A*-algoritmin variantit

Diplomityö
TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos
Ohjelmistotekniikka
Ville Nygård
Joulukuu 2020

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -järjestelmällä.

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos

NYGÅRD,VILLE: A*-algoritmin variantit

Diplomityö, 60 s.
Ohjelmistotekniikka
Joulukuu 2020

Yksittäistä agenttia käsitteleviä paras ensin -hakualgoritmeja on käytetty vuosikymmenien ajan ratkaisuina hyvin monentyyppisiin ja monia aloja koskettaviin ongelmiin. Yksi tärkeimmistä on jo vuonna 1968 esitelty A*-algoritmi (lausutaan "A tähti").

Vaikka A* on hyvin laajalti käytetty, tietyissä tilanteissa se on osoittautunut riittämättömäksi. Tämän vuoksi A*:een on esitelty suuri määrä laajennoksia vuosien saatossa. A*:n laajennoksia on jo vuosien ajan kategorisoitu sen mukaan mitä A*:n ongelmia tai puutteita kyseinen laajennos pyrkii korjaamaan. Vaikka kategorioita on löydettävissä kirjallisuudesta lukuisia, tässä tutkielmassa keskitytään neljään kategoriaan. Näitä ovat *inkrementaaliset*, *anytime*, *reaaliaikaiset* sekä *any-angle*-algoritmit.

Inkrementaaliset algoritmit käyttävät peräkkäisiä keskenään hyvin samankaltaisia hakuja hyväksi seuraavissa hauissa. Anytime-algoritmit kykenevät antamaan toteuttamiskelpoisen arvion reitistä milloin tahansa (paitsi ensimmäisen laskennan aikana). Algoritmin oletetaan kykenevän myös parantamaan reittiä mitä enemmän laskentaan annetaan aikaa. Reaaliaika-algoritmien yhteinen piirre on, että ne kykenevät toimimaan hyvin tarkkojen aikarajojen puitteissa. Any-angle-algoritmit eivät rajoita liikkumista solmujen välillä lainkaan.

Tässä tutkielmassa esitellään ensin yksi algoritmi jokaisesta kategoriasta, D* Lite, ARA*, LSS-LRTA* sekä Field D*. Nämä algoritmit toteutetaan Python-ohjelmointikielellä ja niiden suorituskykyä testataan tuntemattomassa ympäristössä. Tämän jälkeen niiden suorituskykyä testiympäristössä arvioidaan.

Asiasanat: A*, D* Lite, Field D*, ARA*, LSS-LRTA*

Sisällys

Luku 1

Johdanto

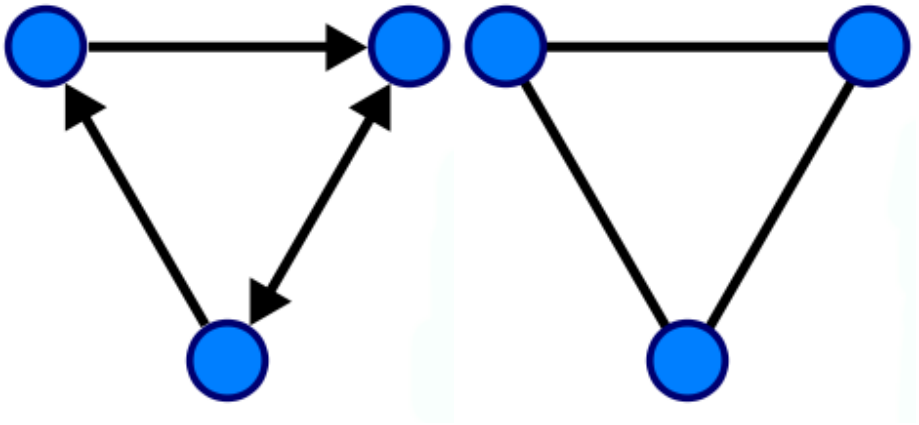
Tietotekniikan käyttö on levinnyt kaikille inhimillisen toiminnan alueille ja nyky-yhteiskunnassa on hyvin vaikea tulla toimeen ilman tietotekniikkaa ja tietokoneita. Tietokoneita käytetään nykyään myös lähes poikkeuksetta kaikilla tieteenaloilla ratkaisemaan reaalimaailman ongelmia. Tietokone ei kuitenkaan kykene ratkaisuihin, jollei ratkaistavaa ongelmaa kyetä mallintamaan johonkin sen ymmärtämään muotoon.

Mallintaminen on mallien luomista ja niiden käyttämistä selittämään ympäröivän todellisuuden ilmiöitä. Jonkin asian malli on kuvaus kyseisestä asiasta, kuten esimerkiksi kartta on malli todellisesta maastosta tai pienoismalli on pienennetty malli todellisesta esineestä. Tietokone ei kykene ymmärtämään karttaa ilman, että siitä luodaan jokin malli sen muistiin, jota se ymmärtää. Filosofi Immanuel Kantin mukaan meillä ei itse asiassa voi olla tietoa maailmasta sinänsä, vaan kaikki, mitä voimme siitä tietää muodostuu havaintojemme ja kokemuksen kautta kertyneistä ajatusmalleista. Kun havaintoja ja kertynyttä kokemusta jäsennetään systemaattisesti, niistä voidaan muodostaa tieteellisiä malleja, so. malleja, jotka täyttävät tieteellisyyden vaatimukset, kuten objektiivisuuden, kriittisyyden, autonomisuuden ja yleisyyden vaatimukset.

Numeerinen, eli laskennallinen, mallinnus on tärkeä tutkimuksen ja palvelujen tuottamisen apuväline. Esimerkiksi säätä, ilmastoa ja muuta ilmakehän toimintaa kuvaavat yhtälöt ovat niin monimutkaisia, että ilmakehän toiminnan selvittämiseksi ja ennustamiseksi on turvauduttava numeeriseen laskentaan tietokoneilla. Tietokoneet toimivat ikään kuin laboratorioina, joissa voidaan tutkia ilmakehää ja ennustaa sen toimintaa erilaisissa olosuhteissa. Vasta tietokoneiden myötä on numeerisesta mallinnuksesta tullut todellinen työkalu monimutkaisten ilmiöiden simulointiin ja tutkimiseen. Tietotekniikan huima kehitys mahdollistaa nykyään entistä tarkemman ilmiöiden kuvauksen ja mallinnuksen. [1] [2]

[3]

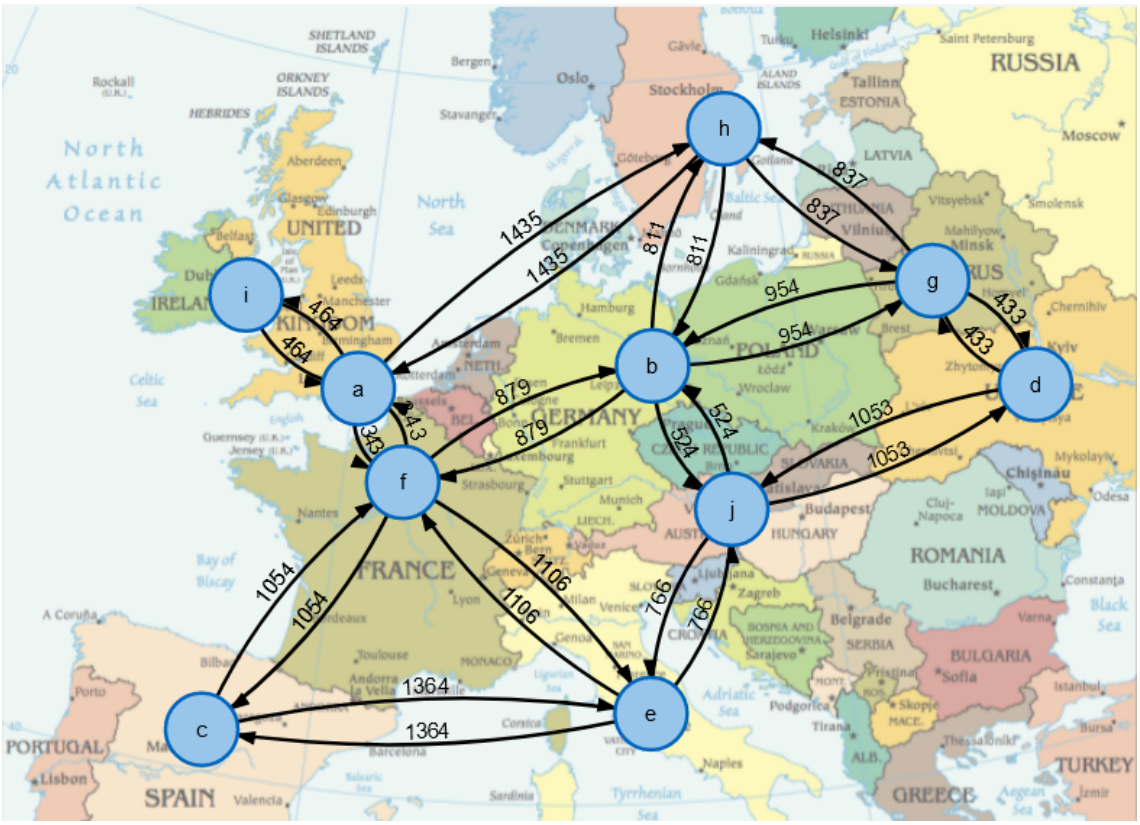
Eräs tapa mallintaa reaalimaailman ongelmia on verkko eli graafi. Graafi koostuu solmuista (*node*, *vertex*), solmuja yhdistävistä särmistä (*edge*, *arc*, *line*) ja solmujen erottamista alueista (*face*). Graafi voi olla suunnattu, jolloin särmille on määritetty suunta. Graafeilla voidaan mallintaa monia asioita ja usein ongelman esittäminen graafimuodossa on eniten kiinni kyvystä hahmottaa sen keskeiset rakenteet sekä mielikuvituksesta. On olemassa tietoverkkoja, liikenneverkkoja, kemiallisia verkkoja, sosiometrisiä verkkoja, lingvistisiä verkkoja, jne.



(a) suunnattu graafi (b) suuntaamaton graafi

Kuva 1.1: Yksinkertaisia graafeja.

Graafi on yksinkertaisesti joukko solmuja, jotka on linkitetty toisiinsa särmillä. Mallinnettavan ongelman mukaan on mahdollista liittää jokaiseen särmään jokin paino (*weight*), joka voi olla vakio tai vaihdella solmujen välillä. Esimerkiksi maantieteellistä karttaa voitaisiin mallintaa graafilla, jossa solmut edustavat kaupunkeja ja solmujen väliset yhteydet eli särmät teitä. Painot voisivat olla kaupunkien välisiä etäisyyksiä (kuva 1.2). Voitaisiin valita kaksi kaupunkia ja kysyä mikä on lyhin mahdollinen reitti näiden välillä eli mitkä graafin särmät tulisi valita, jotta painot huomioon ottaen kustannus kahden valitun solmun välillä olisi mahdollisimman pieni. Ääritapauksessa voitaisiin käydä läpi kaikki mahdolliset reitit valittujen solmujen välillä ja valita näistä pienin arvo. Kuitenkin kun solmujen ja särmien määrä kasvaa, esimerkissä kaupunkien ja teiden, kasvaa mahdollisten kombinaatioiden määrä valtavasti ja lopulta suorittaminen voisi osoittautua liian hitaaksi tai muistinkulutus liian suureksi. Suurilla määrillä solmuja ja särmiä kaikkien mahdollisten kombinaatioiden läpikäynti on osoittautunut käytännössä mahdottomaksi. Tarvitaan siis menetelmä, joka edellä mainitun menetelmän tavoin löytää pienimmän kustannuksen reitin kahden solmun välille ilman, että menetelmän tarvitsee käydä läpi kaikkia mahdollisia

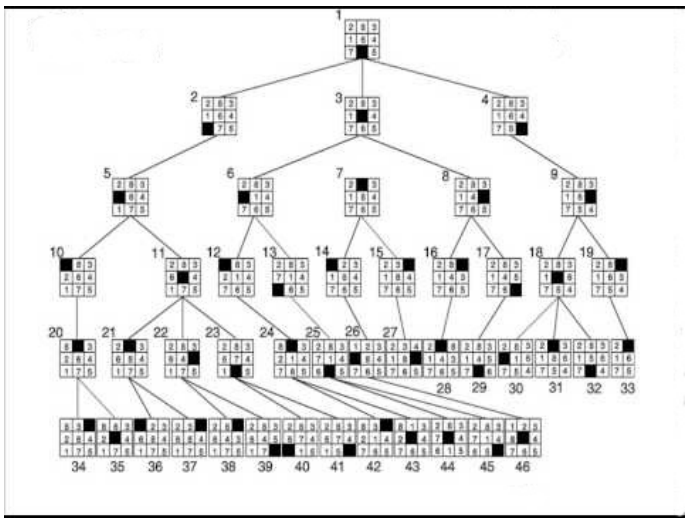


Kuva 1.2: Graafi, joka mallintaa eurooppalaisten kaupunkien etäisyyksiä. Paino on merkitty särmien yläpuolelle. [4]

kombinaatioita. [5] [6]

Erilaisia parempia menetelmiä toteuttaa hakuja graafissa on kehitetty vuosikymmenien ajan ja niitä käytetään nykyään hyvin monilla aloilla, kuten tietokonepeleissä, joissa pelin hahmot joutuvat navigoimaan kartalla sekä väistämään esteitä sekä vihollisia, sekä myös autonomisissa roboteissa, jotka joutuvat väistämään liikkuessaan erilaisia esteitä. Lisäksi menetelmiä voidaan käyttää esim. lentoratojen generointiin sekä biologisten signaalien seurantaan. Kuten edellä graafeja voidaan käyttää myös kustannusten minimointiin. Lyhimmän reitin lisäksi voidaan pyrkiä minimoimaan esim. aikaa, rahaa, polttoaineen kulutusta jne. [7] [8] [9] [10] [11]

Yksi tärkeimmistä algoritmeista on jo vuonna 1968 esitelty A*-algoritmi (lausutaan "A tähti"). A* ei käy kaikkia mahdollisia kombinaatioita läpi vaan se pyrkii löytämään optimaalisen reitin suosimalla reittejä, joita se pitää todennäköisesti parhaina ja jättää muut reitit käsittelemättä kokonaan. [12]



Kuva 1.3: A*-algoritmin toiminta *8-Puzzle*-pelissä. [31]

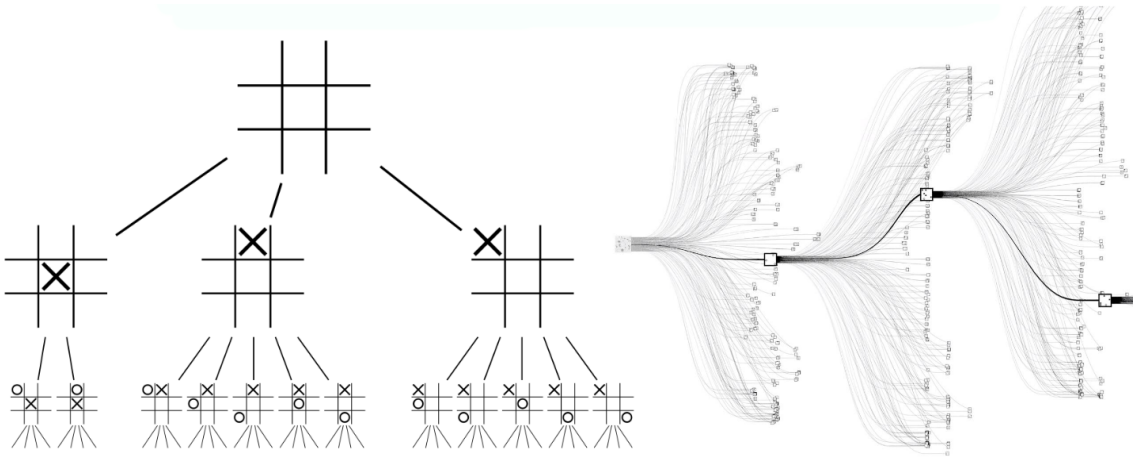
1.1 A*

A*-algoritmin historia alkoi, kun joukko SRI-Internationalin tutkijoita ryhtyi kehittämään liikkuvaa *"automatonia"*. Työn tuloksena syntyi *Shakey the robot*, ensimmäinen liikkuva robotti, joka kykeni liikkumaan itsenäisesti ja havainnoimaan ympäristöään kameralla. Robotin tarpeisiin kehitettiin vuosien 1964 ja 1968 välillä algoritmi, joka kykeni nopeasti hakemaan lyhimmän reitin kahden solmun välillä, kun robotin ympäristöä mallinnettiin graafin avulla. Algoritmi sai nimekseen A*, jossa A tarkoittaa algoritmia ja * kuvaa algoritmin kykyä löytää lyhin reitti. [13]

Algoritmin kehittämisen jälkeen se on ollut hyvin suosittu ja sitä on käytetty niin teollisissa kuin kaupallisissa roboteissa. Sitä on käytetty alkuperäisen tarkoituksen lisäksi myös hyvin monilla muilla tavoilla, kuten karttasovelluksissa, peleissä sekä analysoimaan erilaisia verkkoja.

Peleissä algoritmia on käytetty erityisesti kahdella eri tavalla. Toisaalta sitä voidaan käyttää pelihahmojen liikkumiseen sekä esteiden kiertämiseen peliympäristössä ja toisaalta pelaamaan vuoropohjaisia pelejä tai ratkaisemaan yksinpelejä kuten *8-Puzzle* (kuva 1.3). Näissä peleissä graafi muodostaa puurakenteen, joka on graafin erikoistapaus. Eri solmut vastaavat mahdollisia pelitiloja ja hakualgoritmi pyrkii hakemaan pelitiloja, jossa sen voittoehto toteutuu (kuva 1.4).

Moderneissa tietokonepeleissä useiden pelihahmojen on kyettävä liikkumaan, etsimään reittejä sekä väistämään esteitä sekä muita agentteja. A*-algoritmi on ollut hyvin suosittu tämänkaltaisissa peleissä. A*-algoritmi on suosittu myös reitinhaun sekä navigoinnin



Kuva 1.4: A*-algoritmin toiminta vuoropohjaisissa peleissä ristinolla ja Go. [15]

yhteydessä. [7] [14] [16]

1.2 Työn tavoite

A* on hyvin geneerinen algoritmi, jota voi, kuten aiemmin on todettu, käyttää hyvin monenlaisissa tapauksissa. Lisäksi se on toiminnaltaan hyvin helppo toteuttaa ja ymmärtää. Vaikka A* on hyvin laajalti käytetty, tietyissä tilanteissa se on osoittautunut riittämättömäksi. On havaittu, että se ei skaalaudu hyvin todella suurille graafeille sen suurten muistivaatimusten vuoksi eikä se täytä hyvin tiukkoja reaaliaikavaatimuksia. [23] [25] [17] Nykyisin kuitenkin tietokoneet joutuvat usein käsittelemään entistä suurempia määriä dataa ja toisaalta toimimaan tiukempien aikavaatimusten rajoissa. Tämän vuoksi koko ajan enemmän ja enemmän laajennoksia A*-algoritmiin on esitelty. Nämä algoritmit saattavat olla huomattavasti A*:ä erikoistuneempia eli ne pyrkivät olemaan mahdollisimman tehokkaita ratkaisemaan tietynlaisia ongelmia, mutta ne saattavat olla liian erikoistuneita tai kompleksisia kaikkiin ongelmiin. Siksi voi olla hyvin vaikea sanoa kaikkien mahdollisten A*-varianttien joukosta mikä on sopivin algoritmi, kun pitää esimerkiksi analysoida sosiaalista mediaa.

