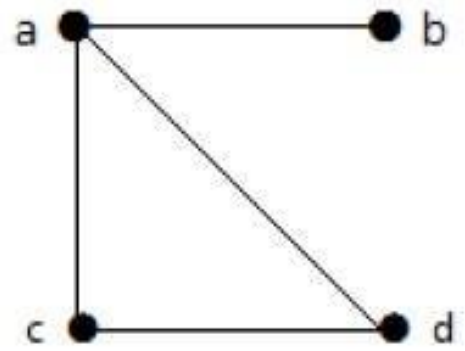
Heikot heuristiset hakutekniikat ovat usein paljon tehokkaampia kuin suorat hakutekniikat. Heikot hakutekniikat tarvitsevat useimmiten aluekohtaista tietoa, joiden avulla ne pystyvät arvioimaan parasta reittiä, jota

kulkea. Paras ensin -haku ja A*-algoritmit kuuluvat heikkoihin hakutekniikoihin. (Dataflair, 2018)

3.2 **Graafi**

Graafit ovat kuvallisia esityksiä esineistä, joissa esineet yhdistetään linkkien avulla. Esineitä graafissa edustavat pisteet. Linkkejä, jotka yhdistävät pisteet, kutsutaan reunoiksi. Graafeja käytetään tietojenkäsittelytieteessä algoritmien tutkimiseen. Graafi on vähintäänkin yksi linkki, joka yhdistää kärkipisteet, mutta yleensä se on ryhmä kärkipisteitä ja reunoja, jotka yhdistävät kärkipisteet. (Tutorialspoint, n.d.)

Graafien peruskokonaisuus koostuu siis muutamista osista kuten pisteistä, viivoista, kärkipisteistä ja reunoista. Piste graafissa on sijainti tai esine yksi-, kaksi- tai kolmiulotteisessa ympäristössä. Pisteitä määritellään yleensä aakkosilla tai numeroilla. Graafissa kärkipiste on periaatteessa sama asia kuin piste. Kärkipisteitä merkitään samoin kuin pisteitä, mutta niissä yhdistyvät useat eri linkit. Viiva eli linkki on yhteys graafissa olevien pisteiden välillä. Linkit merkitään yleensä yhtenäisillä viivoilla, joilla ei ole minkäänlaista määritelmää. Matemaattinen termi *reuna* on nimitys graafissa oleville viivoille, jotka yhdistävät kärkipisteet. Yhdestä kärkipisteestä voidaan luoda graafissa useita reunoja, mutta ilman kärkipistettä ei voida luoda reunaa. Kuvassa 1 nähdään yksinkertainen graafi, jossa on neljä eri pistettä sekä näiden väliset viivat eli linkit. Graafissa on neljä reunaa sekä neljä kärkipistettä. Graafin reunat ovat ab, ac, ad ja cd ja kärkipisteet ovat a, b, c, ja d. (Tutorialspoint, n.d.)



Kuva 1. Yksinkertainen graafi (Tutorialspoint, n.d.).

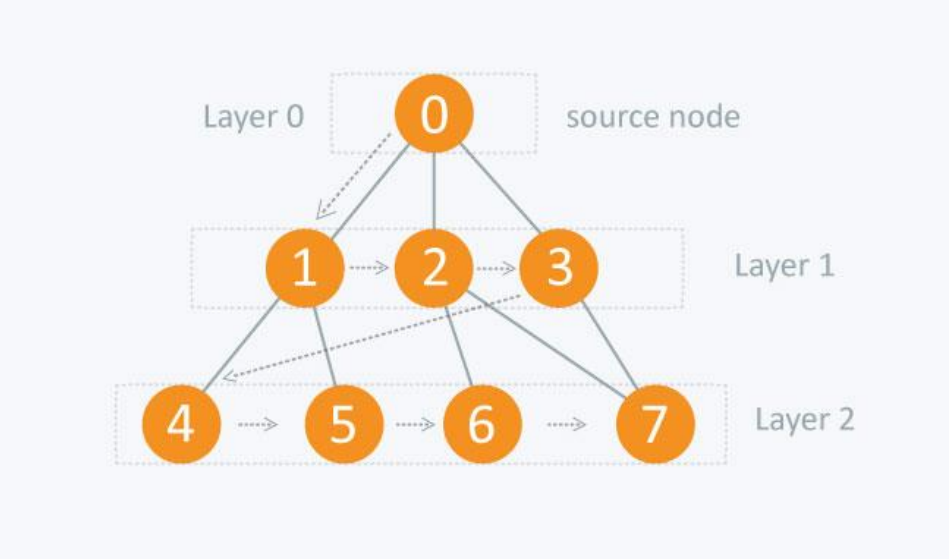
3.3 **Reitinhakualgoritmit**

Reitinhakualgoritmeja on olemassa useita erilaisia, joita pystytään tilanteen vaatiessa käyttämään eri tarkoituksiin. Erilaisia reitinhakualgoritmeja on kehitetty vuosien aikana useita, jotka eroavat toisistaan toimintatavoiltaan. Erilaisia reitinhakualgoritmeja on kehitetty eri käyttötarkoituksia varten sekä päivitetty vanhoja algoritmeja.

Ensimmäiset kehitetyt reitinhakualgoritmit kävivät yleensä läpi koko graafia harkitsematta kuinka paljon pisteiden välisen matkan liikkuminen tulisi maksamaan sille. Uudemmat reitinhakualgoritmit ottavat huomioon pisteiden välisen matkan ja pystyvät näin tuottamaan tarkemman ja nopeamman reitin haluttuun pisteeseen. (Oksa, 2014, s. 4) Seuraavaksi käydään läpi viisi erilaista reitinhakualgoritmia.

3.3.1 Leveys ensin -haku (Breadth-first search)

Leveys ensin -haku (Breadth-first search, BFS) on graafietsintäalgoritmi, joka käy läpi jokaisen pisteen graafin kerroksista ennen siirtymistä seuraavalle kerrokselle. BFS algoritmi etsii siis ensin aloituspisteen ja jatkaa tämän jälkeen kaikkiin sen kerroksen vierekkäisiin pisteisiin. Algoritmi käy leveyssuunnassa graafin yhdeltä kerrokselta kaikissa sen kärkipisteissä piste kerrallaan. Käytyään graafin jokaisessa kärkipisteessä siirtyy algoritmi seuraavaan kerrokseen ja käy tämän kerroksen jokaisessa kärkipisteessä kuten asia on esitetty kuvassa 2. (Hackerearth, n.d.) BFS algoritmi käy koko graafin läpi tällä menetelmällä ja löytää varmasti parhaan ja lyhimmän reitin kulkea, jos sellainen on mahdollista (Elmsley, 2019).

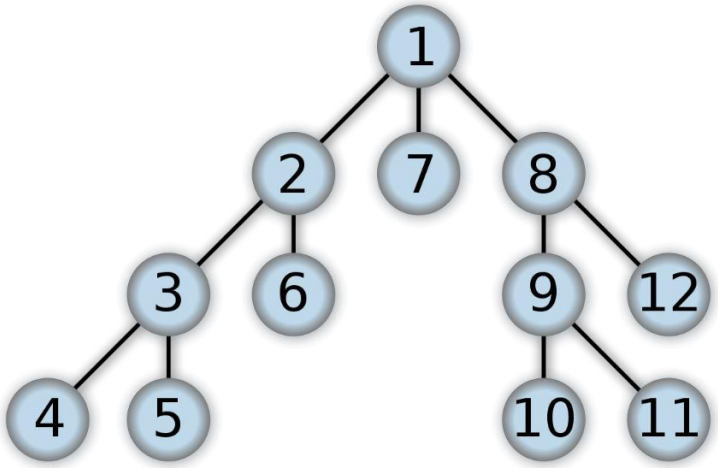


Kuva 2. Leveys ensin -hakualgoritmin liikkuminen graafissa (Hackerearth, n.d.).