Tässä tutkielmassa tutkitaan A*:n varianttien toimintaa tuntemattomassa ympäristössä. On havaittu, että A*-algoritmi ei ole paras mahdollinen valinta tällaiseen ympäristöön mutta toisaalta on vaikea sanoa mikä lukuisista A*-algoritmin varianteista olisi paras valinta missäkin tilanteessa. Tutkielmassa toteutetaan tutkimusympäristö, johon toteutetaan 4 algoritmia ja siihen voidaan lisätä niitä jälkikäteen. Algoritmien tehtävänä on löytää lähtösolmusta maalisolmuun ilman, että algoritmilla on etukäteen mitään käsitystä siitä millaisessa graafissa se suorittaa hakuja. Algoritmien suorituksesta kerätään tilastoja ja

niiden suoriutumista arvioidaan tutkielman lopuksi.

Tuloksia voidaan soveltaa tapauksissa, joissa yksittäinen agentti navigoi tuntemattomassa ympäristössä ja etsii reittiä etukäteen määritettyjen solmujen välille. Tällaisia sovellusalueita ovat esimerkiksi tietokone- ja konsolipelit sekä robotiikka. Molemmilla aloilla on kasvavaa tarvetta tiukemmille reaaliaika- sekä muistivaatimuksille, joten on tärkeää, että tapauksen mukaan valitaan optimaalinen algoritmi.

1.3 Työn sisältö

Luvussa 2 esitellään A*-algoritmi tarkemmin. Sen lisäksi esitellään neljä erilaista kategoriaa A*-variantteja, jotka pyrkivät korjaamaan jonkin A*:n puutteen. Luvun lopuksi esitellään joitain A*-algoritmin soveltamisalueita. Luvussa 3 esitellään yksityiskohtaisemmin yksi algoritmi jokaisesta kategoriasta. Nämä algoritmit toteutetaan käytännössä ja niitä testataan tuntemattomassa ympäristössä ja niiden toiminnasta kerätään tilastoja, joita analysoidaan luvussa 5. Luku 6 on yhteenveto tuloksista sekä lopuksi käsitellään mahdollisia jatkotutkimuskohteita.

Luku 2

A*-algoritmien yleinen teoria

Yksittäistä agenttia käsitteleviä paras ensin-hakualgoritmeja on käytetty vuosikymmenien ajan ratkaisuina hyvin monentyyppisiin ja monia aloja koskettaviin ongelmiin. Yksi tärkeimmistä on jo vuonna 1968 esitelty A*-algoritmi. Sitä on käytetty mm. biologiassa, robotiikassa, peleissä sekä klassisissa tekoälyyn liittyvissä ongelmissa. Sen tehtävä on löytää graafin solmusta x_{start} lyhin reitti solmuun x_{goal}. Algoritmia voidaan pitää laajennoksena Edsger Dijkstran vuonna 1959 esittelemään algoritmiin. Suurimpana erona voidaan pitää A*:n käyttämää heuristiikkafunktiota, jonka avulla se pyrkii suosimaan lähimpänä maalia olevia solmuja. Paras ensin hakualgoritmina se käyttää funktiota f määrittämään mikä solmu seuraavaksi käsitellään. Funktio $f(x)$ pyrkii arvioimaan todellisen $f^*(x)$:n arvon mahdollisimman tarkasti. Nämä kaksi funktiota määritellään seuraavasti: $f(x) = g(x) + h(x)$ sekä $f^*(x) = g^*(x) + h^*(x)$. Termi $g(x)$ on arvio todellisesta kustannuksesta lähtösolmusta x_{start} solmuun x. Termi $h(x)$ on vuorostaan arvio lyhimmästä reitistä solmusta x maalisolmuun x_{goal}. Tällöin $f(x)$ approksimoi kustannusta lähtösolmusta maalisolmuun jokaiselle solmulle x. A*-algoritmi on optimaalinen, eli löytää parhaan mahdollisen reitin, jos $h(x)$ on luvallinen (*admissible*). Tämä tarkoittaa sitä, että $h(x)$ ei koskaan yliarvioi kustannusta saavuttaa maalisolmu solmusta x, joten kaikilla solmuilla x pätee, että $h(x) \leq h^*(x)$. Jokaisella solmulla on joukko seuraajasolmuja x', jotka ovat ne solmut joihin solmusta x voidaan liikkua. Yleensä maksimimäärä seuraajasolmuja on 4 tai 8, riippuen voidaanko solmusta liikkua diagonaalisesti toiseen vai ei.

Muuttuja $g(x')$, jossa x' on solmun x seuraaja on hyvin helppo laskea. Se on x'-solmun etäisyys lähtösolmusta ja lasketaan aina lisäämällä solmun parhaan edeltäjän g-arvoon kustannus siirtyä edeltäjästä seuraajaan, $g(x') = g(x) + c(x, x')$. Jokaiselle solmulle, paitsi lähtösolmulle, lasketaan paras edeltäjä minimoimalla jokainen $g(x)$.

Muuttuja $h(x)$ on vuorostaan aina vain arvio (heuristiikka) etäisyydestä solmusta x maalisolmuun. Jos $h(x)$ voitaisiin aina laskea täysin tarkasti mitään hakua ei edes tarvittaisi sillä funktio h kertoisi aina suoraan kunkin solmun etäisyyden maalisolmusta. Siksi h funktiolla täytyy olla jokin menetelmä, jota se käyttää arvioimaan jokaisen solmun etäisyyttä maalista. Yksinkertaisin tapa on toteuttaa h niin että se antaa vain x-solmun suoran etäisyyden maalisolmusta. Tämä ei ole kuitenkaan käytännössä toimiva tapa. Heuristiikkafunktion valinnassa kannattaa huomioida ympäristön toteutus ja pyrkiä vastaavuuteen. Eri tapoja toteuttaa h on useita. Kuvassa 2.1 on kaksi erilaista tapaa toteuttaa h-funktio.

A*:n toteutukset käyttävät yleensä kahta listaa: avoin (*open*) sekä suljettu (*closed*). Avoin on yleensä toteutettu prioriteettijonona, josta pienimmän f-arvon omaava solmu otetaan aina käsittelyyn, kunnes reitti on löydetty. Avoin lista sisältää ne solmut, jotka ovat mahdollisesti vielä tulossa käsittelyyn, kun taas suljettu lista ne solmut, jotka on jo käyty läpi. Jokaisella iteraatiolla algoritmi poistaa avoimesta listasta sen solmun, jonka f-arvo on pienin, algoritmi arvio siis tämän solmun olevan sillä hetkellä lupaavin solmu, ja lisää sen suljettuun listaan. Kaikki tämän solmun naapurit (solmut, joihin tästä solmusta voi liikkua) jotka eivät ole jo suljetussa listassa lisätään avoimeen listaan niiden f-arvon mukaiseen kohtaan prioriteettijonossa. Näin jatketaan, kunnes maalisolmu tulee käsitellyksi eli se lisätään suljettuun listaan, tai kunnes avoin lista tulee kokonaan läpikäydyksi, tarkoittaen että reittiä lähtösolmusta maalisolmuun ei löytynyt. A*:n toiminta on esitetty

algoritmissa 1.

Algorithm 1: A*-algoritmi

$g(s_{\text{start}}) = 0$; AVOIN = $\emptyset$;
insert s_{start} into AVOIN with $f(s_{\text{start}}) = h(s_{\text{start}})$;
while s_{goal} *is not expanded* **do**
    remove s with the smallest f-value from AVOIN;
    for *each successor* s' *of* s **do**
        if s' *was not visited before* **then**
            f(s') = $g(s') = \infty$;
        end
        if $g(s') > g(s) + c(s, s')$ **then**
            $g(s') = g(s) + c(s, s')$;
            $f(s') = g(s') + h(s')$;
            insert s' into AVOIN with $f(s')$
        end
    end
end

Tämän lähtökohdan suurin ongelma on, että läpikäytyjen solmujen määrä saattaa olla eksponentiaalinen suhteessa optimaalisen reitin pituuteen. Solmujen määrä taas vaikuttaa suoraan avoimen listan kokoon. Näin ollen A*:n suoritusaika kasvaa huomattavasti, kun haettavan alueen kokoa kasvatetaan. A* ei myöskään osaa käyttää edellisiä hakuja hyväkseen vaan jokainen uusi A*-haku alkaa aina samasta alkutilanteesta. Lisäksi haku on suoritettava aina alusta loppuun ja algoritmi palauttaa aina joko optimaalisen ratkaisun tai ei ratkaisua ollenkaan. Algoritmilla ei myöskään ole any-angle-algoritmien ominaisuuksia, joka mahdollistaa suoremmat reitit koska liikkumista solmujen välillä ei ole rajattu kahdeksaan suuntaan. [23] [29]

Vaikka A* on hyvin laajalti käytetty, tietyissä tilanteissa se on osoittautunut riittämättömäksi. Tämän vuoksi A*:een on esitelty suuri määrä laajennoksia vuosien saatossa. Suuri osa näistä pyrkii ratkaisemaan tai korjaamaan tiettyjä A*:een liittyviä ongelmia ja puutteita. Siksi A*:n laajennoksia on jo vuosien ajan kategorisoitu sen mukaan mitä A*:n ongelmia tai puutteita kyseinen laajennos pyrkii korjaamaan. Vaikka kategorioita on löydettävissä kirjallisuudesta lukuisia, tässä tutkielmassa keskitytään neljään kategoriaan. Näitä ovat: *inkrementaalinen*, *anytime*, *reaaliaikainen* sekä *any-angle*. [23] [24]

	1	2	3	4	5	6
A	5	4	3	2	1	0 maali
B	6	5	4	3	2	1
C	7	6	5	4	3	2
D	8	7	6	5	4	3

(a) manhattan-etäisyys

	1	2	3	4	5	6
A	5	4	3	2	1	0 maali
B	5	4	3	2	1	1
C	5	4	3	2	2	2
D	5	4	3	3	3	3

(b) Chebyshev-etäisyys

Kuva 2.1: Kaksi erilaista tapaa toteuttaa h-funktio

2.1 Inkrementaaliset algoritmit

Tämän kategorian algoritmien perusajatuksena on, että peräkkäisiä keskenään hyvin samankaltaisia hakuja suoritetaan useita. Näin edellisessä haussa saatua informaatiota voidaan käyttää hyväksi seuraavissa hauissa. Tämä on erityisen hyödyllistä dynaamisessa ympäristössä, jossa solmujen tila voi muuttua hakujen välillä. Tietoa voidaan uudelleen käyttää pääasiassa kolmella eri tavalla.

- Ensimmäinen tapa on palauttaa A*:n avoin ja suljettu lista aiempaan tilaan, josta uusi haku voi erottautua omaksi haukseen. Siis A*:n aiempaa tilaa voidaan käyttää hyväksi ja jatkaa hakua siitä kohtaa eri tavalla mitä aiemmin on tehty. Ollakseen tehokkaampi kuin A*, A*:n tilan palauttamisen on oltava hyvin tehokasta. Esimerkkejä tällaisista algoritmeista ovat mm. iA* ja Fringe-Saving A* (FSA*)
- Toinen tapa on päivittää ja ylläpitää solmujen h-arvoja (heuristiikkoja). Jokaisen päivitysvaiheen jälkeen algoritmi takaa, että jokaisen solmun h-arvo on konsistentti ennen uuden haun aloittamista, joten jokaisesta solmusta tiedetään etäisyys maaliin ja voidaan taata, että optimaalinen reitti löytyy, jos sellainen on olemassa. Generalized Adaptive A* (GAA*) on esimerkki näin toimivasta algoritmista.
- Viimeinen tapa on ottaa edellisen haun hakupuu suoraan uuden haun hakupuuksi. Jotta haku olisi tässä tilanteessa nopeampaa kuin pelkkä A*-haku, hakujen on oltava hyvin samankaltaisia keskenään, eli ympäristön muutosten tulisi vaikuttaa vain pieneen osaan hakupuuta. Jos muutoksia tapahtuu hyvin lähellä hakupuun juurta nämä algoritmit eivät yleensä ole kovin tehokkaita, sillä mahdollisuus, että hakupuut ovat samankaltaisia keskenään on hyvin pieni. Lifelong Planning A* (LPA*), D* ja D* Lite ovat esimerkkejä tämänkaltaisista A*-laajennoksista.

Erityisesti viimeiseen kategoriaan kuuluvia algoritmeja on käytetty laajalti erityisesti ro-

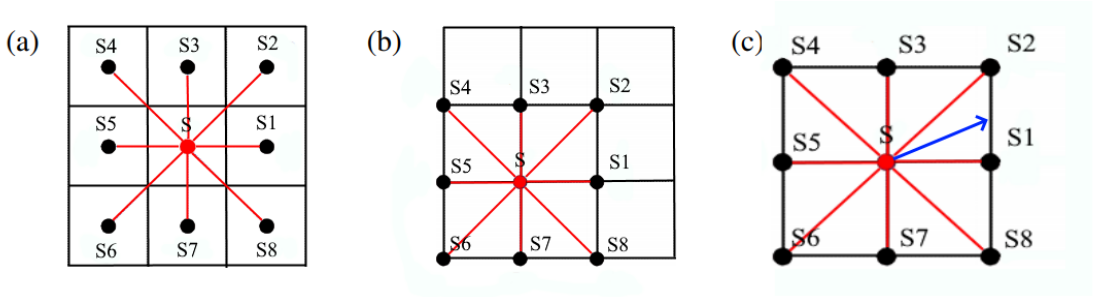
botiikassa ja varsinkin D* Lite kuuluu suosituimpiin algoritmeihin. A*:n tavoin myös D* Lite määrittää jokaiselle solmulle arvon $g(x)$ sekä lisäksi jokaiselle solmulle määritetään ns. ennakointiarvo (lookahead) joka määritetään sen edeltäjien mukaan: $rhs(x) = g(x') + cost(x', x)$. Termi $rhs(x)$ on siis määritelty sen edeltäjän g-arvon mukaan, johon lisätään kustannus siirtyä edeltäjästä solmuun x. Solmuja käsitellään eri tavoin sen mukaan ovatko g ja rhs samat, rhs suurempi vai g suurempi. Solmut, joiden g ja rhs ovat toisistaan poikkeavat sanotaan olevan epäkonsistenttejä ja näin ollen ne tulisi lisätä avoimeen listaan käsiteltäväksi. D* Liten toimintaperiaate on hyvin samankaltainen A*:een verrattuna. Avoin lista sisältää käsittelemättä olevat solmut, joista lupaavimpana pidetty solmu otetaan käsiteltäväksi ensin. Suurimpana erona on rhs-arvon lisääminen sekä haun suorittaminen lopusta alkuun. Näin ollen algoritmin g-arvot pysyvät konsistenttinä, vaikka seuraava haku suoritettaisiinkin eri alkupisteestä. Hakua suoritetaan niin kauan, kunnes yhtään epäkonsistenttiä solmua ei löydy, joka tarkoittaa, että reittiä maaliin ei löytynyt tai s_{start} tulee konsistentiksi, jolloin reitti löytyi. Jos jokin reitillä muuttuu, osa solmuista tulee jälleen epäkonsistenteiksi ja suoritus jatkuu, kunnes uusi reitti löytyy. D* Lite-algoritmiin palataan tarkemmin luvussa 3.

Toinen inkrementaalinen hakualgoritmi, joka on myös any-angle-algoritmi, on Field D*. Field D* perustuu D* Liteen ja myös sitä on käytetty robotiikan alalla. Field D* kuten muut any-angle-algoritmit eivät rajoita robotin liikettä vain kahdeksaan mahdolliseen suuntaan vaan se tuottaa muihin verrattuna suorempia reittejä, jotka voivat poistua solmusta mistä kohtaa tahansa.[23] [26]

2.2 Any-angle-algoritmit

Reitinhakujen ympäristöä mallinnetaan usein ruudukkona tai koordinaatistona, joissa solmun ympärillä olevat kahdeksan naapuria ovat ne solmut, joihin kyseisestä solmusta voidaan liikkua. Jokainen solmu voi olla vapaa tai estetty sekä joissain mallinnuksissa sille on voitu antaa jokin arvo c, joka kuvaa miten vaikeaa kyseiseen solmuun liikkuminen on. Suurin osa algoritmeista on tällaisia, joten ne rajoittavat liikkumista ruudukossa $\pi/4$ askeleisiin. Any-angle-algoritmit eivät rajoita liikkumista solmujen välillä lainkaan.

A*:n toimintaympäristö esitetään yleensä ruudukkona, jossa jokaisella solmulla on 8 ”seuraajaa” tai ”naapuria”. Tämä solmujen välinen suhde määrittää sen, miten robotti voi siirtyä eri solmujen välillä ja toisaalta myös sen millaisia reittejä algoritmi kykenee tuottamaan. Koska mahdollisten siirtymien määrä on hyvin rajattu, myös mahdollisten



Kuva 2.2: (a) Yleisesti käytetty ruudukko, jossa solmut sijaitsevat ruutujen sisällä. Kaaret keskisolmun sisältä edustavat kaikkia niitä siirtymiä joita robotti voi tehdä tästä solmusta. (b) Field D*:n käyttämä muokattu versio, jossa solmut sijaitsevat ruudukon reunalla. (c) Optimaalisen reitin on kuljettava jonkin reunan $\{\overrightarrow{s_1s_2}, \overrightarrow{s_2s_3}, \overrightarrow{s_3s_4}, \overrightarrow{s_4s_5}, \overrightarrow{s_5s_6}, \overrightarrow{s_6s_7}, \overrightarrow{s_7s_8}, \overrightarrow{s_8s_1}\}$ läpi.

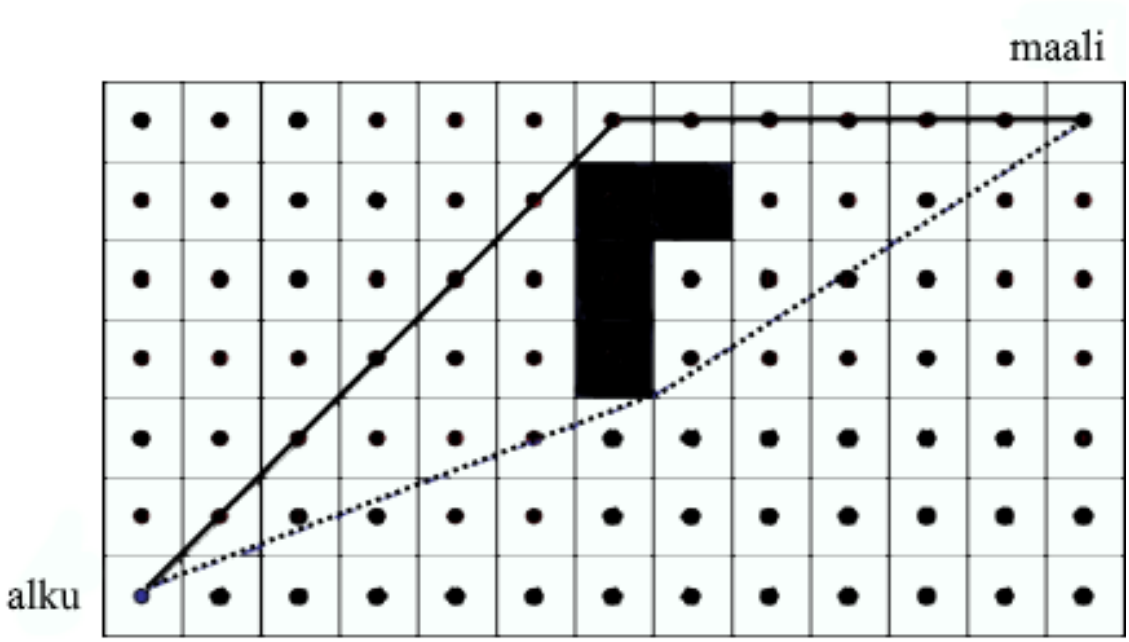
reittien määrä on näin rajattu, vaikka kartan koon mukaan voikin olla hyvin suuri. A* sekä muut A*-laajennokset valitsevat reittiä laskiessaan näiden kahdeksan naapurisolmun joukosta aina parhaan solmun, johon liikkua. Näin ruudukossa tapahtuva liikkuminen on rajattu kuitenkin $\pi/4$ askeliin, jolloin lasketun reitin pituus voi olla epäoptimaalinen tai sisältää turhaa kääntymistä. Tosinaan on mahdollista suorentaa reittiä jälkikäsittelyssä, jolloin estevapaiden pisteiden väleille piirretään suora jana, joka lyhentää reittiä. Tämä ei ole kuitenkaan aina mahdollista pisteiden välissä olevien esteiden takia (kuva 2.3).