BFS algoritmin ongelmana on, että se käy läpi jokaisen graafin kärkipisteen. Jokaisen kärkipisteen läpikäynti vie paljon aikaa. Tämän vuoksi BFS saattaa olla muita algoritmeja hitaampi sekä viedä enemmän muistitilaa kuin muut. BFS pitää muistissaan jokaisen pisteen, jonka se käy läpi ja vie näin paljon tilaa. BFS algoritmia ei kannata käyttää suuriin tehtäviin. Elmsleyn (2019) mukaan tällainen suuri tehtävä voisi olla esimerkiksi rubiikin kuution ratkaisu. (Elmsley, 2019)

3.3.2 Syvyys ensin -haku (Depth-first search)

Syvyys ensin -haku (Depth-first search, DFS) on toistava etsintäalgoritmi, joka ottaa takapakkia etsiessään reittiä graafista. DFS algoritmi liikkuu siis suoraan käyden jokaisen pisteen läpi, kunnes se tulee umpikujaan. Umpikujaan tultaessa se lähtee takaisinpäin kulkien jo käytyjä pisteitä etsien polkua, jossa on piste, jota se ei ole vielä käynyt. Löydettyään polun, jossa algoritmi ei ole vielä käynyt, lähtee se kulkemaan tuota polkua käyden jokaisen pisteen läpi. DFS algoritmin liikkumista sen käydessä läpi graafia on esitetty kuvassa 3. (Hackerearth, n.d.)



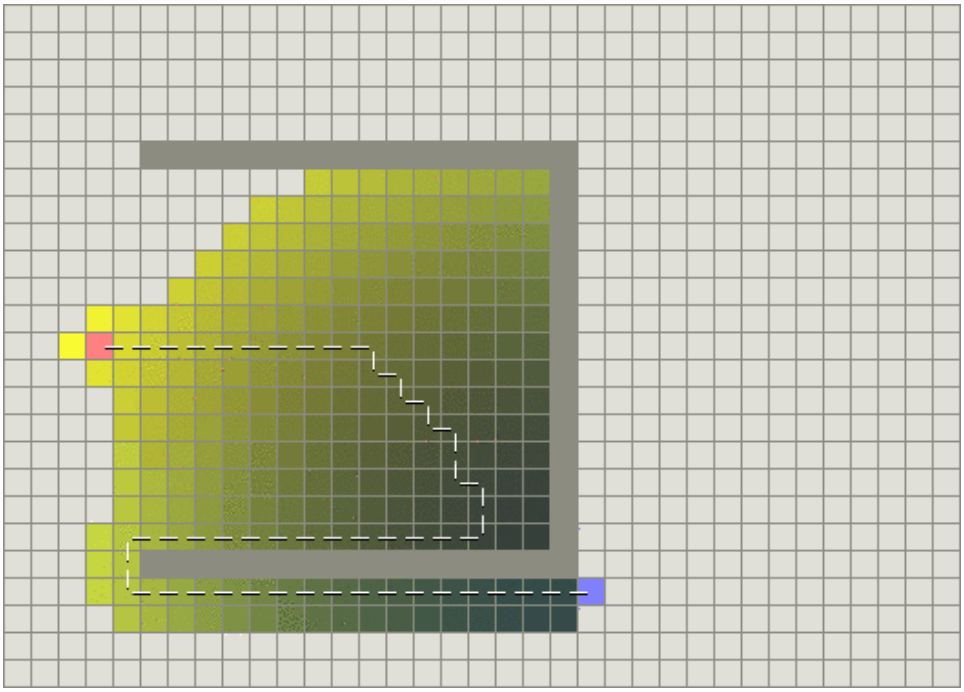
Kuva 3. Syvyys ensin -hakualgoritmin liikkuminen graafissa (Wikipedia, 2020).

DFS algoritmilla on samankaltainen ongelma kuin BFS algoritmilla. Se voi löytää lyhyimmän reitin nopeastikin, jos se valitsee oikeat suunnat, joita se lähtee käymään läpi. Jos DFS algoritmi valitsee suuntansa huonosti, saattaa sillä kestää todella kauan löytää reittinsä. (Oksa, 2014, s. 4) Algoritmin on myös merkittävä jokainen käyty piste käydyksi, jotta se ei lähde käymään samoja pisteitä uudestaan. Jos pisteitä ei merkitä, ja niitä käydään uudestaan läpi, voidaan helposti tulla äärettömän silmukan tilanteeseen. (Hackerearth, n.d.)

3.3.3 Paras ensin -haku (Best-first search)

Molemmissa BFS ja DFS algoritmeissa graafia käytiin läpi melko sokkona. Algoritmit eivät ajatelleet mihin pisteeseen olisi paras liikkua seuraavaksi. Paras ensin -haku (Best-first search) on BFS ja DFS algoritmien yhdistelmistä rakennettu reitinhakualgoritmi. Algoritmi käyttää arviointifunktiota, jonka avulla se päättää mihin pisteeseen olisi parasta liikkua seuraavaksi. Piste, jonka algoritmi valitsee, oletetaan olevan lähimpänä päämäärää. Paras ensin -haku kuuluukin heuristisen haun luokkaan. (GeeksforGeeks, n.d.)

Paras ensin -haku keskittyy siihen suuntaan, jossa sille annettu loppupiste on. Jos algoritmille annettu piste on sen oikealla puolella, keskittyy se ensin vain oikealle liikkumiseen. Paras ensin -haku toimiikin tämän vuoksi erittäin hyvin graafeissa, joissa ei ole esteitä. Jos aloitus- ja loppupisteen välissä kuitenkin on este, ei algoritmi ota sitä huomioon ennen kuin se törmää esteeseen. Tämän jälkeen täytyy algoritmin etsiä uusi reitti, jota se kulkee loppupisteeseen. Kuvassa 4 on vihreällä merkitty alueet, joita algoritmi on tutkinut. Katkoviivalla merkitään reitti, jonka algoritmi on löytänyt loppupisteeseen. Kuvasta nähdään, että algoritmi on aluksi lähtenyt kulkemaan vain siihen suuntaan missä loppupiste on. Esteen tullessa sen eteen on se lähtenyt etsimään uutta reittiä. (Patel, 2019)



Kuva 4. Paras ensin -hakualgoritmin kulku alueessa, jossa on esteitä (Patel, 2019).

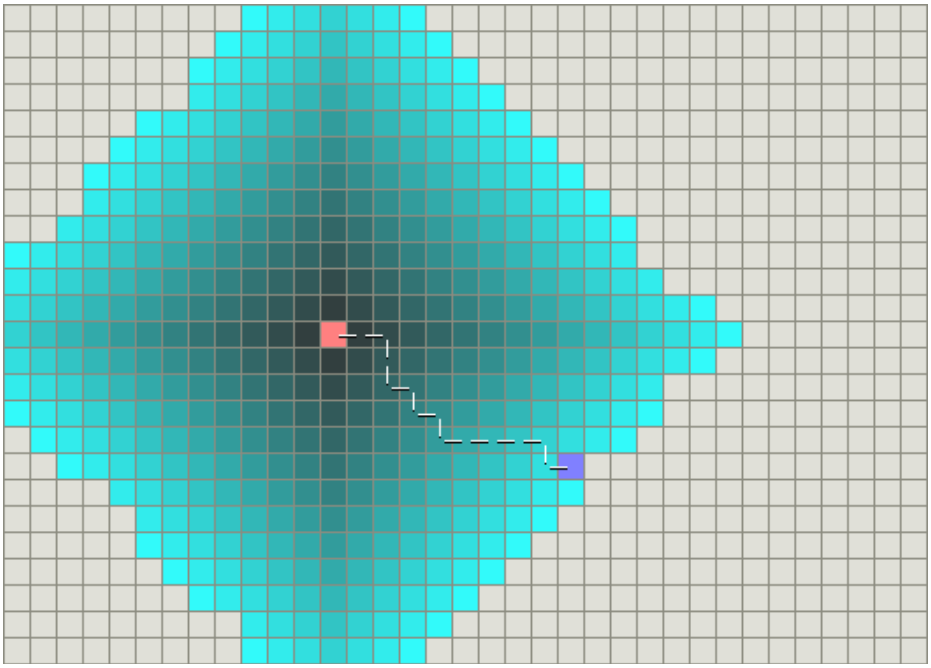
Paras ensin -haku ei tästä syystä välttämättä ole aina paras vaihtoehto reitinhaussa. Koska algoritmi on niin sanotusti ahne, kulkee se aina vain kohti loppupistettä, vaikka tuo reitti ei välttämättä olisikaan paras vaihtoehto.