Field D* on algoritmi, joka luo reittejä, joissa solujen välisiä siirtymiä ei ole rajattu. Field D* käyttää apunaan lineaarista interpolointia, jonka avulla se laskee solujen välisille pisteille arvion. Field D*-algoritmiin palataan tarkemmin luvussa 3. [23] [24]

2.3 Anytime-algoritmit

Algoritmi kuuluu tähän kategoriaan, jos se kykenee antamaan toteuttamiskelpoisen arvion reitistä milloin tahansa (paitsi ensimmäisen laskennan aikana). Algoritmin oletetaan kykenevän myös parantamaan reittiä mitä enemmän laskentaan annetaan aikaa. Riittävän ajan kuluttua ratkaisun laatu ei enää parane koska paras mahdollinen reitti on löydetty. Näin ollen anytime-algoritmit ovat kompromissi suoritusajan ja ratkaisun laadun suhteen. Useita erityyppisiä A*:een perustuvia anytime-algoritmeja on esitetty.

Eräs hyvin yleinen toteutusmetodi on karsia avointa listaa ensimmäisen suorituksen jälkeen. Ensimmäinen suoritus antaa näin ylärajan, jota huonommat ratkaisut voidaan suoraan karsia pois listasta. Anytime Weighted A* (AWA*) on tällainen algoritmi. Suurimpana erona A*:een voidaan pitää sitä, että algoritmin heuristiikkafunktio h ei ole luvalli-



Kuva 2.3: 2D ruudukkojen reittejä ei aina kyetä lyhentämään jälkikäsittelyssä, sillä esteet saattavat olla tiellä. Kuvassa yhtenäisellä janalla esitetty lyhin reitti rajatulla metodilla laskettuna sekä katkoviivalla esitetty optimaalinen reitti.

nen (*admissible*), eikä näin ollen voi taata optimaalista reittiä. Epäluvallinen heuristiikka toteutetaan kertomalla $h(x)$ muuttujalla w, jossa w on käyttäjän määrittämä. Muuttujan suuruusluokka määrittää kompromissin ajan ja laadun välillä. A*:n tavoin AWA* noutaa avoimesta listasta solmuja f-arvon mukaisessa järjestyksessä. Avoimesta listasta voidaan karsia pois kaikki solmut, joiden f-arvo on korkeampi kuin tähän mennessä löydetyn ratkaisun f.

Toinen paljon käytössä oleva anytime-algoritmi on Anytime Repairing A* (ARA*). Se on käytössä sekä robotin käden liikkeen mallintamisessa että robotin liikkumisen mallintamisessa ulkoilmassa. ARA* algoritmia käsitellään tarkemmin luvussa 3. [23] [28]

2.4 Reaaliaika-algoritmit

Reaaliaika-algoritmien yhteinen piirre on, että ne kykenevät toimimaan hyvin tarkkojen aikarajojen puitteissa. Toisaalta koska niiden hakuavaruus on rajattu (ennakointi, lookahead) ne eivät koskaan voi taata optimaalisen reitin löytämistä. Tyypillisesti algoritmit vuorottelevat nopeiden suoritusten ja suunnitteluiden välillä, ja niitä käytetään tapauksissa, joissa nopea toteuttaminen on tärkeämpää kuin ratkaisun laatu. Erityisesti tietokonepe-

lit ovat hyvä käyttökohde reaaliaika-algoritmeille niiden interaktiivisen luonteen vuoksi. Toinen mahdollinen käyttökohde on robotiikan ala.

Learning Real-Time A* (LRTA*) on yksi ensimmäisistä reaaliaika-algoritmeista ja sen jalanjäljissä on kehitetty myös useita muita sen jälkeen. Real-Time Adaptive A* (RTAA*) on käytössä esim. tietokonepeleissä ja erityisesti RTS-peleissä (*real time strategy*). RTAA* käyttää A*:ä aliohjelmana ja rajoittaa sen suoritusta tiettyyn arvoon (*lookahead*). Suorituksen jälkeen algoritmi päivittää kaikkien solmujen h-arvot avoimen listan parhaan solmun g-arvon mukaan.

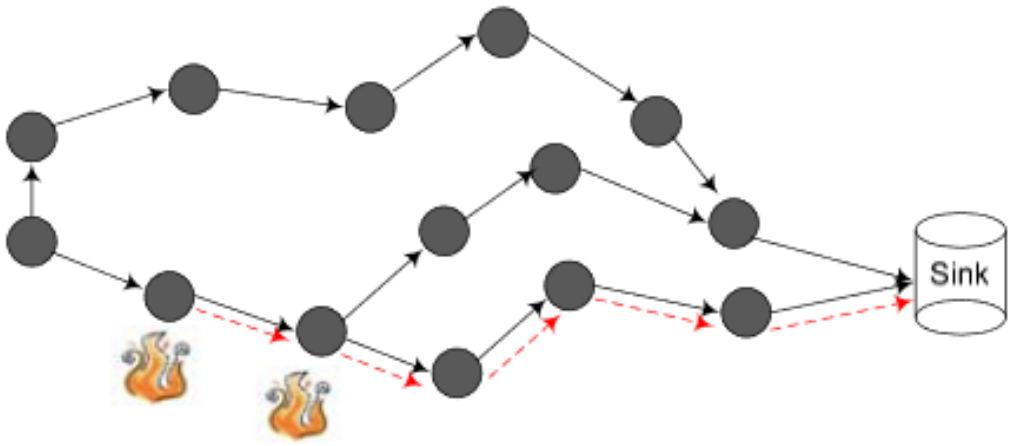
Local Search Space LRTA* (= LSS-LRTA*) on versio Learning Real-Time A*:stä, joka käyttää alkuperäistä A*:ä määrittämään paikallisen hakuavaruuden sekä Dijkstran algoritmia päivittämään h-arvot. Algoritmi on hyvä esimerkki, miten reaaliaika-algoritmien suorituksessa vuorottelevat suunnittelu ja toteutus. LSS-LRTA*-algoritmia käsitellään tarkemmin luvussa 3. [23] [25]

2.5 A*-algoritmin soveltamisalueita

Koska A*:n puutteita on enenemissä määrin pyritty korjaamaan kehittämällä variantteja, jotka korjaavat sen ongelmia, monilla sovellusalueilla A* ei ole varsinkaan enää optimaalinen valinta. A*:ä voitaisiin käyttää esimerkiksi sosiaalisen median analysointiin, sillä sitä voidaan kuvata graafin avulla, mutta sen muistinkulutus osoittautuu liian suureksi. [17] Myös artikkelissa [18] esitetyt menetelmät, jotka tehostavat robotin reitinhakua sekä langattomien verkkojen energiatehokkuutta ovat A*-variantteja.

A*:ä on käytetty langattomien sensoriverkkojen energiatehokkuuden lisäämiseen. On tärkeää löytää mahdollisimman lyhyt matka sensorien välille ja näin säästää energiaa. Langattomat sensoriverkot (*Wireless sensor networks*) ovat joukko sensoreita, jotka monitoroivat ja keräävät dataa keskitettyyn sijaintiin. Verkot mittaavat ympäristöstään esimerkiksi lämpötilaa, kosteutta tai ilman laatua. Verkkojen suurimpana ongelmana on ollut taakan jakautuminen tasaisesti. Erityisesti lähellä dataa keräävää laitetta olevat sensorit kuluttavat enemmän energiaa, joten taakka ei jakaudu tasaisesti sensoreiden kesken. Kun otetaan huomioon neljä seikkaa, saadaan verkosta paremmin toimiva ja energiatehokkaampi.

- Energian kulutuksen tasaaminen
- Kuormituksen tasaaminen
- Lyhimpien reittien laskenta



Kuva 2.4: Sensorit havaitsevat tulipalon ja lähettävät siitä tiedon lyhintä reittiä pitkin. [19]

- Pakettien uudelleenlähettämisen tarpeen vähentäminen

A*-algoritmia voidaan käyttää kolmannessa kohdassa (kuva 2.4) laskemaan lyhimpiä reittejä sensoreiden välille. [19] [20]

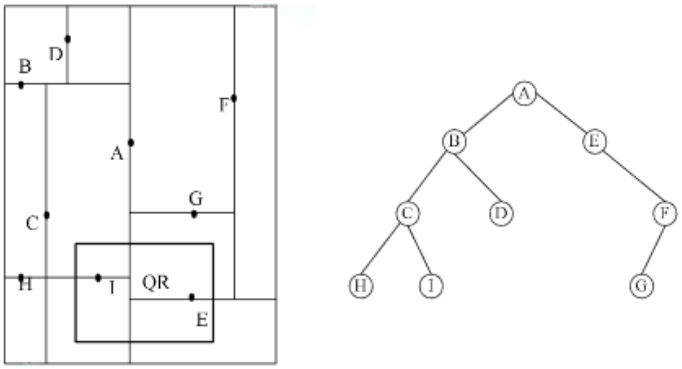
A* on ollut käytössä myös reittien suunnittelussa. Nykyään hyvin suositut navigaattorit ja karttasovellukset tarjoavat käyttäjille nopeasti optimaalisia reittejä sekä kykenevät korjaamaan reittejä, jos tarve tulee. A* kykenee tähän mutta tähänkin tapaukseen on kehitetty siitä paremmin selviytyviä variantteja. [21] A*:n muistiongelman ratkaisuna vain haun kannalta tarvittavat osat graafista ladataan muistiin. Tämä toteutetaan indeksoimalla graafin dataa. Graafi jaetaan alueisiin ja osa graafista ladataan muistiin vasta kun on osoitettu, että sitä tarvitaan haun suorittamiseen. Testauksessa käy ilmi, että variantin muistinkulutus on merkittävästi pienempää kuin A*:n. Uuden A*:n toiminta poikkeaa hieman alkuperäisestä.[16]

- Alustus. Ladataan aloitussolmu avoimeen listaan. Suljettu lista on tyhjä.
- Käsittele solmu, jonka $f(j)$ on pienin.
- Jos solmu j:n alue on jo ladattu, jatketaan normaalisti. Jos ei, ladataan uusi alue muistiin ja jatketaan hakua.
- Kun maalisolmu löytyy, lasketaan lyhin reitti.

Kuvassa 2.5 on verrattu A*-variantin sekä alkuperäisen A*:n laskemia reittejä. Kuvassa 2.6 on vuorostaan kuvattu tapa, miten graafin voi jakaa alueisiin ja muodostaa tästä hakupuun, jonka avulla hakuja voidaan suorittaa nopeasti.

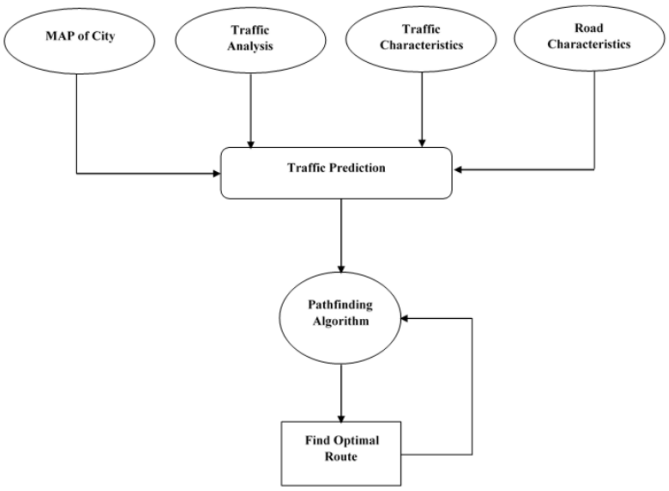
(a) A* (b) A*-variantti

Kuva 2.5: A*:n sekä sen muunnoksen laskemat reitit kartalla. [16]



Kuva 2.6: Tapa indeksoida graafin dataa. [16]

Yksi sovellusalue voisi olla myös älykaupunki. Älykaupunki kerää sensorien avulla erilaista dataa kaupungin sen hetkisestä tilasta ja sen perusteella voidaan tehdä parempia päätöksiä ympäri kaupunkia. Yksi mahdollinen sovelluskohde voisi olla ruuhkien välttäminen. Älykaupungilla voisi olla tietoa kyseisen hetken ruuhkista ja käyttämällä A*:ä tai sen variantteja se voisi laskea parhaan reitin sitä pyytävälle autoilijalle. Kuvassa 2.7 on esitetty miten tällainen järjestelmä voisi toimia. [22]



Kuva 2.7: Kuvaaja liikennettä ennakoivasta järjestelmästä. A*:n tehtävänä on toimia reitinhakijana kohdassa *Pathfinding Algorithm*. [22]

A* kehitettiin alun perin itsenäiseen päätöksentekoon kykenevän robotin tarpeisiin, joka havainnoi ympäristöään ja kykenee laskemaan reittejä sen nykyisestä sijainnista sen määränpäähän. Tässä tutkielmassa tutkitaan A*:ä ja sen variantteja sen alkuperäistä kontekstia muistuttavassa ympäristössä. Robotin on kyettävä navigoimaan tuntemattomassa ympäristössä määrättyyn loppupisteeseen sen tekemien havaintojen perusteella. Tulokset antavat viitteitä siitä miten paljon algoritmien toiminta poikkeaa toisistaan tällaisessa ympäristössä sekä siitä miten paljon tehokkaampia variantit ovat suhteessa alkuperäiseen algoritmiin. Tuloksia voidaan käyttää hyväksi myös esim. peleissä, sillä monet pelit toimivat hyvin samankaltaisissa ympäristöissä.

Luku 3

Algoritmit

Edellä on esitetty neljä erilaista luokkaa tai kategoriaa joihin A*:n muunnoksia on jaettu. Tässä luvussa esitellään tarkemmin jokaiseen luokkaan kuuluva algoritmi.

3.1 D* Lite

D* Lite on inkrementaalinen A*-laajennos. D* Lite käyttää edellisen hakunsa hakupuuta hyväkseen uusissa hauissa sekä heuristiikkoja hakujen keskittämiseen. Näin se löytää huomattavasti nopeammin ratkaisut useisiin peräkkäisiin samankaltaisiin hakuihin, kuin olisi mahdollista, jos jokainen haku suoritettaisiin aina kokonaan alusta loppuun. Näin pienet muutokset ympäristössä johtavat hyvin nopeisiin hakuihin, jotka käyttävät hyväkseen edellistä hakua. Toisaalta suuret äkilliset muutokset tai muutokset hakupuun juuressa hidastavat hakua huomattavasti. D* Lite ei hyödynnä A*:stä tuttua suljettua listaa lainkaan mutta A*:n avoimen listan kaltainen jono, josta algoritmi ottaa solmuja käsittelyyn niiden prioriteettien mukaisesti on tärkeä osa D* Liten suoritusta. Tässä yhteydessä tietorakenne on kuitenkin nimetty vain prioriteettijonoksi tai U:ksi. Solmut järjestetään jonossa avainten mukaan, jotka on laskettu funktiolla: $k(s) = [min(g(s), g(rhs)) + h(s, s_{\text{goal}}); min(g(s), g(rhs))]$. Avaimen toista komponenttia käytetään ratkaisemaan tasatilanteet.

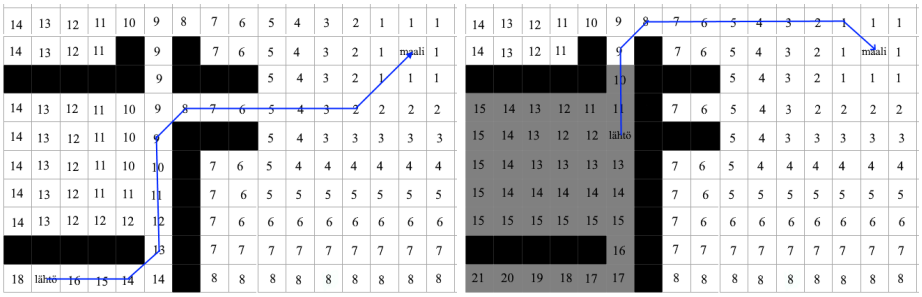
D* Lite on kehitetty erityisesti robotin navigointitehtäviin. Tuntemattomassa ympäristössä liikkuva robotti, joka voi liikkua ympäristönsä johonkin kahdeksasta solusta, jos ne ovat vapaita, havainnoi ympäristöä ja havaittuaan esteen se laskee reitin uudelleen. Robotin tehtävänä on liikkua lähtösolmusta maalisolmuun. Robotti olettaa, että kaikki solmut, joita se ei ole havainnut ja joiden tilannetta se ei tiedä, ovat vapaita. Jos robotti havain-

noi ympäristöään vain pieneltä alueelta voidaan olettaa, että hakujen välillä ei tapahdu kovin suuria muutoksia sillä robotti tuskin voi löytää kovin suurta määrää esteitä kerralla. Toisaalta jos robotti havainnoi ympäristöään kovin laajalta alueelta esteitä voi löytyä useampia. On myös mahdollista, että muutos hakupuun juuressa hidastaa hakua ja pakottaa suuren määrän muutoksia kerralla.

D* Liten toiminta poikkeaa A*:stä myös siinä, että se suorittaa jokaisen uuden haun aina maalisolmusta lähtösolmuun, joten sen g-arvot ovat arvioita maalisolmun etäisyydestä. D* Lite ei aseta mitään erityisvaatimuksia toimintaympäristöään kohtaan, vaan sen on ainoastaan kyettävä määrittämään solmujen seuraajat ja edeltäjät. Algoritmi kykenee näin aina suorituksen jälkeen löytämään lyhimmän reitin lähtösolmusta maalisolmuun liikkumalla aina nykyisestä solmusta s, aloittaen aina lähtösolmusta, aina seuraajaan s' joka minimoi funktion $g(s')+c(s', s)$. Muuttuja c on s solmusta s' solmuun siirtymisen kustannus ja $g(s')$ seuraajan etäisyys maalisolmusta. Haku päättyy, kun maalisolmu on löydetty.

Kyetäkseen liikkumaan itsenäisesti tuntemattomassa ympäristössä algoritmin on kyettävä liikuttaman robottia edellisessä kohdassa laskettua reittiä pitkin. Aina kun robotti havainnoi esteen sen reitillä se laskee uuden reitin päivitettyjen tietojen mukaisesti. Ongelmaksi koituu kuitenkin se, että robotin liikuttua sen prioriteettijono sisältää solmuja, joiden avain ja erityisesti sen heuristiikka on laskettu robotin edellisen sijainnin mukaan. Näin ollen prioriteettijonon avaimet tulisi aina laskea uudelleen robotin sijainnin muuttuessa. Erityisesti avaimen ensimmäinen komponentti, joka sisältää heuristiikan. Robotin liikkuessa kohti maalisolmua ja reitin uudelleenlaskennan tapahtuessa avaimen ensimmäisen komponentin voidaan sanoa pienentyneen korkeintaan $h(s, s')$ verran joka on arvio siitä etäisyydestä jonka robotti liikkui laskentojen välillä. Avainten ensimmäiset komponentit ovat siis juuri tämän verran liian suuria. Tämän arvon poistaminen avainten ensimmäisestä komponentista johtaisi kuitenkin liian pieniin avainten arvoihin joten $h(s, s')$ on sen sijaan lisättävä prioriteettijonossa olevien avainten ensimmäiseen komponenttiin kun robotti löytää muutoksen solmuissa ja solmujen kustannukset muuttuvat. Robotin liikkuessa uudestaan ja löytäessä uusia muutoksia graafissa näiden vakioiden arvot on laskettava yhteen. Tätä muuttujaa k_{m} kasvatetaan aina arvolla $h(s_{\mathrm{last}}, s_{\mathrm{start}})$, joka on arvio siitä etäisyydestä jonka robotti liikkui kustannusten muutosten välillä. Tämä arvo lisätään avainten ensimmäiseen komponenttiin, joten kaikkia prioriteettijonon avaimia ei tarvitse laskea uudelleen.

D* Liten toimintaa on havainnollistettu kuvassa 3.1. Robotti liikkuu laskemaansa reittiä pitkin, kunnes se havaitsee muutoksen ympäristössä ja joutuu sen vuoksi muuttamaan reittiään. Laskentaa ei tarvitse kuitenkaan suorittaa koko reitin mitalta, vaan vain harmaalla



(a) Tieto alueesta aloitushetkellä. (b) Robotti havaitsee esteen.

Kuva 3.1: Yksinkertainen D* Lite esimerkki. Robotti havaitsee esteen solussa (14,8). Harmaiden solujen g-arvot muuttuvat.

merkittyjen solmujen arvot muuttuvat. D* Liten ensimmäinen suoritus on aina täsmälleen samanlainen kuin vastaavan A*-haun, mutta tulevat haut käyttävät aina hyväkseen edellistä hakua. Hakujen nopeus riippuu siitä, miten toisiaan vastaavia perättäiset haut ovat, eli käytännössä suuret muutokset varsinkin hakupuun juuressa (lähempänä maalia) hidastavat uusia hakuja enemmän mitä pienet muutokset lähellä lähtöpistettä. D* Liten toiminta kokonaisuudessaan on esitetty kuvassa 3.2.

D* Lite poikkeaa A*:stä siinä, että se ylläpitää kahta erillistä muuttujaa solmujen etäisyyksistä. Kun A* ylläpitää solmujen etäisyyksiä muuttujassa g(s) joka on solmun s etäisyys lähtösolmusta, D* Lite ylläpitää kahta muuttujaa $g(s)$ sekä $rhs(s)$, molemmat solmun s etäisyyksiä maalisolmusta, sillä D* Lite suorittaa hakunsa maalisolmusta. Muuttuja $rhs(s)$ perustuu vuorostaan sen edeltäjään ja lasketaan kaavassa 3.1 esitetyllä tavalla.

$$rhs(s) = \begin{cases} 0 & jos\ s = s_{\text{start}} \\ min_{s' \in\ Pred(s)}(g(s') + c(s', s)) & muutoin. \end{cases} \tag{3.1}$$

Näin ollen se on mahdollisesti paremmin informoitu, kun $g(s)$. D* Lite käyttää näitä arvoja propagoimaan muutoksia aina seuraaviin solmuihin, kun solmujen arvot muuttuvat. Esim. kun robotti liikkuessaan löytää uuden esteen kaikki ne solmut, joiden g-arvo on laskettu sen mukaan, että solmu s olisi vapaa joutuvat korjaamaan g-arvonsa. Koska $rhs(s)$ lasketaan edeltäjäsolmun g-arvon mukaan, muutokset edeltäjäsolmun g-arvossa propagoituvat seuraajasolmun s rhs-arvoon. Näin solmun $rhs(s)$ sekä $g(s)$ poikkeavat tämän seurauksena toisistaan.

Kun solmun rhs- sekä g-arvot poikkeavat toisistaan sanotaan, että solmu on paikallisesti epäkonsistentti. Jos kaikki solmut ovat paikallisesti konsistenttejä, mistä tahansa solmusta voidaan löytää reitti mihin tahansa toiseen solmuun. Paikallisesti epäkonsistentit

Procedure CalculateKey(s)

return [$min(g(s), rhs(s)) + h(s_{\text{start}}, s) + k_{\text{m}}$; $min(g(s), rhs(s))$];

Procedure Initialize()

$U = \emptyset$;
$k_{\text{m}} = 0$;
for all $s \in S$ $rhs(s) = g(s) = \infty$;
$rhs(s_{\text{goal}}) = 0$;
U.Insert(s_{goal}, CalculateKey(s_{goal}));

Procedure UpdateVertex(u)

if $u \neq s_{goal}$ **then**
    $rhs(u)$ = $\min_{s' \in Succ(u)}$ ($c(u,s')$ + $g(s')$);
if $u \in U$ **then**
    U.Remove(u);
if $g(u) \neq rhs(u)$ **then**
    U.Insert(u, CalculateKey(u));

Procedure ComputeShortestPath()

while *U.TopKey()* < *CalculateKey*(s_{start}) *OR* $rhs(s_{start}) \neq g(s_{start})$ **do**
    k_{old} = U.TopKey();
    u = U.Pop();
    if $k_{old} <$ *CalculateKey*(u) **then**
        U.Insert(u, CalculateKey(u));
    else if $g(u) > rhs(u)$ **then**
        $g(u) = rhs(u)$;
        for all $s \in$ Pred(u) UpdateVertex(s);
    else
        $g(u) = \infty$;
        for all $s \in$ Pred(u) $\cup$ {u} UpdateVertex(s);

Procedure Main()

s_{last} = s_{start};
Initialize();
ComputeShortestPath();
while $s_{start} \neq s_{goal}$ **do**
    s_{start} = arg $\min_{s' \in Succ(s_{start})}$ ($c(s_{\text{start}},s')$ + $g(s')$);
    Move to s_{start};
    Scan graph for changed edge costs;
    if *any edge costs changed* **then**
        $k_{\text{m}} = k_{\text{m}} + h(s_{\text{last}}, s_{\text{start}})$;
        s_{last} = s_{start};
        for *all directed edges (u,v) with changed edge costs* **do**
            Update the edge cost $c(u, v)$;
            UpdateVertex(u);
        ComputeShortestPath();

Kuva 3.2: D* Lite-algoritmi

solmut on käsiteltävä konsistenteiksi, jos niiden kautta halutaan määrittää reittejä. Reittejä konsistenttien solmujen läpi voidaan määrittää, vaikka kaikki solmut eivät olisikaan konsistenttejä. Muutokset solmun g-arvossa propagoivat muutokset aina seuraajien rhs-arvoon, jolloin muutokset propagoituvat aina seuraajiin, kunnes kaikki olennaiset solmut ovat konsistenttejä. [26] [27]

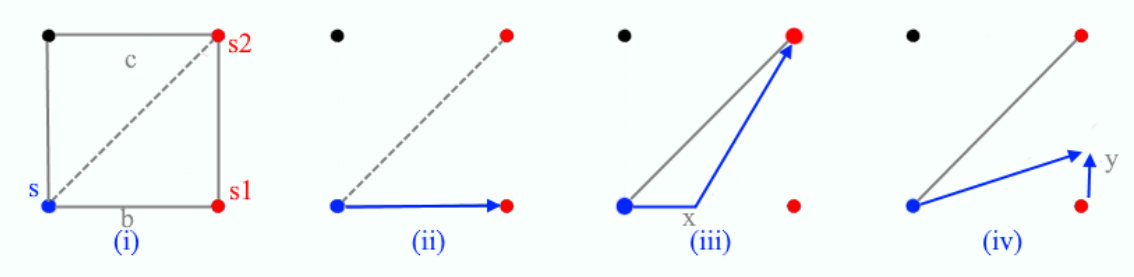
- Solmun sanotaan olevan alikonsistentti, jos: $g(s) < rhs(s)$
- Tämä tarkoittaa käytännössä, että tähän solmuun siirtyminen tuli kalliimmaksi. Esim. seinä ilmestyi katkaisemaan kulkua solmuun.
- Tällaisissa tapauksissa solmu asetetaan alkutilaan (sen $g(x)$ arvoksi asetetaan ∞), jolloin siitä tulee konsistentti tai ylikonsistentti.
- Solmun sanotaan olevan ylikonsistentti, jos: $g(s) > rhs(s)$
- Tämä tarkoittaa käytännössä, että tähän solmuun siirtyminen halpeni. Esim. este poistui reitin tieltä.

3.2 Field D*

Field D* on hyvin vahvasti D* Liteen perustuva any-angle-algoritmi, joka on kehitetty tarkoituksena korjata A*:n ja D* Liten ominaisuus, jossa niiden solmujen välillä tapahtuvat siirtymät tapahtuvat yleensä $^{\pi}/_{4}$ välein. Tämä taas puolestaan pidentää laskettujen reittien pituutta sekä vaikeuttaa liikkumista käytännössä. Field D* käyttää apunaan lineaarista interpolointia arvioimaan g-arvoja solmujen (node) välille.

Field D* määrittää jokaiselle solulle (*cell*), jossa solu on neljän solmun rajoittama alue, kustannuksen, joka kertoo, miten suuri kustannus kyseisen solun läpi liikkuminen on. Yksinkertaisimmillaan kustannus (*cost*) on yksi, jolloin laskennasta tulee yksinkertaisempaa. Field D* mahdollistaa kuitenkin mielivaltaiset c:n arvot eri soluille. Tämä ominaisuus voidaan vähällä vaivalla lisätä myös muihin tässä tutkielmassa esiteltyihin algoritmeihin mutta sitä ei käsitellä tässä työssä tarkemmin. Field D* laskee reittiensä kustannuksen täysin samalla periaatteella kuin klassiset $^{\pi}/_{4}$ välejä käyttävät algoritmit, käyttäen jokaisen solmun naapurisolmuja. Yleisesti tämä lasketaan kaavalla 3.2, jossa *nbrs(s)* ovat kaikki solmun s naapurisolmut:

$$g(s) = min_{s' \in \textit{nbrs(s)}}(c(s, s') + g(s')), \tag{3.2}$$



Kuva 3.3: Solmun s lyhimmän reitin laskeminen pisteeseen s_y solmujen s_1 ja s_2 välissä sen kahden naapurisolun avulla. Muuttuja c on keskisolun kustannus ja b alemman solun kustannus.

if s was not visited before, $g(s) = \infty$;
if $(s \neq s_{goal})$
  $rhs(s) = min_{(s',s'') \in connbrs(s)}$ ComputeCost(s, s', s'');
if $(s \in$ *OPEN*$)$ remove s from *OPEN*;
if $(g(s) \neq rhs(s))$ insert s into *OPEN* with key(s);

Kuva 3.4: Field D*:n UpdateState-funktio. Punaisella merkitty osa on algoritmin suurin ero D* Liteen verrattuna.

Tämä laskenta olettaa, että ainoat siirtymät solmusta s ovat kahdeksan suoraa linjaa sen naapurisolmuihin. Jos ehtoa lievennetään ja siirtymä sallitaan mihin tahansa pisteeseen solujen reunalla, tämä mahdollistaa kaikki mahdolliset siirtymät solmujen välillä. Jos olisi mahdollista tietää jokaisen pisteen s_b arvo tällä reunalla voitaisiin solmun s arvo laskea minimoimalla: $c(s, s_b) + g(s_b)$. Valitettavasti pisteitä s_b on ääretön määrä, joten arvon laskeminen jokaiselle pisteelle on mahdotonta. On kuitenkin mahdollista laskea arvio jokaiselle pisteelle s_b käyttäen lineaarista interpolointia. Ensin koordinaatistoon tehdään kuvassa 2.2 tehty muutos. Muuttunut koordinaatisto mahdollistaa sen, että jokaiselle reunan pisteelle voidaan laskea myös arvo interpoloimalla. Muutoksen jälkeen jokaista solmua kohdellaan näytepisteenä jatkuvassa kentässä kustannuksia.

Jokaisen solmusta s lähtevän reitin on kuljettava jonkun reunan $\overrightarrow{s_a s_b}$ läpi kuten kuvassa 2.2, joten reitti on siis minimikustannus solmusta s johonkin pisteeseen näistä reunoista. Jokaisen pisteen arvo lasketaan interpoloinnilla: $g(s_y) = yg(s_2) + (1 - y)g(s_1)$, jossa y on s_1:n ja s_y:n etäisyys. Pisteen s_y kustannus ei välttämättä ole $g(s_1)$ ja $g(s_1)$ lineaarinen kombinaatio, mutta sen on osoitettu toimivan käytännössä hyvin. Kun mukaan otetaan solmun sisällä tapahtuva siirtymäkustannus, saadaan koko kustannukseksi 3.3:

$$\min_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + g(s_2)y + g(s_1)(1-y)], \tag{3.3}$$

Procedure ComputeCost(s, s_a, s_b)

if *s_a is a diagonal neighbor of s* **then**
    s_1 = s_b; s_2 = s_a;
else
    s_1 = s_a; s_2 = s_b;
c is traversal cost of cell with corners s, s_1, s_2;
b is traversal cost of cell with corners s, s_1 but not s_2;
if $min(c, b) = \infty$ **then**
    $v_\text{s} = \infty$
else if $g(s1) \leq g(s2)$ **then**
    $v_\text{s} = min(c, b) + g(s_1)$;
else
    $f = g(s_1) - g(s_2)$;
    if $f \leq b$ **then**
        if $c \leq f$ **then**
            $v_\text{s} = c\sqrt{2} + g(s_2)$;
        else
            $y = min(\frac{f}{\sqrt{c^2 - f^2}}, 1)$;
            $v_\text{s} = c\sqrt{1 + y^2} + f(1 - y) + g(s_2)$;
    else
        if $f \leq b$ **then**
            $v_\text{s} = c\sqrt{2} + g(s_2)$;
        else
            $x = 1 - min(\frac{b}{\sqrt{c^2 - b^2}}, 1)$;
            $v_\text{s} = c\sqrt{1 + (1 - x)^2} + bx + g(s_2)$;

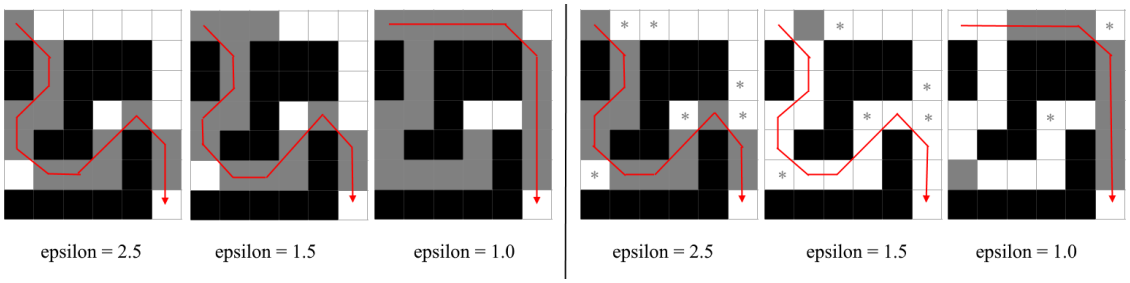
Kuva 3.5: Field D* -algoritmin ComputeCost()-funktio

Muuttuja x on etäisyys, jonka robotti liikkuu solun alaosassa ja käyttää näin solun b kustannusta laskemaan koko liikkumisen kustannuksen, kunnes se siirtyy liikkumaan kokonaisuudessaan c solun sisään ja näin sen kustannus lasketaan c solun mukaan. Jos sekä x ja y ovat nolla tällöin liikkuminen tapahtuu c ja b solujen välissä kohti solmua s_1 mutta kustannus lasketaan c solun mukaan. Mahdolliset reitit on esitelty kuvassa 3.3. Muuttuja x voi olla myös nolla, jolloin reitti kulkee suoraan solmua s_2 kohti. Lyhin reitti riippuu muuttujista c, b sekä f, jossa f on $g(s_1) - g(s_2)$. Field D*-algoritmi ottaa lähtökohdakseen edellä esitellyn D* Liten ja liittää edellä esitellyn tavan interpoloida arvoja solmujen välille. UpdateState-funktion toiminta on kuvattu kuvassa 3.4. Punaisella on merkitty erot D* Lite Algoritmiin. Field D*:n ja D* Liten suurin ero tulee esille funktiossa ComputeCost, joka laskee lyhimmän mahdollisen kustannuksen solmusta s, reunaan $\overrightarrow{s_a s_b}$. Funktion toiminta on esitetty kuvassa 3.5.[24]

3.3 ARA*

Vaikka edelliset kaksi algoritmia paikkaavat joitain A*:n puutoksia yksi yhteinen ominaisuus niillä kaikilla on se, että reitin löytäessään ne löytävät aina optimaalisen reitin, tai Field D*:n tapauksessa niin optimaalisen kuin lineaarisella interpoloinnilla on mahdollista, sekä suorittavat hakunsa aina alkusolmusta maalisolmuun. Optimaalinen haku on kuitenkin usein toteuttamiskelvoton todellisen maailman ongelmiin, sillä hauilla on usein rajattu aika, jonka ne voivat käyttää hakujen suorittamiseen. Anytime-algoritmit kuten ARA* ratkaisevat toisen edellä mainituista ongelmista. Ne kyllä suorittavat haut aina alkusolmusta maalisolmuun mutta ne kykenevät antamaan epäoptimaalisen ratkaisun ongelmaan hyvin nopeasti ja kykenevät myös parantamaan ratkaisua sen mukaan, miten aikaa ratkaisun löytymiseen on saatavilla. Toisaalta haasteena on tietää, kuinka laadukas nykyinen ratkaisu on suhteessa optimaaliseen, jos optimaalista ratkaisua ei ole saatavilla verrokiksi, sekä kontrolloida optimaalisuutta tarkasti. Tämä olisi kuitenkin käytännössä hyvin tärkeä tieto, jotta voitaisiin luotettavasti sanoa kesken suorituksen, että ratkaisu on nyt riittävän optimaalinen, joten suorituksessa voidaan siirtyä eteenpäin ja jättää aikaa seuraavia suorituksia varten.

Yksinkertaisimmillaan anytime-algoritmi on vain sarja A*-hakuja muuttuvalla inflaatiokertoimella ϵ ($\epsilon > 1$), jolla A*:n h kerrotaan ($\epsilon * h$). Kyseinen algoritmi suorittaisi perättäisiä A*-hakuja pienenevällä kertoimella ϵ. Muuttujan ϵ ollessa yksi kyseessä olisi normaali A*-haku, joka kuten aiemmin on todettu, palauttaisi optimaalisen tuloksen, kun taas suuremmilla kertoimilla tuloksen laatu olisi epäoptimaalinen mutta suorituksen kesto kertoimen suuruusluokan mukaan hyvinkin nopeaa. Näin ollen algoritmin optimaalisuutta kont-



Kuva 3.6: Vasemmalla kolme A*-hakua laskevalla ϵ:lla. ARA*-haut oikealla. Harmaalla on merkitty algoritmin käsittelemät solmut. Tähdellä merkityt solut ovat epäkonsistentteja suorituksen jälkeen.

rolloidaan muuttujan ϵ avulla, mutta toisaalta algoritmi kuluttaa paljon resursseja koska jokainen A*-haku on aina oma itsenäinen hakunsa eikä niitä voida hyödyntää seuraavissa hauissa. Toisaalta koska haut suoritetaan aina samasta alkupisteestä loppupisteeseen eikä erona ole kuin eri inflaatiokerroin, haut ovat keskenään hyvin samankaltaisia, joten hakuja voisi varmasti käyttää apuna seuraavissa hauissa. ARA* (Anytime Repairing A*) on algoritmi, joka osaa käyttää edellisiä hakuja hyväkseen sekä myös kontrolloida optimaalisuutta.

ARA* suorittaa A*-hakuja useamman kerran peräkkäin pienenevällä inflaatiokertoimella ϵ, kunnes ϵ on 1 ja haku vastaa normaalia A*-hakua ja palauttaa näin optimaalisen tuloksen. Jokainen haku on myös näin ϵ-optimaalinen. ARA* kuitenkin käyttää edellisiä hakuja apuna uuden haun suorituksessa.