Algoritmi olettaa miten se pääsisi helpoiten ja nopeinten loppupisteeseen ajattelematta, kuinka pitkän reitin se tekee loppupisteeseen. (Patel, 2019)

3.3.4 Dijkstran algoritmi

Dijkstran algoritmi on Edsger Dijkstran vuonna 1959 julkaisema algoritmi lyhimmän reitin etsimiseen. Edsger Dijkstra oli hollantilainen tietojenkäsittelytieteen tutkija. Dijkstran algoritmi käyttää painotettua graafia löytääkseen lyhimmän reitin. Painotetussa graafissa jokaisen graafin pisteen välille annetaan arvo, joita algoritmi pitää muistissaan. (Britannica, 2019; MathWorld, n.d.)

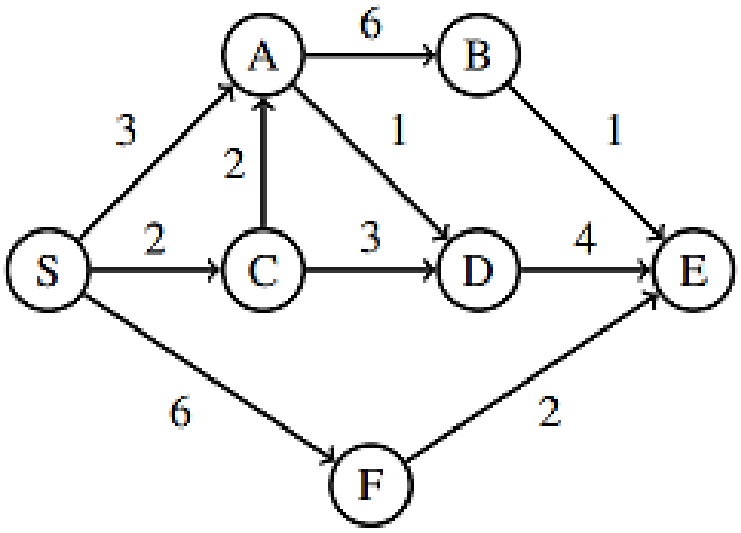
Dijkstran algoritmi aloittaa sille annetusta aloituspisteestä reitin etsimisen. Algoritmi lähtee aloituspisteestään liikkeelle käyden yksi kerrallaan läpi jokaisen mahdollisen vierekkäisen kärkipisteen. Algoritmi lähtee laajenemaan ulospäin alkupisteestään, kunnes se löytää loppupisteen. Kuvassa 5 on ruudukossa merkitty sinisellä alue, jota algoritmi on käynyt läpi ja valkoisella katkoviivalla merkitään reitti, jota se kulkee loppupisteeseen. (Patel, 2019)



Kuva 5. Dijkstran algoritmin liikkuminen ruudukossa (Patel, 2019).

Algoritmille luodaan kaksi listaa. Ensimmäisessä listassa se pitää jo käydyt pisteet, jottei algoritmi käy niitä uudestaan. Toisessa listassa algoritmilla on pisteiden väliset arvot, joilla se pystyy loppupisteen löytyessä laskemaan lyhimmän reitin alku- ja loppupisteen välille. Pisteiden väliset arvot voivat vaihdella esimerkiksi, jos tiedetään että tie on huonokuntoinen ja tämän vuoksi sillä on hankala liikkua. Algoritmi antaa tällöin tuolle reitille suuremman arvon eikä välttämättä käytä tuota reittiä, jos se on löytänyt

loppupisteeseen pienempi arvoisen reitin. Algoritmi laskee yhteen pisteitten väliset arvot ja pienimmän arvon saanut reitti on nopein reitti loppupisteeseen. (Brilliant, n.d.) Kuvassa 6 on esimerkin avulla visualisoitu dijkstran algoritmin kulkua graafin läpi.



Kuva 6. Dijkstran algoritmin graafissa (Brilliant, n.d.).

Kuvassa 6 olevan graafin alkupisteeksi annetaan S ja loppupisteeksi E.

1. Alkutilanteessa on pisteellä S-arvona 0. Pisteestä S algoritmilla on mahdollisuus liikkua pisteisiin A, C ja F.
2. Pisteitten välisten arvojen perusteella algoritmi päättää liikkua pisteeseen C ja antaa tämän jälkeen C pisteelle arvon 2. Pisteestä C liikkuminen pisteeseen A asettaisi reitin kokonaisarvoksi 4 ja D:hen liikkumisen arvoksi tulisi 5.
3. Algoritmi liikkuu pisteestä S pisteeseen A. Pisteeseen A liikkumisen arvoksi tulee 3.
4. A pisteestä liikutaan pisteeseen D. Reitin kokonaisarvoksi tulee 4, joka on tähän asti pienin luku reittien välillä.
5. Algoritmi liikkuu pisteestä S pisteeseen F. Pisteen F-arvoksi tulee 6.
6. Algoritmi liikkuu pisteestä D pisteeseen E löytäen loppupisteen. Pisteen E-arvoksi tulee 8.

Algoritmi kertoo lyhimmäksi reitiksi kulkea reittiä S, A, D, E. Reitti S, F, E olisi ollut samanarvoinen reitti, mutta pisteet D ja E olivat lähempänä toisiaan, joten algoritmi löysi reittinsä tuota kautta ensin. (Brilliant, n.d.; GeeksforGeeks, n.d.)

Dijkstran algoritmi löytää varmasti lyhimmän reitin loppupisteeseensä, kunhan se on mahdollista löytää. Algoritmilla ei ole alussa tietoa missä suunnassa sen loppupiste on kuten Paras ensin -haku. Tämän vuoksi

Dijkstran algoritmi tekee paljon enemmän töitä kuin Paras ensin -haku koska se lähtee sokkona etsimään loppupistettään. (Patel, 2019)

3.3.5 A*-algoritmi

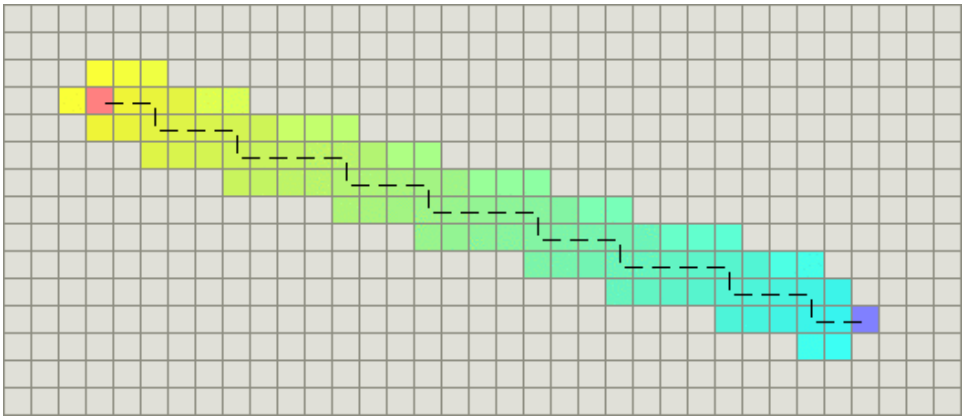
A*-algoritmi (A star) on yksi suosituimmista ja tunnetuimmista reitinhakualgoritmeista. A*-algoritmissa on yhdistetty Dijkstran algoritmia sekä Paras ensin -hakua. A* käyttää samanlaista painotettua graafia kuin dijkstra jolla se pystyy laskemaan lyhimmän reitin. A* keskittyy myös pääasiassa siihen suuntaan, jossa sille annettu loppupiste on kuten Paras ensin -haussa. (Patel, 2019)

Aina kun liikutaan uuteen pisteeseen algoritmi käyttää seuraavanlaista laskukaaviota f(n)=g(n)+h(n). Kaaviossa n on graafissa seuraavana oleva piste. F(n) on pisteitten välinen yhteen laskettu arvo. G(n) on reitin arvo alkupisteestä seuraavan olevaan pisteeseen (n). H(n) on heuristinen funktio, jolla arvioidaan pieniarvoisin reitti seuraavana olevan pisteen (n) ja loppupisteen välille. (Gulsanober, 2019)