ARA* esittelee käsitteen ”paikallinen epäkonsistenssi” (*local inconsistency*). Solmu on paikallisesti epäkonsistentti aina kun sen g-arvoa lasketaan siihen saakka, kun se tulee käsiteltäväksi. Toisin sanoen solmun g-arvon lasku aiheuttaa paikallisen epäkonsistenssin solmun s g-arvon ja sen seuraajien g-arvojen välille, koska g-arvot lasketaan aina edeltävien solmujen mukaan. Muutokset g-arvoissa vaikuttavat siis aina kaikkiin seuraajiin. Aina kun s tulee käsiteltäväksi, paikallinen epäkonsistenssi tila korjataan s:n ja sen seuraajien väliltä, mutta tämä taas aiheuttaa seuraajien tulemisen epäkonsistenteiksi. Näin epäkonsistentti tila propagoituu aina seuraajiin, kunnes minkään solmun g-arvoa ei tarvitse enää laskea. Koska aina kun solmun g-arvoa lasketaan, se lisätään avoimeen listaan ja kun se käsitellään, se poistetaan avoimesta listasta avoin lista sisältää kaikki paikallisesti epäkonsistentit solmut. Avoin lista on siis kokoelma solmuja, joista propagoidaan paikallinen epäkonsistenssi.

A*, joka käyttää luvallista heuristiikkaa ei koskaan käsittele yksittäistä solmua kuin kerran. Jos heuristiikkafunktio kerrotaan kuitenkin inflaatiokertoimella, heuristiikka ei ole

enää luvallinen vaan haku saattaa käsitellä yksittäistä solmua useaan kertaan. Käy ilmi, että jos suoritusta rajoitetaan niin että jokainen solmu voidaan käsitellä vain kerran optimaalisuus raja-arvo ϵ pitää edelleen, siis laskennan tulos ei voi koskaan olla huonompi kuin ϵ kertaa optimaalinen tulos. Solmua ei siis lisätä avoimeen listaan, jos se on jo kertaalleen käsitelty. A*:stä tutusti jo käsitellyt solmut lisätään suljettuun listaan, joten suljettuun listaa lisättyjä solmuja ei oteta enää uudelleen käsiteltäväksi lisäämällä niitä uudestaan avoimeen listaan. Näin voidaan taata, että jokainen solmu käsitellään vain kerran, mutta ongelmaksi koituu se, että avoin lista ei välttämättä sisällä enää kaikkia epäkonsistenttejä solmuja, vaan ainoastaan ne, joita ei ole vielä käsitelty. Algoritmin suorituksen kannalta on kuitenkin pakollista ylläpitää tietoa kaikista epäkonsistenteistä solmuista. Siksi algoritmiin lisätään kokoelma INCONS, joka sisältää kaikki ne solmut, jotka ovat epäkonsistenttejä mutta joita ei ole voitu kuitenkaan lisätä avoimeen listaan.

ARA*:n varsinainen pääfunktio kutsuu sen ImprovePath-funktiota toistuvasti aina pienenevin ϵ-arvoin. Ennen jokaista kutsua uusi avoin lista luodaan yhdistämällä se ja INCONS kokoelma, jonka jälkeen avoin lista järjestetään uudelleen niiden f-arvojen mukaan sillä muuttunut ϵ muuttaa myös f-arvoja. Näin ollen jokaisen kutsun jälkeen saadaan tulos, joka on korkeintaan ϵ kertaa optimaalinen tulos. Optimaalisuuden raja-arvo voidaan laskea myös maalisolmun g-arvon, joka antaa tulokselle ylärajan, sekä pienimmän ei inflaatiokertoimella kerrotun paikallisesti epäkonsistentin solmun suhteena, joka on vastaavasti tuloksen alaraja, optimaalinen tulos ei voi olla tätä parempi. Tämä pätee kaikissa tilanteissa, kun suhde on yksi tai enemmän. Muuten tulos on jo optimaalinen. ARA* laskee siis todelliseksi optimaalisuusrajaksi pienemmän näistä kahdesta vaihtoehdosta. Algoritmin toiminta on esitetty kuvassa 3.7. [28]

3.4 LSS-LRTA*

Jos anytime-algoritmit kuten edellä mainittu ARA* pyrkivät ratkaisemaan A*:n ongelman, jossa se löytää vain optimaalisen reitin tai ei reittiä ollenkaan, LSS-LRTA* (*local search space learning real time a star*) kuten muut reaaliaika-algoritmit, pyrkii ratkaisemaan ongelman, jossa haku on suoritettava aina lähtösolmusta maalisolmuun tai päinvastoin. Nämä algoritmit suorittavat hakua vain rajatulla, etukäteen määrätyllä alueella, jonka jälkeen robotti liikkuu annetun tuloksen mukaan. Laskenta on hyvin nopeaa mutta laskenta-alueen koon mukaan ratkaisut voivat olla hyvin epäoptimaalisia.

Algoritmi ylläpitää tietoa kaikkien solujen h-arvoista, jotka alustetaan algoritmin suorituksen alussa vastaamaan valitun heuristisen metodin arvoja. Näin ollen lähellä maalisol-

Procedure fvalue(s)

return $g(s) + \epsilon * h(s)$;

Procedure ImprovePath()

while *fvalue*(s_{goal}) > $min_{s \in OPEN}$(*fvalue*(s)) **do**
    remove s with the smallest f-value from OPEN;
    CLOSED = CLOSED $\cup \{s\}$;
    for *each successor* s' *of* s **do**
        if s' *was not visited before* **then**
            $g(s')$ = ∞;
        if $g(s')$ > $g(s) + c(s, s')$ **then**
            $g(s')$ = $g(s) + c(s, s')$;
            if $s' \notin$ *CLOSED* **then**
                insert s' into OPEN with $fvalue(s')$;
            else
                insert s' into INCONS;

Procedure Main()

$g(s_{\text{goal}})$ = ∞; $g(s_{\text{start}})$ = 0;
OPEN = CLOSED = INCONS = $\emptyset$;
insert s_{start} into OPEN with $fvalue(s_{\text{start}})$;
ImprovePath();
ϵ' = $min(\epsilon, g(s_{\text{goal}})/min_{s \in \text{OPEN} \cup \text{INCONS}}(g(s) + h(s)))$;
publish current ϵ-suboptimal solution;
while $\epsilon' > 1$ **do**
    decrease ϵ;
    Move states from INCONS into OPEN;
    Update the priorities for all $s \in$ OPEN according to $fvalues(s)$;
    CLOSED = $\emptyset$;
    ImprovePath();
    ϵ' = $min(\epsilon, g(s_{\text{goal}})/min_{s \in \text{OPEN} \cup \text{INCONS}}(g(s) + h(s)))$;
    publish current ϵ-suboptimal solution;

Kuva 3.7: ARA*

mua olevat solut saavat h-arvokseen pienen luvun, kun taas kaukana maalisolmusta olevat solmut suuremman luvun. Algoritmin suorituksen aikana h-arvoja päivitetään vastaamaan paremmin todellisia arvoja, joita algoritmi käyttää määrittämään robotin reitin.

Algoritmin toiminta perustuu käytännössä neljään askeleeseen, joista neljännen jälkeen palataan aina ensimmäiseen. Näitä askeleita toistetaan, kunnes robotti on saavuttanut maalisolmun.

1. Algoritmi suorittaa ensimmäisessä askeleessa A*-haun kohti maalisolmua. Muuttuja lookahead(N) määrittää kuinka suuri tämä haku on, eli se määrittää kuinka monta solmua haku käsittelee ennen kuin se lopetetaan. Kaikki ne solmut, joita A*-haku käsitteli eli lisäsi suljettuun listaan, muodostavat paikallisen hakuavaruuden ja ne annetaan seuraavaan askeleeseen.

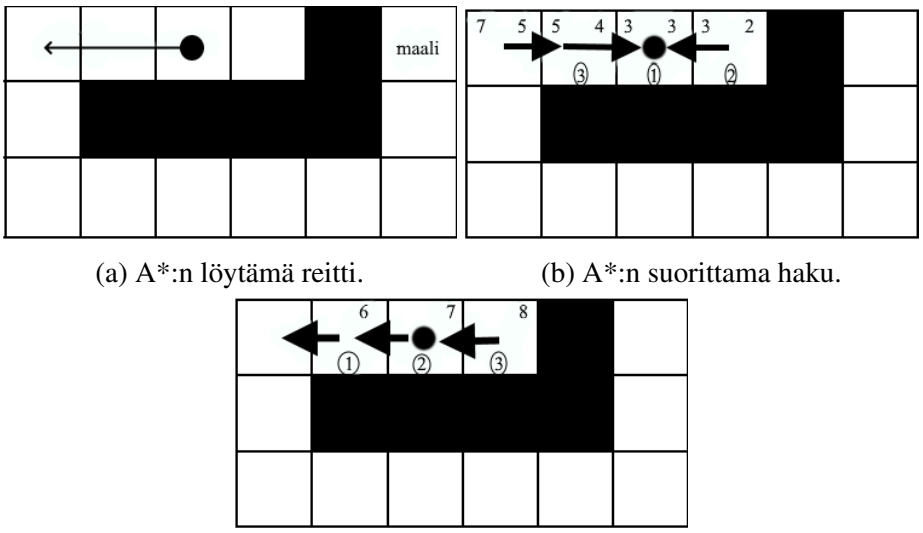
2. Päivitetään kaikkien niiden solmujen, jotka kuuluvat paikalliseen hakuavaruuteen h-arvot. Arvo saadaan, kun lisätään hakuavaruuden rajalla, siis edellisen askeleen avoimessa listassa olevat solmut, olevan pienimmän h-arvon omaavan solmun etäisyys hakuavaruuteen kuuluviin solmuihin. Hakuavaruuden solmujen h-arvot päivitetään siis vastaamaan hakuavaruuden rajan solmujen h-arvoja.

3. Liikutetaan robottia A*:n löytämää reittiä pitkin, kunnes se saavuttaa paikallisen hakuavaruuden rajan tai se havaitsee muutoksia ympäristössään.

4. Jos nykyinen tila on eri kuin maalisolmu siirry takaisin askeleeseen 1, muuten lopeta suoritus onnistuneesti.

Käytännössä askel 1 on rajoitettu A*-haku ja askeleessa 2 käytetään Dijkstran algoritmia päivittämään hakuavaruuden solmujen h-arvot. Kaikki solmut käsitellään kummassakin tapauksessa vain kerran, joten haku on tehokasta. Agentti liikkuu kohdassa kolme kohti sitä solmua, jonka A* olisi ottanut käsittelyyn seuraavaksi. Ideana on käyttää kohdassa kaksi Dijkstran algoritmia päivittämään paikallisen hakuavaruuden h-arvot paikallisesti konsistenteiksi propagoiden informaatiota paikallisen hakuavaruuden reunoilta hakuavaruuden solmuihin. Näin solmujen h-arvot päivittyvät kohti aitoja arvoja.

Algoritmin toimintaa esitellään kuvassa 3.8. Tässä erimerkissä algoritmin lookahead-arvo on asetettu kolmeen. A* käy läpi kolme solmua kuvassa esitetyssä järjestyksessä. Kuvassa on esitetty algoritmin solmuille laskemat f sekä h arvot. Dijkstran algoritmi päivittää solmujen h-arvot sen solmun mukaan, joka olisi ollut seuraavana A*:n suoritettavana. Koska kyseisen solmun h-arvo on viisi, päivitetään käsiteltyjen solmujen, paikallisen hakuavaruuden, arvoiksi 6, 7 ja 8. Näin algoritmi oppii askel askeleelta toimintaympäris-



(a) A*:n löytämä reitti. (b) A*:n suorittama haku.

(c) Dijkstran algoritmi.

Kuva 3.8: LSS-LRTA* toiminta kun lookahead-arvo on asetettu kolmeen. Ympyröidyt luvut kertovat suoritusjärjestyksen.

tönsä todelliset h-arvot. Algoritmin toiminta kokonaisuudessaan on esitetty kuvassa 3.9. [25]

Procedure AStar()

for *each* $s \in S$ **do**
    $g(s) = \infty$
g(s_{start}) = 0;
insert s_{start} into OPEN;
expansions = 0;
while $g(s_{goal}) > min_{s' \in OPEN}(g(s') + h(s'))$ *AND* expansions < lookahead **do**
    expansions = *expansions* + 1;
    delete a state s with the smallest f-value $g(s) + h(s)$ from OPEN;
    CLOSED = CLOSED $\cup \{s\}$;
    for *each* $a \in A(s)$ **do**
        if $g(succ(s,a)) > g(s) + c(s,a)$ **then**
            $g(succ(s,a)) = g(s) + c(s,a)$;
            $tree(succ(s,a)) = s$;
            if $succ(s,a)$ *is not in OPEN* **then**
                insert $succ(s,a)$ into OPEN;

Procedure Dijkstra()

for *each* $s \in$ *CLOSED* **do**
    $h(s) = \infty$;
while *CLOSED* $\neq \emptyset$ **do**
    delete a state s with the smallest h-value $h(s)$ from OPEN;
    if *if* s *is in CLOSED* **then**
        delete s from CLOSED;
    for *each* $s' \in S$ *and* $a \in A(s')$ *with succ(s',a) =* s **do**
        if $s' \in$ *CLOSED AND* $h(s') > c(s',a) + h(s)$ **then**
            $h(s') = c(s',a) + h(s)$;
            if s' *is not in OPEN* **then**
                insert s' into OPEN;

Procedure Main()

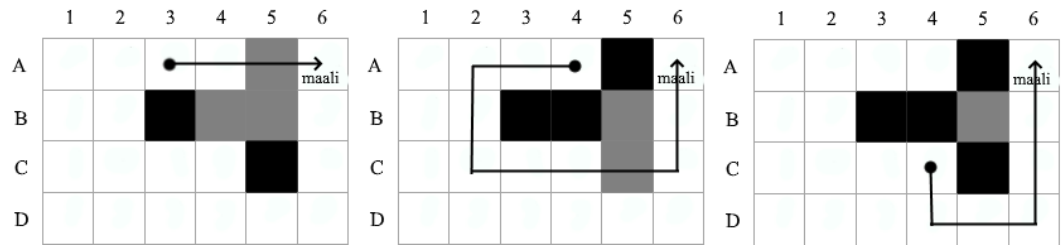
while $s_{start} \neq s_{goal}$ **do**
    AStar();
    if *OPEN* = $\emptyset$ **then**
        stop;
    s_{goal} = arg $\min_{s' \in \text{OPEN}}(g(s') + h(s'))$ (assign s_{goal} if possible);
    Dijkstra();
    move the agent along the path, until goal is found or action costs change.
    if action costs change, update costs.

Kuva 3.9: ARA*

Luku 4

Navigoinnin testaus

Algoritmien toiminta tuntemattomassa ympäristössä lisää huomattavan määrän haastetta robotin navigointiin verrattuna tunnettuun ympäristöön. Tunnetussa ympäristössä algoritmi voi vain laskea reitin alkusolmusta maalisolmuun sekä liikkua reittiä pitkin maaliin. Muutoksia reittiin tarvitsee tehdä vain, jos suoritusympäristö jostain syystä muuttuu. Tuntemattomassa ympäristössä robotin on havainnoitava ympäristöään ja ylläpidettävä tietoa sen tekemistä havainnoista suorituksen aikana. Sen lisäksi sen on päätettävä, miten se suhtautuu tuntemattomiin solmuihin. Yleinen tapa on olettaa, että tuntemattomat solmut ovat vapaita (*freespace assumption*). Näin robotti suorittaa hakuja aina havaintojensa perusteella ja olettaa tuntemattomat solut hakujensa aikana vapaiksi. Liikkuessaan laskemillaan reiteillä se havainnoi ympäristöään. Törmätessään esteeseen se laskee reitin uudelleen tekemiensä havaintojen perusteella. Näin robotti vuorottelee aina hakujen ja siirtymisten välillä. Näin robotin todellinen reitti saattaa poiketa paljonkin sen tekemistä hauista. Robotti ei voi suorittaa liikkeitä, jotka tekevät sen pääsyn maaliin mahdottomaksi sillä robotti voi aina palata takaisin samaa reittiä mitä se kulki, kunhan maalisolmun ja lähtösolmun välinen reitti ei ole blokattu ja mahdollista löytää.[25]



Kuva 4.1: Navigointi tuntemattomassa ympäristössä.

4.1 Testijärjestely

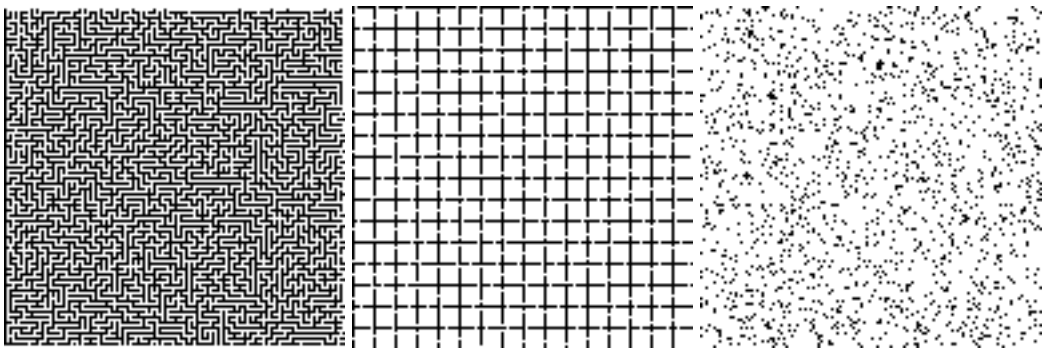
Algoritmien testaamisessa toteutetaan Sven Koenig and Xiaoxun Sunin artikkelissa [25] esiteltyä skenaariota hieman muokattuna. Kaikkien solmujen oletetaan olevan vapaita (*freespace assumption*) jollei solmua ole havaittu esteeksi sekä algoritmien heuristiikkana käytetään heikkoja h-arvoja (Chebyshev-etäisyys). Robotti näkee alueen ympärillään ennalta määrätyn alueen ja päivittää näkymäänsä sen mukaan. Diagonaalisten siirtymien pituus on $\sqrt{2}$ sekä suorien siirtymien 1. Tämä on valittu erityisesti siksi että Field D*:n generoimien reittien pituus olisi mahdollisimman verrannollinen muiden algoritmien kanssa. Field D*-algoritmiin on tehty myös muita muutoksia johtuen siitä, että algoritmin alkuperäinen ComputeCost-funktio olettaa, että solmuilla voi olla useita erilaisia c-arvoja (cost), joten eri solmujen siirtymän kustannus voi olla mielivaltainen arvo. Tässä tutkielmassa siirtymien oletetaan olevan joko 1 tai $\sqrt{2}$ ja mielivaltaisen kustannuksen toteuttavan testausjärjestelyn toteuttaminen jätetään tulevaksi työksi.

Robotin on kyettävä liikkumaan autonomisesti sattumanvaraisesta alkupisteestä sattumanvaraiseen maalipisteeseen. Suoritusalue on diskretisoitu soluihin, jotka ovat joko ummessa tai avoinna. Agentti ei tiedä etukäteen mikä solujen tila on ennen kuin se havainnoi solun. Agentin havainnointialuetta voidaan säätää ennen suoritusta mutta ei sen aikana. Tämän vuoksi robotin kulkema matka on todennäköisesti hyvin paljon pidempi kuin jos robotti tietäisi etukäteen suoritusalueen tilat.

Jokainen algoritmi suorittaa kolme erilaista kenttää. Jokainen kenttä suoritetaan sata kertaa sattumanvaraisilla alku ja loppusolmuilla. Nämä suoritetaan kolme kertaa ja lasketaan niistä keskiarvo. Testit suoritetaan laitteella, jonka prosessorina on Intel(R) Core(TM) i7-4720HQ CPU@2.60GHz. Keskusmuistia on kahdeksan gigatavua. Testien suoritusajat vaihtelivat suuresti kymmenistä minuuteista muutamiin sekunteihin. Robotin näkemän alueen vaikutus jätetään myöhemmäksi työksi.

4.2 Testiaineisto

Haut suoritetaan kolmella erityyppisellä kartalla. Näin saadaan käsitystä, miten algoritmit toimivat erilaisissa ympäristöissä. Kartat ovat nimeltään Maze128, Room128 sekä Ran128 ja ne on esitetty kuvassa 4.2. Ne ovat kooltaan 128×128 solua ja ovat tyypiltään sokkelo, huoneita sisältävä kartta sekä kartta, jossa on täysin sattumanvaraisia esteitä. Karttojen kokoa on pienennetty, jotta 100 testiajoa ei kestäisi liian pitkään. Alkuperäiset kartat ovat ladattavissa osoitteesta https://github.com/nathansttt/hog2/tree/master/maps.



(a) Maze128 (b) Room128 (c) Ran128

Kuva 4.2: Kartat Maze128, Room128 sekä Ran128.

4.3 Algoritmien valinta

Testattavat algoritmit valittiin artikkelissa “A Survey and Classification of A* based Best-First Heuristic Search Algorithms” [23] mainituista luokista. Luokista valittiin 3 sekä myös artikkelissa mainittu any-angle/inkrementaali-hybridi, Field D*.

Koska agentti ei tiedä onko yksittäinen solmu vapaa vai ei, ennen kuin se on havainnoinut kyseisen solmun, ohjelmassa on oltava tieto koko ajan siitä, millainen ympäristö on todellisuudessa, sekä toisaalta myös tieto siitä millaisena algoritmi näkee kyseisen ympäristön kyseisellä hetkellä. Haut suoritetaan aina algoritmin näkemässä ympäristössä ja robotin liikkuessa se havainnoi ympäristöään. Jos jossain kohtaa robotin liikkumista se havainnoi esteen edessään se laskee uuden reitin nykyisten havaintojen perusteella.

Kaikki algoritmeja käsittelevät artikkelit sisälsivät myös kuvauksen ympäristöstä, jossa algoritmi toimii ja millaisessa ympäristössä sitä on testattu. Ympäristöt poikkesivat toisistaan, joten itse toteuttamani testiympäristö on jonkinlainen kompromissi näistä. Erityisesti Field D*:n testiympäristö poikkeaa muiden algoritmien testiympäristöstä. Muut algoritmit olettavat siirtymäkustannusten olevan aina yksi tai ääretön kun taas Field D* käyttää mielivaltaisia siirtymäkustannuksia, joissa kustannus on siirtymän pituus kertaa kustannus. Toisaalta Field D*:n testiympäristössä ei ole esteitä lainkaan. Vain soluja, joiden kustannus on suuri. Kompromissina oman testiympäristön kaikkien solujen kustannus asetetaan yhdeksi mutta siirtymien pituus lasketaan tietyllä tarkkuudella. Tarkoittaen käytännössä, että diagonaalisten siirtymien kustannus on $\sqrt{2}$ ja muiden aina 1. Heuristisina arvoina käytetään kokonaislukuja sillä käytännön testien perusteella se lisää varsinkin Field D*:n luotettavuutta sekä nopeuttaa sen suoritusaikoja merkittävästi. Field D*-algoritmia on muutenkin yksinkertaistettu, sillä testiympäristössä kaikkien solujen kustannukseksi on asetettu yksi, joten testiympäristössä mitataan vain kunkin algoritmin tuottaman reitin

pituutta.

Artikkelissa "Comparing Real-Time and Incremental Heuristic Search for Real-Time Situated Agents" esitelty testiympäristö vuorostaan ei salli diagonaalisia siirtymiä lainkaan vaan mahdollisia siirtymiä on vain neljä. Siirtymien määrä muilla kuin Field D*:llä on asetettu näissä testeissä kahdeksaan, jotta Field D*:ä voitaisiin verrata muihin algoritmeihin.

Artikkelissa "Fast Replanning for Navigation in Unknown Terrain" esitelty testausympäristö vuorostaan sallii diagonaaliset siirtymät mutta kaikkien siirtymien kustannus on tässä tapauksessa asetettu joko äärettömään tai yhteen. Koska Field D* laskee omien reittiensä pituuden hyvinkin tarkasti, on tässä testiskenaariossa asetettu kaikkien muiden algoritmien diagonaalisten siirtymien pituudet neliöjuuri kahteen, jotta ne vastaisivat mahdollisimman tarkasti Field D*-algoritmin laskemien reittien pituuksia.

4.4 Testiohjelmisto

Algoritmeja testataan Python-ohjelmointikielellä tätä tarkoitusta varten kehitetyllä ohjelmalla. Ohjelma koostuu algoritmeista, jokaiseen algoritmiin liittyvästä navigaattorista, sekä testit suorittavasta agentista. Näiden lisäksi on luokka Environment, joka sisältää tiedot ympäristöstä, jossa testit suoritetaan. Jokaisen suorituksen aikana luodaan kaksi instanssia Environment-luokasta. Instanssi, joka sisältää tiedon ympäristön todellisesta tilasta sekä instanssi, joka sisältää tiedon siitä millaisena ympäristössä liikkuva robotti näkee ympäristön sillä hetkellä. Robotin havainnot sisältävä luokka alustetaan sisältämään vain robotin alkupisteessä havaitsevat esteet ja tämän jälkeen suoritus alkaa. Osa Environment-luokan lähdekoodia on esitetty kuvassa 4.1. Luokka sisältää tarvittavat tiedot ympäristöstä. Lisäksi siltä voidaan pyytää tietoa yksittäistä solua ympäröivistä soluista, sekä esteistä. Luokalta voidaan pyytää myös yksittäisen solun tilaa eli tietoa onko kyseinen solu blokattu vai vapaa. Algoritmit sekä navigaattorit käyttävät tätä luokkaa reittien laskemiseen sekä liikkumiseen.

```
1000 class Environment:
       """
1002   A class used to represent a 2D environment
       Single cell is either open or blocked
1004
       ...
1006
       Attributes
1008   ----------
       y_range : int
1010     the height of the map
```

```
    x_range : int
      the width of the map
    map : list
      contains all cells of the map
    obs : dict
      dictionary containing information if a cell is blocked or not
      format: tuple : bool    (y,x) : (True or False)

    Methods
    -------
    get_obstacles(self, pos: tuple, range_: int = 1) -> list
      returns all obstacles around pos with range range_

    get_neighbors_range(self, pos: tuple, range_: int) -> list
      returns all open cells around pos with range range_

    get_neighbors(self, pos: tuple) -> list
      returns all open cells around pos with range 1
      checks that the route does not go through an obstacle

    test_obstacle(self, s: tuple) -> bool
      returns bool value telling if cell s is obstacle or not
    """

    def __init__(self, height: int, width: int, map_2dlist: list, obs_dict: dict):
      """
      Parameters
      ----------
      height: int
        the height of the map
      width: int
        the width of the map
      map_2dlist: list
        contains all cells of the map
      obs_dict: dict
        dictionary containing information if a cell is blocked or not
        format: tuple : bool    (y,x) : (True or False)
      """
```

Kuva 4.1: Luokka Environment

Algoritmien tehtävänä on laskea reittejä robotin nykyisestä sijainnista maaliin sen nykyisten havaintojen perusteella. Jokaista algoritmia kutsutaan antamalla sille nykyinen sijainti parametrina ja jokainen algoritmi palauttaa lasketun reitin sekä reitin pituuden. Navigaattorien tehtävänä on suorittaa hakuja robotin liikkeiden välissä. Navigaattori laskee ensin reitin nykyisten havaintojen perusteella kutsumalla valittua algoritmia. Algoritmin palautettua reitin se liikuttaa robottia reittiä pitkin ja havainnoi ympäristöä samalla. Jos reitti katkeaa robotin havaittua esteen sen edessä navigaattori palaa laskentavaiheeseen, jossa se kutsuu algoritmia ja saa siltä uuden reitin. Tätä jatketaan, kunnes robotti saavuttaa maalin. Kuvissa 4.2 ja 4.3 on esitetty osa ARA*-algoritmin navigaattorista sekä algoritmista.

```
class ARANavigator(AbstractNavigator):
    """
    This class contains navigation logic for ARA algorithm
    It moves the agent in the environment and replans the route
    every time the agent finds a blocking obstacle

    ...

    Attributes
    ----------
    s_goal : tuple
        target of the search
    s_start : tuple
        starting point of the search
    o_start : tuple
        original starting position
    position : tuple
        current position of the agent, where agent has moved
    env : Environment
        the environment where agent is moving and searching is done
    y : int
        the height of the map
    x : int
        the width of the map
    e : float
        initial value for inflation factor epsilon
    current_observed : Environment
        how the agent sees the environment currently, found obstacles
    path : list
        path from start to goal
    visited : list
        searched nodes in order
    totalPathCost : float
        total cost of the path

    Methods
    -------
    run(self)
        run the search
    """

    def __init__(self, env: Environment, s_start: tuple, s_goal: tuple, e: \
    float = 3, range: int = 1):
        """
        Parameters
        ----------
        env : Environment
            the environment where agent is moving and searching is done
        s_goal : tuple
            target of the search
        s_start : tuple
            starting point of the search
```

Kuva 4.2: Luokka ARANavigator

```
class AraStar(AbstractAlgorithm, Motions8Algorithm):
    """
    A class used to represent ARA* algorithm
    How e changes between searches can be varied to make searching
    more efficient

    ...

    Attributes
    ----------
    s_goal : tuple
        target of the search
    e : float
        inflation factor epsilon, used to inflate h to make searching faster
    heuristic_type : str
        heuristic method
    env : Environment
        the environment where agent is moving and searching is done
    s_start : tuple
        starting point of the search
    e_delta : float
        delta of e, how much e changes between searches
        lower e makes searching more accurate but slower
    g : dict
        G values in dictionary, key is tuple (y, x), value is current g
    OPEN : dict
        OPEN list of the search, key is tuple (y, x), value is f-value
    PARENT : dict
        relations, used when extracting path
    INCONS : set
        nodes that have been visited already
        and should have been reopened and are therefore inconsistent
    CLOSED : set
        nodes that have been visited already
    path : list
        path from start to goal
    visited : list
        searched nodes in order

    Methods
    -------
    run(self, s_start: tuple) -> tuple
        run the search
    """

    def __init__(self, s_goal: tuple, e: float, heuristic_type: str, env: Environment):
        """
        Parameters
        ----------
        s_goal : tuple
            target of the search
        e : float
            inflation factor epsilon, used to inflate heur to make searching faster
        heuristic_type : str
```

```
1054        heuristic method
            env : Environment
1056          the environment where agent is moving and searching is done
            """
```

Kuva 4.3: Luokka AraStar

Agentti-luokan tehtävänä on hakujen aloittaminen, lopettaminen sekä tilastojen kerääminen. Agentti suorittaa valittua algoritmia valituilla parametreillä. Se arpoo jokaiselle suoritukselle alku- ja loppupisteen ja suorittaa hakuja valitun määrän.

```
1000 class RepeatingAgent(AbstractAgent):
       """
1002   Agent class starts and finishes searches and collects statistics


1004


1006   ...

1008   Attributes
       ----------
1010   navigator : AbstractNavigator
         navigator class doing the search
1012
       Methods
1014   -------
       run(self)
1016     run the search
       """
1018
       def __init__(self, env: Environment, s_start: tuple, s_goal: tuple, \
1020   nav: str, times: int = 100, range: int = 1):
         """
1022     Parameters
         ----------
1024     env : Environment
           the environment where agent is moving and searching is done
1026     s_goal : tuple
           target of the search
1028     s_start : tuple
           starting point of the search
1030     nav : str
           selected navigator
1032     times : int
           number of times simulation runs, default 100
1034     """
```

Kuva 4.4: Luokka RepeatingAgent

Luku 5

Tulosten analysointi

Testien suoritusten aikana kävi nopeasti selväksi, että testien suoritusajat vaihtelevat suuresti kentän, algoritmin sekä erityisesti valittujen parametrien mukaan. Muut algoritmit on toteutettu hyvin tarkasti niiden artikkeleissa esitettyjen algoritmien mukaan, mutta Field D*-algoritmiin tehtiin muutoksia, joissa sitä yksinkertaistettiin vastaamaan muissa artikkeleissa esitettyjä testausympäristöjä ja toisaalta sen suoritustehokkuutta yritettiin näin myös hieman parantaa. Suoritettujen hakujen määrä vaihteli huomattavasti eri karttojen välillä. Hakujen määrä kertoo suoraan, kuinka usein robotin lasketun reitin tielle tuli este ja se joutui laskemaan reitin uudelleen muiden kuin LSS-LRTA*:n tapauksessa. Sen ollessa kyseessä hakujen syvyys on rajattu, joten sen hakujen määrässä näkyy niin valitun kartan vaikutus myös valitun N:n vaikutus. Muilla kolmella algoritmilla hakujen määrä on itseasiassa hyvin samankaltainen mikä ei toisaalta ole suuri yllätys sillä niiden tapa tehdä hakuja ja toimia tuntemattomassa ympäristössä on myös hyvin samankaltainen. Ne laskevat aina nykyisen käsityksen mukaisen reitin maaliin ja liikkuvat sitä kohti, kunnes havaitsevat reitin muutosta vaativan esteen.

Maze128-kartan kaltaisten sokkeloiden kohdalla hakujen määrä kasvaa hyvin suureksi sillä esteitä tulee vastaan hyvin usein. Sen takia voisi olla edullista rajata hakuavaruutta jollain tavalla sillä haut varsinkin isoissa kartoissa maaliin asti saattavat olla turhia sillä robotti ei mahdollisesti liiku hakujen välissä kuin ehkä yhden solun verran. Tämä kävi ilmi hyvin selkeästi testeissä.

Testien suoritusajoissa nähtiin suuria vaihteluita. Lyhimmät testit kestivät pari sekuntia, kun taas pisimmät yli tunnin. Selkeästi hankalin kartta nyt testatussa tuntemattomassa ympäristössä navigoinnissa oli labyrinttimäinen Maze128. Sen hakuajat olivat huomattavan pitkiä muihin verrattuina. Suurimpana syynä on suuri hakujen määrä, joka johtuu

suuresta määrästä pysähdyksiä liikkumisen aikana. Kyseisessä kartassa robotti havaitsee muutoksia lähes joka liikkeen jälkeen, joten jokainen algoritmi suoritti myös selkeästi eniten hakuja siinä.

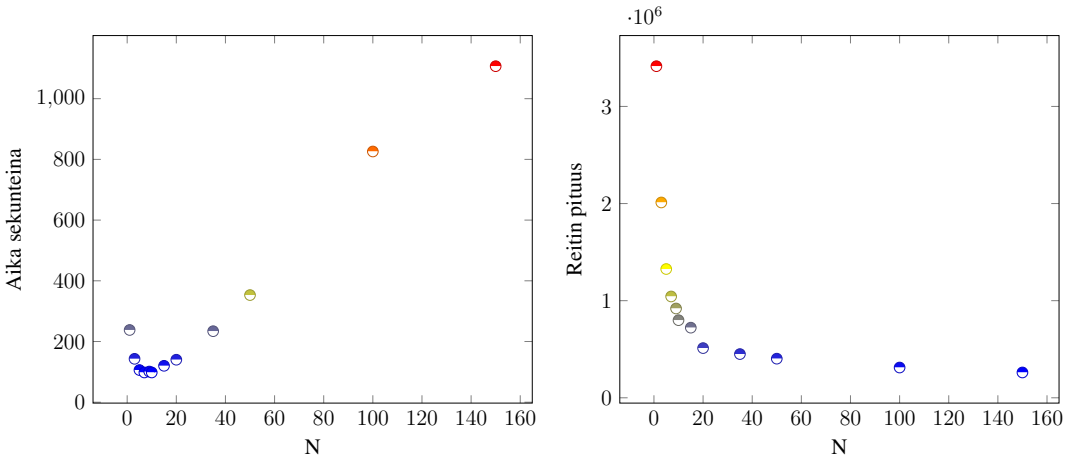
Robottien kulkemat reitit ympäristössä muistuttavat kolmen algoritmin suhteen toisiaan hyvin paljon. Poikkeus sääntöön on LSS-LRTA*. Sen N-parametri vaikuttaa sen reitteihin todella merkittävästi. Mielenkiintoinen yksityiskohta on se, että N-arvojen vaikutus laskettuihin reitteihin riippuu vahvasti ympäristöstä.

5.1 LSS-LRTA*

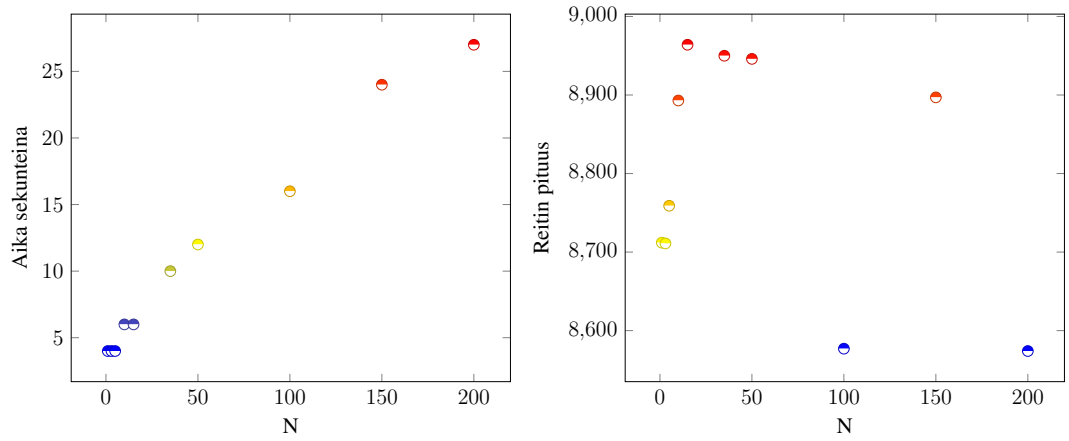
LSS-LRTA*:n erityspiirre on sen parametri N, joka määrittää kuinka syvän A*-haun se tekee ennen liikkumista. N-parametrin vaikutus robotin liikkumiseen tuntemattomassa ympäristössä on hyvin merkittävä ja sen valinta on hyvin tärkeää. Room128 ja Ran128 kartoilla algoritmin suoritusajat kasvoivat tasaisesti, kun N:ää kasvatettiin. Maze128 kartan kohdalla taas löytyi optimaalinen N-arvo, jonka kohdalla suoritusaika oli pienin. Tämä johtui todennäköisesti siitä, että sokkelomaisessa kartassa haun ei tarvitse olla liian pitkä sillä uusia esteitä löytyy koko ajan ja algoritmin on parempi hakea pieneltä alueelta. Suuret syvät haut kuluttavat turhaan resursseja, kun algoritmin on joka tapauksessa laskettava reitti pian uudestaan, kun uusia esteitä löytyy. Room128-kartan kohdalla N-arvon kasvattaminen pidentää suoritusaikoja mutta myös lyhentää reittejä. Mitä suuremmaksi N-arvo kasvaa sitä vähemmän hyötyä sen kasvattamisesta on pituuteen nähden, joten tässä tapauksessa kannattaa valita sopiva N-arvo riippuen paljonko suoritusaikaa on. Ran128-kartan kohdalla N-arvolla ei ollut merkittävää vaikutusta reitin pituuteen mutta suoritusaika kasvoi kyllä merkittävästi. Tästä voi päätellä, että tällaisessa kartassa haun ei tarvitse olla syvä vaan robotti kykenee liikkumaan oikeaan suuntaan hyvin lyhyilläkin hauilla.