Kuten Paras ensin -haussa, A*-algoritmi käyttää myös heuristista funktiota kohdistaakseen hakunsa paremmin. Heuristisen funktion avulla algoritmi pystyy päättämään mihin pisteeseen olisi missäkin kohtaa paras liikkua. Heuristista funktiota voidaan myös muunnella, joka tekee A*-algoritmista hyvinkin joustavan. Heuristiikkaa muuttamalla pystytään algoritmin käyttäytymistä muuttamaan sen käydessä graafia läpi. Heuristiikan muuttaminen vaikuttaa algoritmiin Patelin (2019) mukaan esimerkiksi seuraavasti:

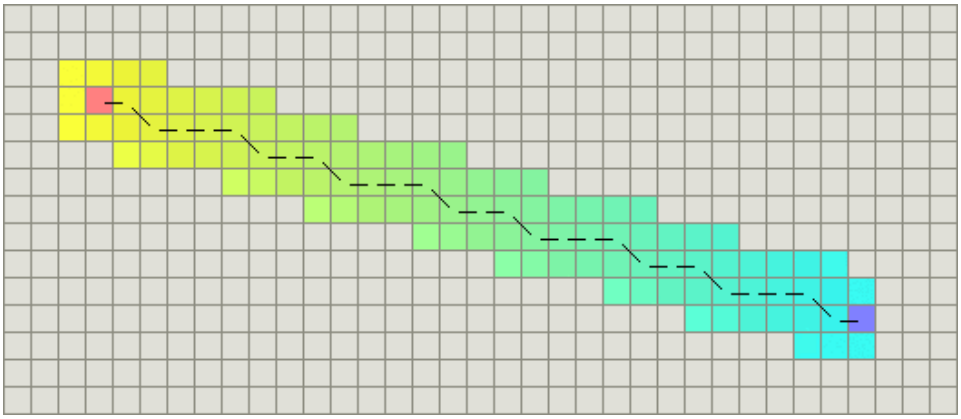
1. Jos muutetaan heuristinen funktio h(n) nollaksi käytetään tällöin vain reitin arvoa alkupisteestä seuraavana olevaan pisteeseen g(n). Tämä tekisi A*-algoritmista Dijkstran algoritmin, joka löytäisi varmasti lyhimmän reitin, mutta olisi myös hitaampi.
2. Jos h(n) on asetettu olemaan aina pienempi tai yhtä suuri kuin liikkuminen seuraavaan pisteeseen (n) käy algoritmi tuolloin suurempaa aluetta läpi, joka tekee siitä hitaamman. Algoritmi löytää kuitenkin parhaan reitin myös tällä tavoin.
3. Jos h(n) on suurempi kuin n tulee tällöin A*-algoritmista nopeampi. Tällöin ei kuitenkaan ole varmaa, että reitti olisi lyhin mahdollinen.
4. Jos h(n) arvo muutetaan todella paljon suuremmaksi kuin g(n), käytetäisiin tuolloin vain h(n) arvoa. Tämä muuttaisi A*-algoritmin samankaltaiseksi kuin Paras ensin -hakualgoritmi.

Heuristiikkaa muuttamalla pystytään siis A*-algoritmia muuttamaan siihen tarkoitukseen mitä tilanne vaatii. Algoritmista pystytään tekemään hitaampi, mutta tällöin siitä tulee myös tarkempi reittiä haettaessa. Algoritmista voidaan tehdä myös nopea, mutta silloin se menettää tarkkuutensa eikä takaa lyhintä reittiä. (Patel, 2019)



Kuva 7. A*-algoritmin käyttö Manhattan etäisyys heuristiikalla (Patel, 2019).

A*-algoritmissa käytetään erilaisia etäisyysheuristiikoita. Yleisimpiä tällaisia heuristiikoita ovat Manhattan, Viisto ja Euklidinen etäisyys. Etäisyysheuristiikoiden eroavaisuus on se, miten ne pystyvät liikkumaan graafin läpi. Esimerkiksi graafissa, jossa halutaan liikkua vain neljään eri suuntaan, käytetään Manhattanin etäisyyttä. Viisto etäisyysheuristiikka sallii liikkeen kahdeksaan eri suuntaan ja Euklidinen sallii liikkeen kaikkiin suuntiin. Kuvassa 7 on A*-algoritmia käytetty Manhattan etäisyysheuristiikkaa ja kuvassa 8 on käytetty Viisto etäisyysheuristiikkaa. (Patel, 2019)

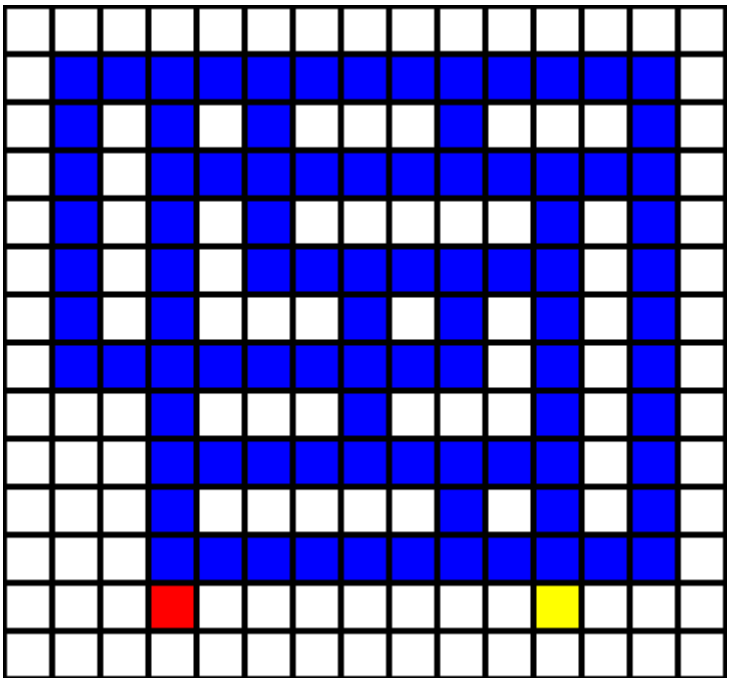


Kuva 8. A*-algoritmin käyttö Viisto etäisyys heuristiikalla (Patel, 2019).

4 OPINNÄYTETYÖN TAVOITE JA TARKOITUS

Opinnäytetyön tavoitteena on saada tekoälymenetelmiä käyttäen rakennettua reitinratkaisija ohjelma. Ratkaisijalle pystyy antamaan erilaisia sokkeloiden pohjapiirroksia, joista se osaa laskea lyhyimmän reitin alku- ja loppupisteestä. Ratkaisija liitetään tietojenkäsittelyn koulutusohjelman ICT-projektissa tehdyn robotin kanssa, jotta robotti pystyy itse kulkemaan sokkelon läpi.

Työn alussa tarkoituksena on saada reitinratkaisija suorittamaan alla oleva sokkelo (kuva 9) liikkumalla vain eteenpäin tai kääntymällä oikealle. Sokkelon reitin etsiminen aloitetaan oikealta alakulmasta keltaisen ruudun kohdalta ja pyritään pääsemään vasempaan alakulmaan punaiseen ruutuun. Ratkaisijaa aletaan rakentamaan käyttämällä A*-algoritmia ja muokkaamalla sitä tarvittavaan tehtävään.



Kuva 9. Sokkelo, jonka reitinratkaisija halutaan suorittavan.

Opinnäytetyö tehdään toiminnallisena opinnäytetyönä. Toiminnallisen opinnäytetyön tavoitteena on esimerkiksi ohjeistaa tai järkeistää käytännön toimintoja. Tämän työn tuotoksena tulee olemaan Python-ohjelmointikielellä kirjoitettu reitinratkaisijaohjelma.

5 TOIMINNALLISEN OPINNÄYTETYÖN PROSESSI

Opinnäytetyön toiminnallinen osuus tehtiin Visual Studio Code (VS Code) tekstieditorilla. VS Code on ilmainen ja kevyt editori, joka on saatavilla useille eri alustoille kuten Windowsille ja Linuxille. VS Code valittiin opinnäytetyön editoriksi, koska se oli jo entuudestaan tuttu ja siihen pystyy lisäämään erilaisia laajennuksia. Yksi laajennus, joka VS Codeen lisättiin, oli Python-ohjelmointikielen laajennus. Opinnäytetyön aikana tehty ohjelma kirjoitettiin kokonaan käyttäen Pythonia ohjelmointikielenä.