Algoritmin laskemien reittien pituudet vaihtelevat eniten kaikista algoritmeista. Pienillä N-arvoilla robotin reitit ovat kaikista algoritmeista selkeästi pisimpiä erityisesti Maze128 ja Room128 kartoilla. Pienillä N-arvoilla robotilla kestää hyvin pitkään päivittää sen h-arvot vastaamaan paremmin todellisuutta, joten algoritmi käyttää hyvin pitkän aikaa oppimaan ympäristöään. Ran128 kartassa alussa alustetut arvioidut h-arvot vastaavat paremmin todellisuutta koska yhtenäisiä esteitä on vähemmän, joten robotti kykenee liikkumaan niiden avulla paremmin kohti maalia.

Hakujen määrä riippuu selkeästi algoritmin N-arvosta. Jos N-arvo on hyvin pieni ja haku lyhyt on selvää, että hakujen määrän on silloin oltava suurempi. Haku käy läpi pienillä N-



Kuva 5.1: LSS-LRTA*-algoritmin N-parametrin vaikutus Maze128-kartalla.



Kuva 5.2: LSS-LRTA*-algoritmin N-parametrin vaikutus Ran128-kartalla.

arvoilla suhteellisesti hyvin pienen määrän solmuja. Koska hakuja on paljon, solmujakin käydään läpi paljon mutta toisaalta solmuja per haku ei ole kovinkaan paljoa. Suurilla N-arvoilla solmujen määrä kasvaa sillä haut ovat syvempiä mutta toisaalta hakujen määrä pienenee koska hakuja ei tarvitse tehdä niin monta.

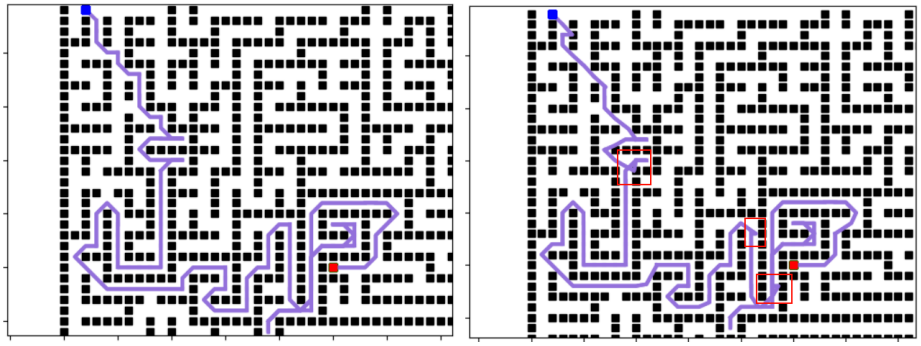
Kuvissa 5.1 ja 5.2 on esitetty N-parametrin vaikutusta suoritusaikoihin sekä reitin pituuteen. Maze128-kartalla on selkeästi huomattavissa N-arvo, jolla suoritusaika on nopein. N-arvoa kasvattamalla hakujen tarkkuus paranee mutta suoritusajat alkavat kasvaa merkittävästi, joten liian korkea N-arvo ei ole kannattavaa vaan arvoksi kannattaa valita mahdollisimman pieni arvo, jolla tarkkuus on vielä riittävä. Ran128-kartalla taas havaitaan, että N-arvolla ei ole juuri merkitystä reitin pituuteen. Näin ollen N-arvon valinta on myös hyvin riippuvainen kartasta, joten sen valinta kannattaa tehdä huolella.

5.2 Field D* ja D* Lite

Field D* menestyi hyvin pituuskategoriassa ja sen laskemat reitit erityisesti Ran128-kartassa olivat hyviä. Kuten artikkelissa [23] mainitaan D* Liten sekä Field D* toiminta perustuu siihen, että peräkkäiset haut ovat keskenään hyvin samankaltaisia ja muutokset erityisesti hakupuun juureen hidastavat hakua huomattavasti. Erityisesti Maze128-tyyppinen kartta mahdollistaa muutokset hakupuun juureen sillä yksittäinen löydetty este saattaa muuttaa löydettyä reittiä todella paljon, jos uusi reitti poikkeaa hyvin paljon vanhasta. Vertailu ARA*:n kertoo, että sekä D* Liten että Field D*:n käsittelemiä solmuja on huomattavasti vähemmän, joten ne hyötyivät solmujen määrässä selkeästi siitä, että niiden ei tarvinnut aloittaa hakujaan aina alusta niiden välillä. Silti Field D*:n suoritusajat olivat hyvin pitkiä jonka suurin selittävä tekijä on todennäköisesti sen suurin ero D* Liteen verrattuna eli ComputeCost()-funktion toiminta. Sen suoritusajat varsinkin Maze128-kartassa olivat hyvin hitaita. Kernprof-ohjelmalla mitattaessa Field D*:n suoritusajasta suurin osa kuluu solmujen käsittelyyn sekä ComputeCost()-funktion suoritukseen jolla lasketaan algoritmin g-arvoja uusiksi muutosten tapahtuessa. Muutokset suoritusalueella osoittautuivat siis hyvin raskaiksi Field D*:lle, mikä voidaan päätellä myös algoritmin hitaista suoritusajoista Maze128-kartalla.

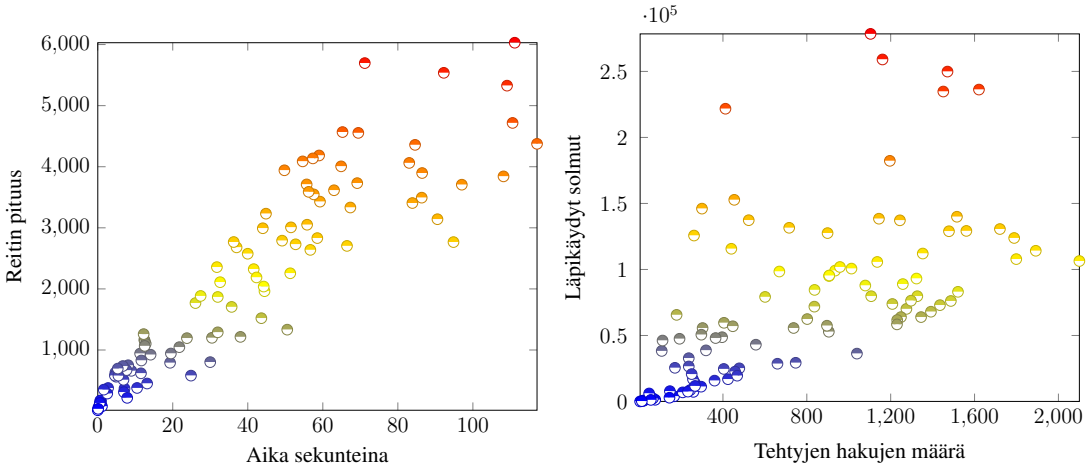
Field D*:n laskemissa reiteissä on otettava huomioon, että koska Field D* kykenee liikkumaan solujen reunoilla se saattaa löytää esteen ollessaan reunalla, siis pisteessä karttaa, joka sisältää desimaaleja. Koska Field D*:n alkupisteen on oltava kuitenkin pelkästään kokonaislukuja sisältävä, robotin on liikuttava aina laskentojen välissä johonkin solmuun (esim. $(3, 3)$ ei $(3.3, 3.72)$). Se mihin robotti liikkuu, voidaan määritellä algoritmin ulkopuolella. Pisteen laskeminen on toteutettu näissä testeissä niin että robotti pyrkii jatkamaan liikettään aina siihen suuntaan, johon se oli menossa. Tämä ei ole välttämättä optimaalinen tapa ja se saattaa lisätä reitin pituuksia. Algoritmin reitti oli kuitenkin poikkeuksetta lyhyempi kuin algoritmilla johon se perustuu, joten vaikka suoritusajat jäivät hieman pettymyksiksi ComputeCost()-funktio kykeni kuitenkin parantamaan D* Liten reittien pituuksia merkittävästi. Suorituskykyongelmat olivat näissä testeissä Field D*:n suurin kompastuskivi.

Verrattaessa ARA*:n sekä Field D*:n reittejä kuten kuvassa 5.3 havaitaan, että reitit muistuttavat hyvin paljon toisiaan mutta ARA*:n reitti on hieman lyhyempi. Field D*:n reitissä havaitaan kohtia, jossa se pyrkii kohti maalisolmua ja törmää esteeseen solun reunalla. Navigaattorin on tällöin päätettävä mihin robotti liikkuu. Kehittyneempi navigaattori voisi liikkua näissä tilanteissa paremmin ja mahdollisesti lyhentää reittiä.

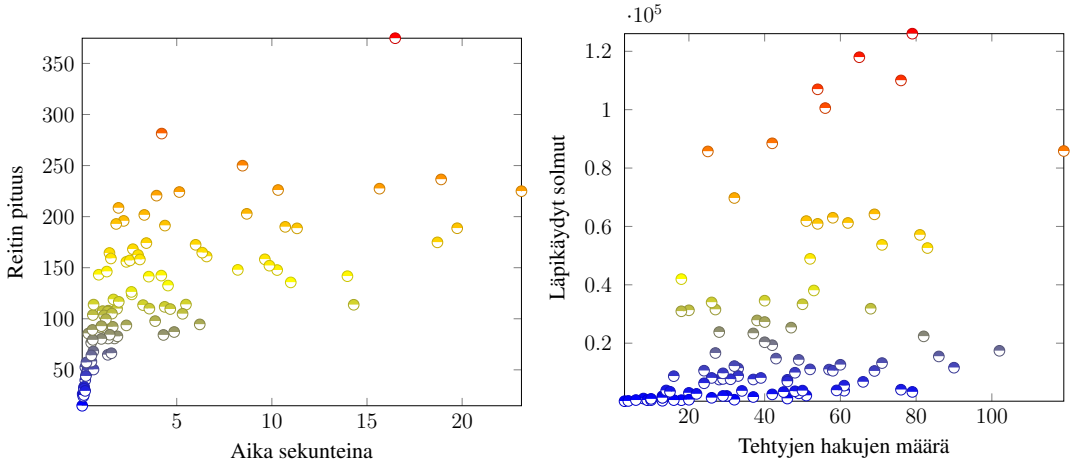


(a) ARA* (b) Field D*

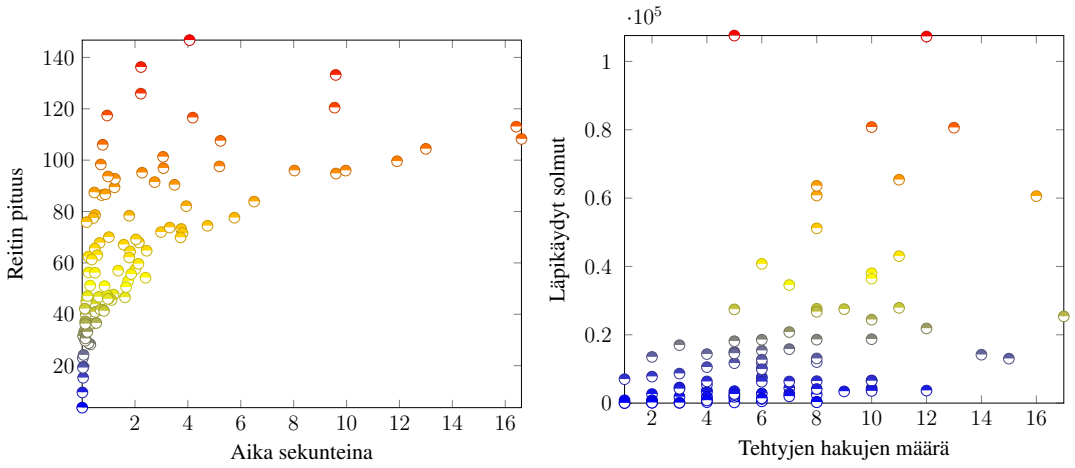
Kuva 5.3: ARA*-robotin sekä Field D*-robotin reitit sokkeloympäristössä



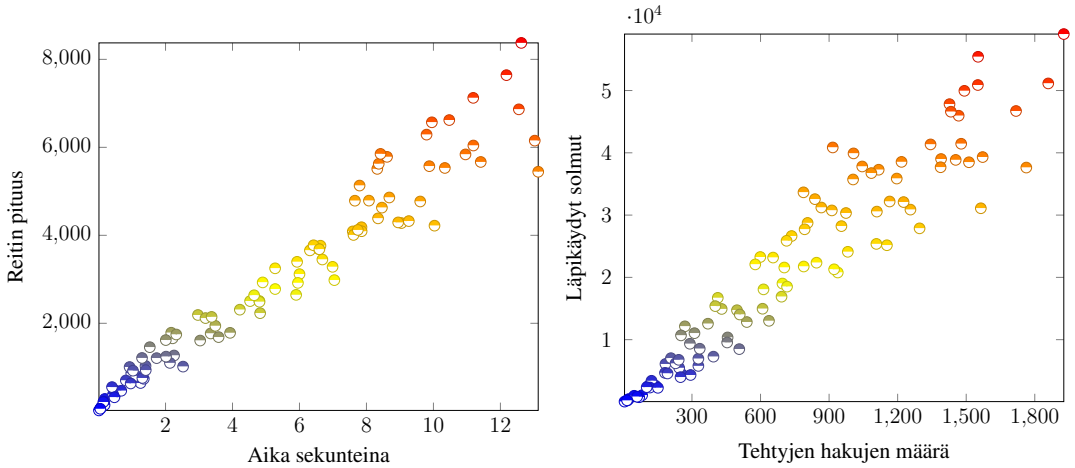
Kuva 5.4: Field D*-algoritmin 100 suoritusta Maze128-kartalla.



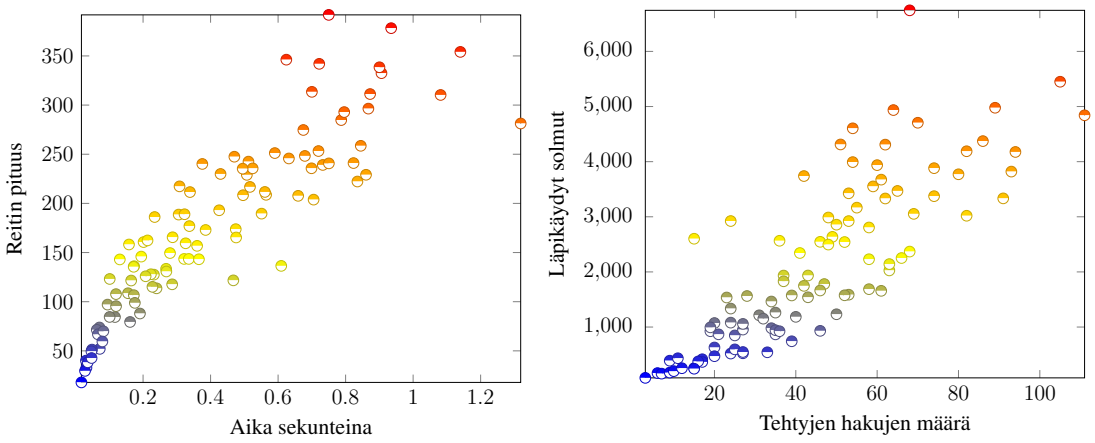
Kuva 5.5: Field D*-algoritmin 100 suoritusta Room128-kartalla.



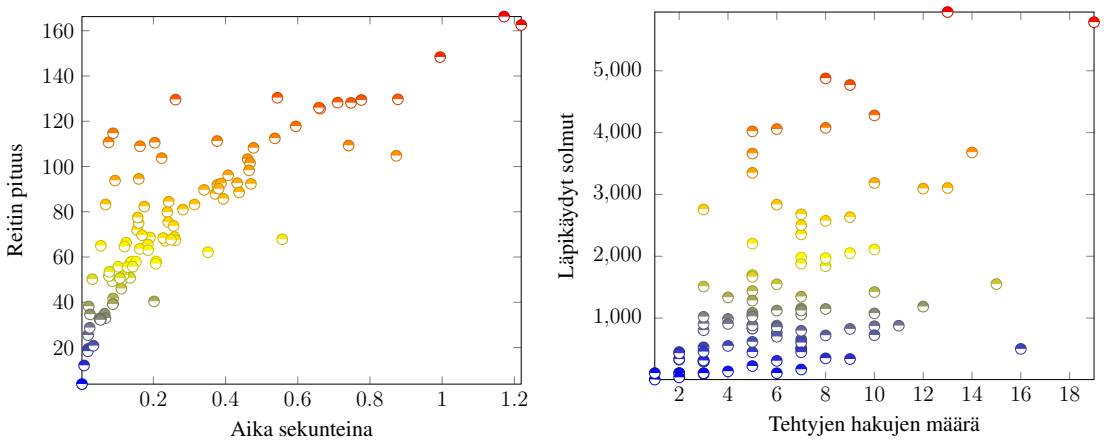
Kuva 5.6: Field D*-algoritmin 100 suoritusta Ran128-kartalla.



Kuva 5.7: D* Lite-algoritmin 100 suoritusta Maze128-kartalla.



Kuva 5.8: D* Lite-algoritmin 100 suoritusta Room128-kartalla.



Kuva 5.9: D* Lite-algoritmin 100 suoritusta Ran128-kartalla.

Kuvissa 5.4, 5.5, 5.6, 5.7, 5.8 ja 5.9 on esitetty yksi testi kummallekin algoritmille kaikissa kolmessa kartassa. Koska yksi testi käsittää sata suoritusta kaikissa kuvaajissa on sata pistettä. Erityisesti Field D*-algoritmilla eri tapausten suoritusajat vaihtelevat suuresti ja pidemmät reitit vaativat yleensä pidemmän suoritusajan. Tosin muutamia tapauksia, joissa reitin pituus on korkea mutta suoritusaika on myös lyhyt, on havaittavissa.

D* Liten suoritusaika oli merkittävästi Field D*:ä parempi. Vaikka algoritmi hävisi reittien pituudessa, sen suoritusajat olivat poikkeuksetta merkittävästi parempia näissä testeissä. D* Liten heikkous oli todennäköisesti sen tarkkuus. Suurilla näissä testeissä käytetyissä kartoissa ARA*:ssa käytetty metodi kertoa heuristiikkafunktiota ϵ:lla voisi nopeuttaa myös D* Liten suoritusta. AD* (anytime dynamic A*) on algoritmi, joka lisää D* Liteen ARA*:sta tutun ϵ:n.

5.3 ARA*

ARA* pärjäsi testeissä hyvin. Heuristiikkafunktion kertominen ϵ:lla osoittautui hyvin tehokkaaksi tavaksi nopeuttaa A*:n suoritusta ilman suurta menetystä tarkkuudessa. Menetetty tarkkuus ei vaikuttanut merkittävästi robotin reitin pituuteen. Tämä johtuu todennäköisesti siitä, että robotti liikkuu hyvin harvoin kovin pitkään sen tekemän haun jälkeen, kunnes se taas löytää esteen ja joutuu hakemaan uuden reitin. Maaliin asti suoritettavien hakujen kannattaa olla nopeita sillä suurimmasta osasta haun tuottamasta reitistä ei ole mitään hyötyä sillä se kaikki on laskettava uudelleen pian. Lyhyen liikkumisen aikana robotilla ei ole aikaa eikä mahdollisuuksia liikkua paljoa epäoptimaalista reittiä vaan se seuraa vain mahdollista seinää, kunnes se löytää mahdollisen aukon ja riittää että robotti liikkuu vain eteenpäin kohti maalia havainnoiden ympäristöä.

Koska ARA* suoritti kaikki haut aina alkutilanteesta, sen läpi käymien solmujen määrä kasvoi suureksi. Jos ϵ on 1 suoritukset hidastuivat merkittävästi ilman vastaavaa hyötyä tarkkuudessa. Myös Maze128-kartan suorittaminen aiheutti muistivirheen (memoryError) 32-bittisellä Pythonilla, joten jos ϵ oli 1 muistinkulutus suurilla kartoilla oli myös hyvin suurta. Tällainen haku vastaa hyvin tarkasti perinteistä A*-hakua, joten siitä voi päätellä, että perinteisen A*-algoritmin tarkkuus tällaisessa ympäristössä on kyllä hyvä mutta suorituskyky ei pärjää tässä työssä käsitellyille algoritmeille.