5.1 Reitinratkaisija

Opinnäytetyössä tehtävään reitinratkaisijaan valittiin reitinhakualgoritmiksi A* (A star). A* valittiin reitinhakualgoritmiksi koska se on todella suosittu, tehokas ja joustava algoritmi. Sen avulla pystytään myös löytämään varmasti lyhin reitti sokkelosta. Reitinratkaisija pystyy lukemaan sille annettavan sokkelon joko kuvasta tai erilaisista ASCII-piirustuksista. ASCII-piirustuksilla tarkoitetaan kuvaa, joka on piirretty käyttämällä yleisesti tietokoneen näppäimistöstä löytyvillä merkeillä. Tällaisissa kuvissa voidaan esimerkiksi kuvata 1:lla seinää ja 0:lla aluetta, johon voidaan liikkua.

Reitinratkaisijalle haluttiin myös antaa mahdollisuus asettaa erilaisia rajoitteita sille, miten se etsii reittiä sokkelon läpi. Tämä tarkoitti sitä, että haluttiin asettaa liikkumiseen rajoite, ettei ratkaisija pystyisi liikkumaan sokkeloa kulkiessaan kuin eteenpäin tai kääntymään oikealle. Ratkaisijalle voitaisiin myös muuttaa näitä rajoitteita esimerkiksi vaihtamalla vain oikealle kääntyminen vain vasemmalle kääntymiseksi.

A*-algoritmiin löytyi helposti erilaisia oppaita ja pseudokoodia. Pseudokoodilla tarkoitetaan koodia, joka ei ole mitään ohjelmointikieltä. Pseudokoodilla yritetään selittää algoritmista vain sen perusrakenne jättämällä pois kaikki eri ohjelmointikielten erilaisuudet pois. Kuvassa 10 on lyhyt osa A*-algoritmin pseudokoodista, jossa on yritetty selittää koodin toiminto tavalla, jolla sitä pystyisi käyttämään eri ohjelmointikielillä.

```
// Loop until you find the end
while the openList is not empty

    // Get the current node
    let the currentNode equal the node with the least f value
    remove the currentNode from the openList
    add the currentNode to the closedList

    // Found the goal
    if currentNode is the goal
        Congratz! You've found the end! Backtrack to get path
```

Kuva 10. A*-algoritmin pseudokoodia (Swift 2017).

Työn aikana käytettiin muutamia eri oppaita liittyen A*-algoritmiin. Tämän opinnäytetyön pohjana käytettiin Baijayanta Royn kirjoittamaa artikkelia A*-algoritmista. (Baijayanta, 2019) Artikkelissa on opastettu selkeästi A*-algoritmi koodin rakenne. Työn aikana muokattiin A*-algoritmia ICT-projektissa rakennetun legorobotin tarpeisiin. Algoritmi ei alussa ota huomioon suuntaa, johon se on kulkemassa, joten sille oli tehtävä niin sanottu suuntavaisto. Suuntavaiston avulla algoritmille voitaisiin asettaa erilaisia liikkumisrajoitteita. Algoritmiin oli myös lisättävä yksinkertainen visualisointi, jolla sen kulkua sokkelossa voitaisiin seurata.

Aluksi reitinratkaisijalle tehtiin luokka Node (kuva 11), jolla jokaisesta pisteestä sokkelossa tehdään objekti. Pythonissa funktioita määritetään def avainsanalla. Ensimmäisessä Node-luokan funktiossa kerrotaan pisteille, joista sokkelo koostuu, niiden attribuutit. Position kertoo reitinratkaisijalle pisteen, jolla ratkaisija on sillä hetkellä. Direction attribuutilla kerrotaan suunta, josta ollaan tulossa seuraavaan pisteeseen. G, h ja f antavat pisteelle tarvittavat arvot, joilla lasketaan pisteen arvo lyhintä reittiä varten. G:n arvo esittää polun tarkkaa kustannusta aloituspisteestä mihin tahansa pisteeseen. H-arvo on heuristinen arvio mistä tahansa pisteestä loppupisteeseen. F on pisteen kokonaiskustannusarvo.

Node-luokan __eq__-funktiossa seurataan mistä suunnasta ollaan seuraavaan pisteeseen liikkumassa. A*-algoritmi piirtää normaalisti niin sanotusti viivaa perässään, jota se ei pysty ylittämään. Tulosuunnan avulla pystytään sokkeloa läpi käydessä liikkumaan risteyksissä samojen pisteiden yli, jos tulosuunta pisteeseen on eri kuin aikaisemmin pisteessä käydessä.

```
class Node:

    def __init__(self, parent=None, position=None, direction=UP):
        self.parent = parent
        self.position = position
        self.direction = direction

        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        if self.position == other.position and self.direction == other.direction:
            return True
        elif self.position == other.position and other.direction == ANY:
            return True
        else:
            return False
```

Kuva 11. Reitinhakijan Node-luokka (Muokatusti Baijayanta, 2019)

Reitinratkaisijan etäisyysheuristiikkana käytettiin Manhattanin etäisyyttä. Manhattanin etäisyydellä annetaan ratkaisijalle mahdollisuus liikkua neljään eri suuntaan ylös, oikealle, alas ja vasemmalle. Select_moves-funktiossa (kuva 12) on listattu nämä mahdolliset liikkumissuunnat. Liikkumissuunnat on merkitty x- ja y-koordinaateilla kaksiulotteiseen moves-

taulukkoon. Select_moves-funktio (kuva 12) saa muuttujat direction_constraints, jolla sille kerrotaan mitkä suunnat ovat tällä hetkellä sallittuja liikkua ja direction, jossa on suunta, josta ollaan tulossa pisteeseen. Kelaa-funktiolla muutetaan tulosuuntaa, kun liikutaan pisteiden välillä oikealle.

```
def wind_up(lista, n):
    tmp = lista.copy()
    i = 1
    while i<n:
        tmp = [tmp.pop(len(tmp)-1)] + tmp
        i=i+1

    return tmp

def select_moves(direction_constraints, direction):
    fil=direction_constraints.copy()
    moves  =  [[-1, 0 ], # up
              [ 0, 1 ], # right
              [ 1, 0 ], # down
              [ 0, -1]] # left
    fil = wind_up(fil, direction)
    return list(compress(moves, fil))
```

Kuva 12. Reitinratkaisijan liikkumismahdollisuudet

Reitinhakijan search-funktiossa (kuva 13) tapahtuu suurin työ mitä A*-algoritmi tekee. Funktion alussa alustetaan alku- ja loppupisteet (start_node ja end_node). Start_nodelle kerrotaan mistä pisteestä aloitetaan haku ja end_node on piste, johon haku lopetetaan. Alku- ja loppupisteiden jälkeen alustetaan yet_to_visit_list ja visited_list. Visited_listaan lisätään pisteet, joissa on jo käyty, jotta näissä ei käydä useammin. Jos ei pidetä muistissa pisteitä, joissa on jo käyty, voi reitinhakija jäädä ikuiseen silmukkaan hakemaan reittiä samoista pisteistä. Silmukoilla tarkoitetaan koodin osaa, jota toistetaan niin kauan kuin sille annettu ehto on tosi. Pisteisiin saa kuitenkin tulla uudestaan, jos tulosuunta pisteeseen on eri.

Yet_to_visit_list lisätään aloituspiste ja pisteet, joissa ei olla vielä käyty. Outer_iterations ja max_iterationseilla estetään ikuisen silmukan syntymistä. Max_iterations jakaa ensin kahdella sokkelon koon ja laittaa saadun luvun potenssiin 10. Outer_iterations lisää itseensä aina yhden jokaisen silmukan alussa. Jos Outer_iterations kasvaa suuremmaksi kuin max_iterations lopettaa reitinhakija ohjelman. Viimeisellä rivillä selvitetään montako riviä ja saraketta sokkelossa on.

```
def search(maze, cost, start, end):

    start_node = Node(None, tuple(start))
    end_node = Node(None, tuple(end), ANY)

    yet_to_visit_list = []
    visited_list = []

    yet_to_visit_list.append(start_node)

    outer_iterations = 0
    max_iterations = (len(maze) // 2) ** 10

    no_rows, no_columns = np.shape(maze)
```

Kuva 13. Reitinratkaisijan search-funktion alku (Muokatusti Baijayanta, 2019)

Seuraavaksi aloitetaan search-funktiossa ohjelman pääsilmukka, jota jatketaan niin kauan, kunnes on löydetty loppupiste tai sokkelosta ei ole mahdollista löytää annettua pistettä. Funktion pääsilmukassa käytetään while-silmukkaa. While-silmukka toistaa sen sisällä olevaa ohjelmaa niin kauan kuin sille annettu ehto on tosi. Silmukan alussa lisätään outer_iterationsiin aina yksi, jonka avulla pystyttään määrittelemään, jos ollaan ikuisessa silmukassa ja reitinhaku lopetetaan.

Tämän jälkeen verrataan nykyisen pisteen f-arvoa muihin mahdollisten pisteiden f-arvoon. Reitinhakija lähtee laajentamaan hakuaan suuntaan, josta saadaan pienin f-arvo. Kun on löydetty pienimmän f-arvon omaava piste, siirretään nykyinen piste pois listalta, jossa voitaisiin vielä käydä ja siirretään se listaan, jossa on jo käyty. Seuraavaksi liikutaan pisteeseen, jolla oli pienin f-arvo ja tarkastetaan, onko nykyinen piste loppupiste. Jos nykyinen piste on loppupiste, lopetetaan silmukka ja kutsutaan return_path-funktiota, joka piirtää reitin alkupisteestä loppupisteeseen. Funktio return_path (kuva 14) pitää sisällään listaa path, jossa on muistissa ne pisteet, joita pitkin kulkemalla päästään aloituspisteestä loppupisteeseen parhaiten.

```
def return_path(current_node,maze):
    path = []
    no_rows, no_columns = np.shape(maze)
    result = [[-1 for i in range(no_columns)] for j in range(no_rows)]
    current = current_node
    while current is not None:
        path.append(current.position)
        current = current.parent
    path = path[::-1]
    start_value = 0
    for i in range(len(path)):
        result[path[i][0]][path[i][1]] = start_value
        start_value += 1
    return result
```

Kuva 14. Funktio return_path jolla kerrotaan paras reitti sen löytyessä (Baijayanta, 2019)

Search-funktion while-silmukan sisällä on myös muutamia erillisiä silmukoita, joita tarvitaan, jotta algoritmi toimisi moitteettomasti. Yksi tärkeimmistä on for-silmukka, jolla seurataan pisteen nykyistä sijaintia ja onko tuon sijainnin ympärillä mahdollisia muita pisteitä, joille voidaan liikkua seuraavaksi (kuva 15). Silmukalle annetaan taulukkomuodossa kaikki suunnat, joista se voi hakea seuraavaa mahdollista pistettä. For-silmukka toistaa sen sisällä olevan ohjelman jokaisen taulukossa olevan esineen kohdalla. For-silmukan alussa tarkastetaan tulosuunta pisteelle. Jos pisteessä on jo käyty, mutta tulosuunta pisteeseen on eri kuin aiemmin, on siihen tuolloin mahdollista liikkua.

Node_position antaa pisteen sijainnin sokkelossa ja siitä tehdään niin sanottu vanhempipiste ja tämän pisteen ympärillä olevista pisteistä tehdään vanhempipisteen lapsia. Seuraavaksi tarkastetaan node_positionin ympärillä olevat pisteet. Pisteistä tarkastetaan ovatko ne liikkumiskantaman sisällä ja ovatko ne liikkumiskelvollisia eivätkä esimerkiksi seiniä. Kelvollisista pisteistä tehdään new_node ja nämä lisätään children-listaan.

```
for new_position in moves:
    direction = check_direction(new_position)

    node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

    if (node_position[0] > (no_rows - 1) or
        node_position[0] < 0 or
        node_position[1] > (no_columns -1) or
        node_position[1] < 0):
        continue

    if maze[node_position[0]][node_position[1]] != 0:
        continue

    new_node = Node(current_node, node_position, direction)

    children.append(new_node)
```

Kuva 15. For-silmukka pisteen sijainnin seuraamiseksi ja viereisten pisteiden tutkimiseen (Baijayanta, 2019)

Edellisen for-silmukan jälkeen tulee toinen for-silmukka, jolla käydään läpi kaikki lapsipisteet children-listassa (kuva 16). Aluksi tarkastetaan, onko lapsipiste jo käytyjen pisteiden listalla (visited_list). Jos lapsipiste on jo tuolla listalla, se ohitetaan ja tarkastetaan seuraava piste. Tämän jälkeen lasketaan lapsipisteille niiden g, h ja f arvot. Arvojen laskun jälkeen tarkastetaan, onko lapsipistettä vielä yet_to_visit_list-listassa. Jos lapsipistettä ei ole vielä yet_to_visit_list-listalla, lisätään se siihen. Jos lapsipiste kuitenkin on jo yet_to_visit_list-listalla tarkastetaan tällöin sen g-arvo. Alempi g-arvo verrattuna muihin mahdollisiin pisteisiin tarkoittaa, että tuo reitti on parempi.

```
for child in children:

    if len([visited_child for visited_child in visited_list if visited_child == child]) > 0:
        continue

    child.g = current_node.g + cost
    child.h = (((child.position[0] - end_node.position[0]) ** 2) +
               ((child.position[1] - end_node.position[1]) ** 2))
    child.f = child.g + child.h

    if len([i for i in yet_to_visit_list if child == i and child.g > i.g]) > 0:
        continue

    yet_to_visit_list.append(child)
```

Kuva 16. For-silmukka lapsi pisteiden läpikäymiseksi (Baijayanta, 2019)

Ohjelman lopussa käynnistetään reitinratkaisijan pääohjelma muuttujan __name__ kautta. Muuttuja __name__ toteuttaa pääfunktion toiminnallisuuden. Reitinratkaisija lähtee käyntiin, kun pääohjelma käynnistyy. Pääohjelman sisälle on piirretty sokkelon pohjapiirros numeroista 1 ja 0. Pohjapiirroksessa numero 0 on piste, johon on mahdollista liikkua ja 1 on seinä, jonka läpi ei voi liikkua. Reitinratkaisija saa pääohjelman kautta tämän pohjapiirroksen, jota se pystyy käyttämään reittiä hakiessaan. Pääohjelman sisään asetetaan myös x- ja y-koordinaatit aloitus- ja loppupisteille. Pääohjelman lopuksi tulostetaan konsoliin ne x- ja y-koordinaatit, joita pitkin on paras kulkea.

5.2 Sokkelon pohjapiirroksen visualisointi

Reitinratkaisijassa käytettävä sokkelon pohjapiirros on aluksi vain koodiin numeroista 0 ja 1 piirretty kuva. Tällaisesta pohjapiirroksesta reitinratkaisija saa kaikki tarvittavat tiedot eli mistä pisteestä haetaan reittiä mihinkin pisteeseen. Pohjapiirroksesta reitinratkaisija myös näkee sokkelon seinät ja alueet, joissa voidaan liikkua. Reitinratkaisijaa ei pysty myöskään seuraamaan silloin kun se käy läpi sokkeloa. Reitinratkaisijan käyttäjä ei siis näe mitä ratkaisija tekee etsiessään parasta reittiä. Ohjelman lopussa, kun paras reitti on löydetty, tulostaa ratkaisija konsoliin ne koordinaatit, joita pitkin kulkemalla pääsee alkupisteestä loppupisteeseen parhaiten.

Tällainen tyyli katsoa pohjapiirrosta ja seurata reitinratkaisijaa ei ole välttämättä aina käytännöllinen. Käyttäjän saattaa olla melko hankala erottaa kuvasta numeroita toisistaan. 0:sta ja 1:stä piirretyn kuvan piirtäminen käsin koodin sisään on myös melko hankalaa ja aikaa vievää. Reitinratkaisijan työskentelyn seuraaminen, sen käydessä sokkeloa läpi, on käyttäjälle hyödyllistä. Reitinratkaisijaa seuraamalla voidaan mahdollisesti löytää siinä mahdollisia olevia virheitä.

Opinnäytetyön alussa suunniteltiin pohjapiirroksen muuttamista tavalliseksi png-kuvaksi. Ratkaisijalle voitaisiin png-kuvasta kertoa samalla tavalla esimerkiksi, että musta väri kuvassa tarkoittaa seinää ja valkoinen väri tarkoittaa aluetta, johon voidaan liikkua. Tällaisen mustavalkoisen kuvan tekeminen ei ole hankalaa ja siitä on käyttäjän helppo erottaa seinät ja alueet. Mustavalkoisen sokkelon kuvan piirtäminen onnistuu melko helposti lähes millä tahansa kuvaeditoriohjelmalla.

Png-kuvan ollessa isokokoinen tuli reitinetsimisen ongelmaksi hitaus. Reitinratkaisijalle jokainen pikseli kuvassa on yksi piste, joka sen täytyy käydä läpi löytääkseen parhaan mahdollisen reitin. Isommissa kuvissa saattaa olla tuhansia pikseleitä ja näiden läpi käyminen ratkaisijalle voi kestää useita minuutteja.

Reitinratkaisijan merkitsemä paras reitti alkupisteestä loppupisteeseen merkittiin punaisella. Ratkaisijan merkitsemää reittiä on myös hankala seurata kuvasta, koska se merkitsee vain pikselit, joita pitkin se on kulkenut. Ohjelma toimii tällöin kuten sen pitääkin, mutta pikselin kokoista viivaa on käyttäjän yleensä melko hankala erottaa muusta kuvasta. Reitinratkaisijan työskentelyä tällä tyylillä, sen hakiessa reittiä, ei voitu seurata reaaliajassa.

Reitinratkaisijan visualisointiin käytettiin loppujen lopuksi kilpikonnagrafiikkaa (turtle graphics), joka on vektorigrafiikkaa. Kilpikonnagrafiikan avulla voidaan niin sanotun kilpikonnan avulla piirtää esimerkiksi tietokoneen ruudulle erikokoisia ja erivärisiä muotoja. Kuvassa 17 luodaan ensiksi ruutu. Ruudulle voidaan itse asettaa sen taustaväri ja koko. Seuraavaksi tehdään luokka, jossa kerrotaan piirrettävälle kohteelle sen muoto ja väri. Kilpikonnagrafiikan piirtäessä se jättää jälkeensä viivaa, joka on poistettu käyttämällä penup:a. Lopuksi asetetaan kilpikonnalle speed. Sillä voidaan asettaa animoinnille, jos sellaista käytetään, sen nopeus millä se esimerkiksi kääntyy.

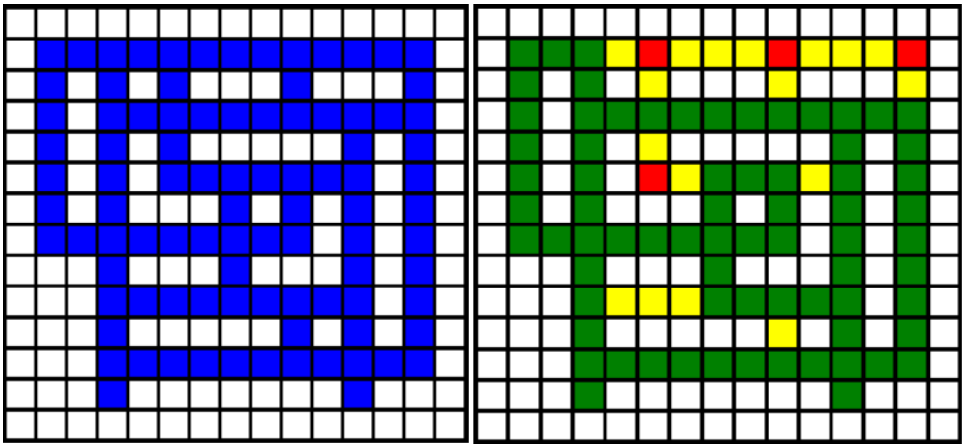
```
wn = turtle.Screen()
wn.bgcolor("black")
wn.setup(1000,700)

class Maze(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")
        self.color("white")
        self.penup()
        self.speed(0)
```

Kuva 17. Kilpikonnagraafikan alustus

Ratkaisijan visualisoinnissa esimerkiksi numerot 1, jotka kuvaavat seiniä, on piirretty valkoisiksi neliöiksi. Numerot 0, jotka kuvaavat aluetta, johon voidaan liikkua, on piirretty sinisiksi neliöiksi. Kilpikonnagrafiikalla voitiin helposti seurata reitinratkaisijan kulkua sen käydessä sokkeloa läpi. Kun ratkaisija liikkuu uuteen pisteeseen, piirretään sen paikalle keltainen neliö.

Opinnäytetyössä haluttiin myös merkitä jokainen mahdollinen käännöskohta, joita sokkelossa voi olla. Käännöksiä merkattiin punaisilla neliöillä. Lopuksi kun ratkaisija on löytänyt reittinsä läpi sokkelon merkkaa se parhaan reitin vihreillä neliöillä. Kuvassa 18 on vasemmalla puolella kuva sokkelosta, jossa on piirretty seinät ja liikkumiseen sallitut alueet. Oikealla puolella on kuva sokkelosta, kun reitinratkaisija on löytänyt parhaan reitin.



Kuva 18. Sokkelon pohjapiirros ennen ja jälkeen reitin löytämistä

Kilpikonnagrafiikan avulla saatiin rakennettua selkeästi seurattava visualisointi reitinratkaisijalle. Sokkelon muuttaminen kuitenkin tällä tyylillä on hankalaa. Tätä tyyliä käyttämällä joutuu käyttäjä piirtämään sokkeloa itse koodin puolelta. Kilpikonnagrafiikkaa on kuitenkin selkeämpi seurata kuin kahta edellistä vaihtoehtoa eli konsoliin piirrettyä reittiä ja png-kuvaa.

6 JOHTOPÄÄTÖKSET JA POHDINTA

Opinnäytetyön prosessi aloitettiin valitsemalla työn aihe. Ammattikorkeakoulun kautta saaduista aiheista valitsin syksyllä ICT-projektissa aloitetun legorobotin jatkokehityksen. Työssä yhdistyi muutamia itseäni kiinnostavia aihealueita kuten tekoäly, reitinhaku ja Python-ohjelmointikieli. Opinnäytetyö oli melko haastava, koska perehtyminen näihin itselle uusiin aiheisiin vei paljon aikaa.

6.1 Tutkimuskysymyksien onnistuminen

Opinnäytetyöhön asetettuihin tutkimuskysymyksiin saatiin vastattua onnistuneesti. Työn tavoitteena oli saada rakennettua reitinratkaisija käyttämällä A*-algoritmia. Työn aikana oli myös tarkoitus rakentaa ratkaisijalle mahdollisuus muuttaa sen tapaa liikkua sokkelon läpi. Opinnäytetyön aikana tutustuttiin tekoälyn historiaan ja muutamiin erilaisiin reitinhakualgoritmeihin. Tavoitteena oli myös saada reitinratkaisijalle helppo ja yksinkertainen visualisointi, joka näyttäisi sen käyttäjälle parhaan mahdollisen reitin.

Opinnäytetyön aikana saatiin rakennettua toimiva reitinratkaisija käyttämällä A*-algoritmia, jolle pystyttiin asettamaan rajoituksia sen liikkumiselle. Ratkaisijalla pystytään löytämään erilaisista sokkeloista lyhin reitti alku- ja loppupisteen väliltä. Ratkaisijalle voidaan myös asettaa esimerkiksi ehto, että se ei voi sokkeloa selvittäessään liikkua kuin eteenpäin tai oikealle. Tätä rajoitetta pystytään helposti muuttamaan erilaiseksi esimerkiksi vain eteenpäin ja vasemmalle kääntymiseksi tai voidaan sallia liikkuminen kaikkiin mahdollisiin suuntiin.

Reitinratkaisijalle saatiin tehtyä yksinkertainen visualisointi käyttämällä kilpikonnagrafiikkaa. Kilpikonnagrafiikan avulla saatiin koodissa kirjoitettu sokkelon pohjapiirros piirrettyä ruudulle. Sokkelon pohjapiirros saadaan piirrettyä käyttämällä erivärisiä neliöitä, jotka kuvaavat seiniä ja alueita, joihin voidaan liikkua. Reitinratkaisijaa pystytään myös seuraamaan, kun se käy läpi sokkeloa.

6.2 Oppimiskokemukset

Opinnäytetyön aloitin tutustumalla tarkemmin Python-ohjelmointikieleen. Ohjelmointikielen ollessa melko tuntematon oli myös koodin kirjoittaminen aikaa vievää. Opinnäytetyön alussa jouduin etsimään paljon tietoa A*-algoritmista, jolla reitinratkaisijaohjelma oli tarkoitus rakentaa. Opinnäytetyön teoriaosuudessa läpikäydyt erilaiset reitinhakualgoritmit olivat myös täysin uusia asioita. Algoritmeista tutkittiin, kuinka ne erosivat toisistaan ja mitkä niiden hyvät ja huonot puolet olivat.

Reitinratkaisijaan oli päätetty käytettäväksi A*-algoritmia opinnäytetyön ohjaajan kanssa heti työn alussa. Reitinratkaisijan suurimpana ongelmana oli selvittää, miten algoritmille kerrottaisiin mihin suuntaan se on missäkin tilanteessa menossa. A*-algoritmi ei alkuperäisessä tilassaan ymmärrä suuntia esimerkiksi eteenpäin tai oikealle, joten sille oli jotenkin rakennettava niin sanottu suuntavaisto. Tähän ongelmaan sain työn aikana apua opinnäytetyön ohjaajalta.

Opinnäytetyön aikana sain uutta tietoa tekoälyn ja reitinhaun menetelmistä. Työn aikana opin paljon reitinhaussa käytettävistä erilaisista algoritmeista ja niiden eroavaisuuksista. Pythonin käyttö opinnäytetyön toiminnallisen osuuden työn tekoon toi uutta tietoa kielestä. Työn aikana oppi esimerkiksi Pythonin syntaksin eroavaisuuksia muista ohjelmointikielistä. Tiedonhakutaitoja tarvittiin kokoajan opinnäytetyön aikana, joten nämäkin taidot paranivat. Opinnäytetyölle asetetussa aikataulussa pysyttiin hyvin.

6.3 **Jatkosuunnitelmat**

Opinnäytetyön alussa suunnitelmana oli liittää reitinratkaisija ICT-projektin aikana tehtyyn legorobottiin. Työn toiminnallisen osuuden aikana ei ollut tarpeeksi aikaa suunnitella ratkaisijan liittämistä robottiin. Reitinratkaisijan jatkokehityksinä voitaisiin ratkaisija liittää robottiin ja ratkaisijan koodia voitaisiin parannella.

Reitinratkaisijan visualisointia on mahdollista parannella. Ratkaisijan kilpikonnagrafiikalla luodusta kuvasta ei saa tarkkaa kuvaa mihin suuntaan ratkaisijan reitti aina kulkee. Ratkaisijasta ei erota, jos se kulkee jo käydyn alueen yli eri suuntaan. Tällaisessa tilanteessa voitaisiin esimerkiksi yrittää käyttää neliöiden sijasta nuolia tai käyttää eri värejä, jos pisteessä on jo käyty.

Png-kuvasta reitin etsimistä voitaisiin myös yrittää kehittää. Ratkaisijan löydettyä kuvasta reitti voitaisiin ratkaisijan kulusta sokkelon läpi tehdä esimerkiksi gif-tiedosto. Gif-tiedostoon pystyttäisiin merkkaamaan ne pisteet, joissa ratkaisija on käynyt ja lopuksi näyttämään paras reitti. Ratkaisijan käydessä pikseli kerrallaan läpi png-kuvaa voitaisiin muuttaa. Ratkaisija voisi käydä kerralla isomman alueen läpi kuin yhden pikselin kuvassa. Parhaan reitin löydettäessä kuvaan merkattu reitti olisi hyvä olla suurempi kuin yhden pikselin kokoinen.

LÄHTEET

Anyoha, Rockwell. (2017). The History of Artificial Intelligence. Haettu 23.1.2020 osoitteesta http://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/

Baijayanta, Roy. (2019). A-Star (A*) Search Algorithm. Haettu 10.5.2020 osoitteesta https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb

Brilliant. (n.d.). Dijkstra's Shortest Path Algorithm. Haettu 29.1.2020 osoitteesta https://brilliant.org/wiki/dijkstras-short-path-finder/#citation-1

Britannica. (2019). Edsger Dijkstra. Haettu 29.1.2020 osoitteesta https://www.britannica.com/biography/Edsger-Dijkstra

Built in. (2019). What is Artificial Intelligence?. Haettu 23.1.2020 osoitteesta https://builtin.com/artificial-intelligence

Copeland, B.J. (2019). Alan Turing. British mathematician and logician. Haettu 23.1.2020 osoitteesta https://www.britannica.com/biography/Alan-Turing

Dataflair. (2018). What is Heuristic Search – Techniques & Hill Climbing in AI. Haettu 30.1.2020 osoitteesta https://data-flair.training/blogs/heuristic-search-ai/

Elmsley, Andy. (2019). Short circuit: breadth-first search, part II. Haettu 23.1.2020 osoitteesta https://medium.com/the-sound-of-ai/short-circuit-breadth-first-search-part-ii-a63f4320d11b

GeeksforGeeks. (n.d.). Best First Search (Informed Search). Haettu 23.1.2020 osoitteesta https://www.geeksforgeeks.org/best-first-search-informed-search/

GeeksforGeeks. (n.d.). Dijkstra’s shortest path algorithm | Greedy Algo-7. Haettu 29.1.2020 osoitteesta https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Gulsanober, Saba. (2019). A-Star Algorithm Python Tutorial – An Introduction To A* Algorithm In Python. Haettu 1.2.2020 osoitteesta https://www.simplifiedpython.net/a-star-algorithm-python-tutorial/

Hackerearth. (n.d.). Breadth First Search. Haettu 23.1.2020 osoitteesta https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Hackerearth. (n.d.). Depth First Search. Haettu 23.1.2020 osoitteesta https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Hemmendinger, David. (2016). Lisp. Computer language. Haettu 25.1.2020 osoitteesta https://www.britannica.com/technology/LISP-computer-language

Lewis, Tanya. (2014). A Brief History of Artificial Intelligence. Haettu 23.1.2020 osoitteesta https://www.livescience.com/49007-history-of-artificial-intelligence.html

MathWorld. (n.d.). Weighted Graph. Haettu 29.1.2020 osoitteesta http://mathworld.wolfram.com/WeightedGraph.html

Neittaanmäki, P. Ojalainen, A. Tuominen, H. Vähäkainu, P. & Äyrämö, S. (2019). *Tekoälyn perusteita ja sovelluksia.* Haettu 1.3.2020 osoitteesta https://tim.jyu.fi/view/kurssit/tie/tiep1000/tekoalyn-sovellukset/kirja#ChaSovellukset

Swift, Nicholas. (2017). Easy A* (star) Pathfinding. Haettu 24.4.2020 osoitteesta https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

Oksa, S. (2014). *Pathfinding in a 3D-environment Using Unity3D*. Opinnäytetyö. Business Information Technology. Kajaanin ammattikorkeakoulu. Haettu 23.1.2020 osoitteesta https://www.theseus.fi/handle/10024/83765

Patel, Amit. (2019). Introduction to A*. Haettu 23.1.2020 osoitteesta http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Patel, Amit. (2019). Heuristics. Haettu 1.2.2020 osoitteesta http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Siukonen, T. & Neittaanmäki, P. (2019). *Mitä tulisi tietää tekoälystä*. Jyväskylä: Docendo Oy.

The Medical Futurist. (2019). From Surgeries To Keeping Company: The Place Of Robots In Healthcare. Haettu 1.3.2020 osoitteesta https://medicalfuturist.com/robotics-healthcare/

Tuominen, H. & Neittaanmäki, P. (2019). *Tekoälyn perusteita ja sovelluksia.* Haettu 1.2.2020 osoitteesta https://tim.jyu.fi/view/kurssit/tie/tiep1000/tekoalyn-sovellukset/kirja#DKUvbnUuGytQ

Tutorialspoint. (n.d.). Graph Theory – Introduction. Haettu 31.1.2020 osoitteesta https://www.tutorialspoint.com/graph_theory/graph_theory_introduction.htm

Wikipedia. (2020). Depth-first search. Haettu 24.4.2020 osoitteesta https://en.wikipedia.org/wiki/Depth-first_search

Zwass, Vladimir. (2016) Expert system. Computer science. Haettu 23.1.2020 osoitteesta https://www.britannica.com/technology/expert-system