5.4 Havaintoja

Erityisesti Maze128 kartalla suoritettujen testien kohdalla kävi ilmi, että algoritmit suorittavat hyvin usein kokonaisia hakuja alusta loppuun saakka mutta liikkuvat vain yhden tai kaksi solua. Tämä pätee erityisesti, kun robotti havainnoi ympäristöään vain yhden ruudun eteenpäin. Tavat vähentää resurssien tuhlaamista turhaan osoittautuivat näissä testeissä erinomaisiksi tavoiksi parantaa tehokkuutta hauissa. ARA*:n tapa käyttää ϵ:a ja keventää hakuja, LSS-LRTA*:n tapa rajoittaa hakujen syvyyttä sekä D* Liten tapa käyttää edellistä hakua suoraan uudessa nopeuttivat kaikki hakuja verrattuna vastaavaan A*-hakuun ilman että tarkkuus kärsi juurikaan erityisesti ARA*:n sekä D* Liten kohdalla. LSS-LRTA*:n kohdalla haun tarkkuus on säädettävissä N-parametrilla ja vaikka se ei ihan kykene muiden algoritmien tarkkuuteen sen suorittamien on hyvin nopeaa. Field D*:n suorituskyky jäi näissä testeissä hieman puutteelliseksi mutta sen tarkkuus oli hyvä.

Taulukoissa 5.1, 5.2, 5.3 ja 5.4 on esitetty sadan haun keskimääräinen yhteenlaskettu aika, reittien pituus, suoritettujen hakujen määrä sekä kaikkien läpikäytyjen solmujen yhteismäärä kaikille kolmelle kartalle.

Taulukko 5.1: Field D*

Kartta	Aika	Pituus	Haut	Solmut
Maze128	1:02:35	196339	71614	6655020
Room128	0:09:31	14226	4862	2923623
Ran128	0:05:36	6998	733	1983473

Taulukko 5.2: D* Lite

Kartta	Aika	Pituus	Haut	Solmut
Maze128	0:08:20	280754	72859	1995731
Room128	0:00:57	19663	5055	257390
Ran128	0:00:35	7908	661	173122

Taulukko 5.3: LSS-LRTA*

Kartta	N	Aika	Pituus	Haut	Solmut
Maze128	1	0:03:58	3414277	3134829	6269658
Maze128	3	0:02:23	2011231	1047447	6284650
Maze128	5	0:01:46	1325927	510966	5109244
Maze128	7	0:01:38	1043241	336367	4707828
Maze128	9	0:01:41	919477	267355	4809262
Maze128	10	0:01:38	799137	226535	4526164
Maze128	15	0:02:00	722069	179436	5366660
Maze128	20	0:02:20	511682	134397	5343342
Maze128	35	0:03:54	450225	119985	8264352
Maze128	50	0:05:53	402889	112787	10976060
Maze128	100	0:13:46	311615	100858	18677446
Maze128	150	0:18:27	260717	87099	22911656
Maze128	200	0:25:02	253396	84763	28043802
Room128	1	0:00:09	59735	51661	118366
Room128	10	0:00:11	22855	12410	246594
Room128	100	0:00:46	17748	6010	1004330
Room128	150	0:01:07	16892	5575	1300418
Ran128	1	0:00:04	8712	6110	12220
Ran128	3	0:00:04	8711	4191	25084
Ran128	5	0:00:04	8759	4025	39986
Ran128	10	0:00:06	8893	3433	67758
Ran128	15	0:00:06	8964	3455	101810
Ran128	35	0:00:10	8950	2803	189522
Ran128	50	0:00:12	8946	2413	230472
Ran128	100	0:00:16	8577	1602	286990
Ran128	150	0:00:24	8897	1455	379740
Ran128	200	0:00:27	8574	1273	425926

Taulukko 5.4: ARA*

Kartta	ϵ	Aika	Pituus	Haut	Solmut
Maze128	5	0:08:49	203034	59748	23515767
Maze128	3	0:06:44	188107	59366	17899022
Maze128	2	0:06:27	173086	56512	16779299
Room128	5	0:00:07	14297	4360	205051
Room128	3	0:00:08	14174	4428	203410
Room128	2	0:00:08	13869	4437	198446
Room128	1	0:01:45	13122	4521	3429329
Ran128	5	0:00:02	7794	716	29243
Ran128	3	0:00:02	7794	716	29243
Ran128	2	0:00:02	7794	716	29243
Ran128	1	0:00:26	7641	728	784151

Luku 6

Yhteenveto

Tässä tutkielmassa esiteltiin aluksi A*-algoritmin erilaisia luokkia tai kategorioita joihin A*:n variantteja on jaettu. Jokaisesta neljästä kategoriasta valittiin yksi algoritmi ja nämä toteutettiin Python-ohjelmointikielellä toteutetussa ympäristössä, joka kykenee keräämään tilastoja algoritmin suorituksesta. Ympäristössä liikkuva robotti ei tiennyt suorituksen alussa ympäristöstä mitään vaan sen oli kyettävä löytämään reitti havainnoimalla ympäristöä. Jokainen algoritmi testattiin kolmella erilaisella kartalla, jossa ne suorittivat 100 hakua sattumanvaraisilla alku- ja loppupisteillä. Tämä toistettiin kolme kertaa. Jokaisesta hausta kerättiin suoritusajat, reittien pituudet, hakujen määrät sekä käsitellyt solmut.

Kaksi algoritmeista oli parametrisoituja, joten niiden parametrien arvojen vaikutuksia tutkittiin. Kahden muun algoritmin tapauksessa näin ei voitu tehdä. Parametrisoitujen algoritmien tapauksessa niillä huomattiin olevan merkittävä vaikutus algoritmin suoritukseen. N-parametrin vaikutus oli hyvin merkittävä ja sen havaittiin olevan myös vahvasti kartasta riippuva, joten sen valinta tulisi tehdä harkitusti. ARA* nopeutti normaalin A*:n suoritusta merkittävästi tekemällä sen heuristiikasta epätarkempia. Näin suoritusajat paranivat huomattavasti mutta hauista ei tullut merkittävästi epätarkempia.

D*-Lite ei suoriutunut testeistä huonosti mutta toisaalta siitä puuttuva parametrisointi voi olla kriittinen puute arvioidessa sen monikäyttöisyyttä. Toisaalta edellä mainittu AD* voi mahdollisesti korjata tämän puutteen. Yhdistämällä D* Lite ja ARA* voidaan mahdollisesti saada D* Liten suoritusta nopeutettua sen verran että se kykenee toimimaan paremmin tilanteissa, joissa suoritusaika on kriittinen.

Suoritusaikoja vertailtaessa on hyvin selvää, että Field D* oli valituista algoritmeista hitain. Sen saaminen toimimaan riittävän nopeasti, että testaus oli ylipäätään mahdollista toteuttaa 128×128- kokoisilla kartoilla vaati hyvin paljon työtä. Se suoriutui erityisen

huonosti Maze128-kartasta, jossa sen kyvystä tuottaa mahdollisimman suoria reittejä oli vähiten hyötyä.

6.1 Impakti

Nasan mukaan Marsia tutkineet Spirit ja Opportunity käyttivät Field D*-algoritmia navigointiin. Tämä onkin todennäköisesti hyvä tapa käyttää algoritmia sillä se mahdollistaa hyvin suorat reitit pisteiden välillä mikä vähentää tarvittavaa kääntymistä mikä on hyvin tärkeä ominaisuus Marsissa navigoinnissa. Toisaalta Marsissa liikkuminen ei tapahdu nopeasti eikä nopeat muutokset ole muutenkaan toivottavia. [30] Field D* onkin mielestäni sopiva algoritmi tilanteisiin, jossa tarkkuus on nopeutta tärkeämpää. Siitä on mielestäni mahdollista tehdä vielä tarkempi paremmalla navigoinnilla, joka toimii tehokkaammin tilanteissa, jotka vaativat reitin uudelleenlaskentaa. Nykyinen toteutus on turhan yksinkertainen tällaiseen. Field D*:n toteutuksessa myös reitin määrittäminen g-arvojen perusteella oli paljon monimutkaisempaa koska reitit saattoivat kulkea solmujen välistä. Näiden reittien laskeminen osoittautui odotettua haastavammaksi. Field D* toimiikin parhaiten tilanteissa, joissa sillä on riittävän paljon aikaa laskea sekä mahdollisuus tuottaa suoria reittejä.

Sekä D* Liteä että Field D*:ä koski sama desimaalilukuihin liittyvä haaste. Koska molemmat algoritmit olettavat niiden heuristiikkojen olevan kokonaislukuja, desimaalilukuja palauttavat heuristiikat saivat algoritmin pysähtymään usein liian aikaisin desimaalilukuihin liittyvien epätarkkuuksien takia. Näin usein laskenta jäi kesken eikä algoritmi näin kyennyt löytämään reittiä maaliin. Kokonaislukuja palauttavat heuristiikat toimivat huomattavasti paremmin, mutta niidenkin kanssa suoritus jäi joskus harvoin kesken, jolloin avaimia vertailevaan funktioon lisättiin marginaali, joka esti desimaaliluvuista johtuvat epätarkkuudet. Näin suoritus jatkui loppuun saakka.

LSS-LRTA* oli reaaliaikaympäristöjen tarpeisiin toteutettu algoritmi. Se toimii parhaimmillaan hyvin nopeasti, mutta sen tuottamat reitit olivat poikkeuksellisen pitkiä ainakin Maze128 ja Room128-kartoissa. Näissä kartoissa oletus h-arvot eivät vastaa juurikaan todellisia, joten laskenta on haastavampaa. N-arvon vaikutus osoittautuikin karttariippuvaiseksi, joten sitä ei tulisi valita sen mukaan mitä testit jossain toisessa kartassa kertovat. N-arvolla havaittiin vaikuttavan olevan karttariippuvainen N-arvo, jolla sen suoritus on kaikkein nopein ja myös että sen suoritustarkkuus paranee N-arvoa kasvattamalla tietyissä kartoissa. Koska N-arvon kasvattaminen lisää suoritusaikoja voidaan sanoa, että N-arvo tulisi asettaa niin pieneksi kuin mahdollista niin että hakujen tarkkuus on riittävä

tarkoitukseen nähden. Suurilla N-arvoilla sen kasvatuksesta saatava hyöty pienenee.

Nopeutensa ansiosta LSS-LRTA* soveltuu hyvin reaaliaikaympäristöihin kuten tietokonepeleihin. Sen suorittamat haut ovat hyvin nopeita pienillä N-arvoilla. Lisäksi mahdollisuus nostaa N-arvoa antaa mahdollisuuden lisätä tarkkuutta tarvittaessa. Lisäksi se oli myös hyvin toimintavarma. Toteuttaminen oli hiukan ARA*:ä monimutkaisempaa, mutta ei yhtä hankalaa kuin Field D*:n.

LSS-LRTA* nopeutti hakuaan rajaamalla sen vain pieneen osaan ympärillään. ARA* nopeutti hakuaan vuorostaan käyttämällä ϵ-arvoa kertomaan heuristiikkafunktion näin tekemällä siitä epätarkemman mutta toisaalta huomattavan paljon nopeamman. Koska se kykenee toisaalta parantamaan tarkkuuttaan, jos sille annetaan aikaa mutta toisaalta laskemaan epätarkemman reitin hyvin nopeasti se soveltuu mielestäni hyvin tilanteisiin korvaamaan A*:n. Erityisesti tilanteissa, joissa suuri ϵ ei merkittävästi vaikuta lasketun reitin pituuteen ARA* on erinomainen valinta korvaamaan A*, koska laskennan nopeus paranee merkittävästi. Tämä on tietysti hyvin karttariippuvaista. ARA* on myös hyvin helppo toteuttaa eikä sen toiminnassa havaittu minkäänlaista epävarmuutta. Se oli neljästä testatusta algoritmista toimintavarmin sillä valmis algoritmi ei epäonnistunut reitin löytämisessä kertaakaan.

6.2 Jatkotutkimus

Robotin havainnointialue piti alun perin olla yksi tutkittavista attribuuteista ja se toteutettiin jo suoritusalustalla. Se jätettiin kuitenkin pois, ettei muuttujia tulisi liikaa ja tutkielman laajuus kasvaisi liian suureksi. Tehdyissä testeissä kävi ilmi, että havainnointialue vaikuttaa merkittävästi robotin toimintaan ympäristössä, joten tätä voisi tutkia myös tarkemmin. Myös erilaisia muotoja havainnoida ympäristöä voisi tutkia. Esim. robotti voisi nähdä vain eteensä, tai ei näkisi esteiden läpi.

Tässä tutkielmassa Field D*-algoritmin toimintaa yksinkertaistettiin, jotta se toimisi valitussa toimintaympäristössä ja sen toteutuksesta ei tulisi turhan monimutkainen. Field D*:een kuuluva tuki eriarvoisille soluille voitaisiin lisätä kohtuullisen vähällä vaivalla muihin algoritmeihin ja tutkia miten muut algoritmit toimivat ympäristössä, jossa Field D*:n pitäisi toimia parhaiten. Tutkimuksessa voitaisiin esim. poistaa esteet kokonaan ja antaa jokaiselle solulle jokin arvo, joka on kustannus siirtyä tähän soluun. Nämä arvot ovat tuntemattomia, kunnes agentti on ne havainnut.

Mahdollisuus esitellä myös kokonaan uusi algoritmi, joka lisäisi ARA*:n ominaisuu-

det Field D*:een olisi yksi mahdollinen tutkimusaihe. AD* on algoritmi, joka yhdistää ARA*:n ja D* Liten ominaisuudet samaan algoritmiin. Voitaisiin tutkia voiko AD*:een lisätä myös any-angle-ominaisuus, jolloin saataisiin algoritmi, joka olisi inkrementaalinen, anytime sekä any-angle.

Myös tuntemattomien esteiden määrää voitaisiin vaihdella. Nyt robotti ei tunne alueesta mitään mutta olisi myös mahdollista tutkia miten tuntemattomien esteiden määrä vaikuttaa suoritukseen. Lisäksi kuten todettua tutkielman alussa A*-variantteja on esitetty vuosien saatossa suuri määrä ja niissä kaikissa riittää valtava määrä tutkittavaa.

Viitteet

[1] Jukka Hietaniemi *MALLINTAMINEN JA SEN KÄYTTÖ PALOTEKNIIKASSA* VTT Rakennus- ja yhdyskuntatekniikka 2005 https://www.researchgate.net/publication/242400937_MALLINTAMINEN_JA_SEN_KAYTTO_PALOTEKNIIKASSA

[2] *Ilmatieteenlaitos - Tekniikka ja menetelmät* Ilmatieteenlaitos, Luettu 21.12.2020 https://www.ilmatieteenlaitos.fi/mallinnus

[3] *Stanford Encyclopedia of Philosophy* Stanford University https://plato.stanford.edu/entries/models-science/

[4] Lisa Velden *IDP Project of Lisa Velden* Chair M9 of Technischen Universität München https://www-m9.ma.tum.de/graph-algorithms/spp-a-star/index_en.html

[5] Jorma Kyppö *VERKKOTEORIAA* Jyväskylän yliopisto http://users.jyu.fi/~jorma/verkot.htm

[6] Pertti Koivisto, Riitta Niemistö *Graafiteoriaa* Tampereen yliopisto https://coursepages.uta.fi/mttma8/wp-content/uploads/sites/43/2017/11/GraafiteoriaLuennot17.pdf

[7] Xiao Cui and Hao Shi *A*-based Pathfinding in Modern Computer Games* IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, January 2011 https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games

[8] Hai-fei YU and Wei XIANG *Application of A-Star Algorithm in Robot for Catering Service Based on Laser Radar Navigation* 2017 International Conference on Electronic and Information Technology (ICEIT 2017) https://www.dpi-proceedings.com/index.php/dtcse/article/viewFile/19846/19334

[9] Eran Rippel, Aharon Bar-Gill and Nahum Shimkin *Fast Graph-Search Algorithms for General Aviation Flight Trajectory Generation* JOURNAL OF GUIDANCE, CONTROL, AND DYNAMICS Vol. 28, No. 4, July–August 2005 https://pdfs.semanticscholar.org/6821/c5f4616d2b7152a6a3b6f346c7dabcc2096b.pdf

[10] Rupin Dalvi, Adrian Suszko, Vijay S. Chauhan *Identification and Annotation of Multiple Periodic Pulse Trains Using Dominant Frequency and Graph Search: Applications in Atrial Fibrillation Rotor Detection* 2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC) https://ieeexplore.ieee.org/document/7591500

[11] Robert Sedgewick and Kevin Wayne *Algorithms, 4th Edition* Addison-Wesley Professional https://algs4.cs.princeton.edu/44sp/

[12] P. E. Hart, N. J. Nilsson and B. Raphael *A Formal Basis for the Heuristic Determination of Minimum Cost Paths* IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968 https://www.cs.auckland.ac.nz/courses/compsci709s2c/resources/Mike.d/astarNilsson.pdf

[13] Nils J. Nilsson *THE QUEST FOR ARTIFICIAL INTELLIGENCE - A HISTORY OF IDEAS AND ACHIEVEMENTS* Cambridge University Press https://ai.stanford.edu/~nilsson/QAI/qai.pdf

[14] Daniel R. Kunkle *Solving the 8 Puzzle in a Minimum Number of Moves:An Application of the A* Algorithm* Rochester Institute of Technology, Introduction to Artificial Intelligence October 8, 2001 https://web.mit.edu/6.034/wwwbob/EightPuzzle.pdf

[15] Steve Mussmann and Abi See *Graph Search Algorithms* Stanford University https://cs.stanford.edu/people/abisee/gs.pdf

[16] Wen Cao, Hui Shi, Shulong Zhu, Baoshan Zhu *Application of An Improved A* Algorithm in Route Planning* 2009 WRI Global Congress on Intelligent Systems https://ieeexplore.ieee.org/document/5208978

[17] Dennis Nii Ayeh Mensah, Hui Gao and Liang Wei Yang *Approximation Algorithm for Shortest Path in Large Social Networks* Algorithms 2020 https://www.mdpi.com/1999-4893/13/2/36/htm

[18] Antti Autere *EXTENSIONS AND APPLICATIONS OF THE A* ALGORITHM* Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 98, Espoo 2005 http://lib.tkk.fi/Diss/2005/isbn9512279487/isbn9512279487.pdf

[19] Ali Ghaffari *An Energy Efficient Routing Protocol for Wireless Sensor Networks using A-star Algorithm* Journal of applied research and technology, 2014 http://www.scielo.org.mx/pdf/jart/v12n4/v12n4a18.pdf

[20] David C. Wyld, Michal Wozniak, Nabendu Chaki, Natarajan Meghanathan, Dhinaharan Nagamalai *Trends in Network and Communications* International Conferences, NeCOM, WeST, WiMoN 2011, Chennai, India, July 15-17, 2011 https://link.springer.com/book/10.1007/978-3-642-22543-7

[21] Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai *A FAST ALGORITHM FOR FINDING BETTER ROUTES BY AI SEARCH TECHNIQUES* Proceedings of VNIS'94 - 1994 Vehicle Navigation and Information Systems Conference https://ieeexplore.ieee.org/document/396824

[22] Gurpreet Singh Shah, Ranbir Singh Batth, Simon Egerton *A Comparative Study on Efficient Path Finding Algorithms for Route Planning in Smart Vehicular Networks* International Journal of Computer Networks and Applications (IJCNA), Volume 7, Issue 5 https://www.ijcna.org/Manuscripts/IJCNA-2020-O-14.pdf

[23] Luis Henrique Oliveira Rios and Luiz Chaimowicz *A Survey and Classification of A* based Best-First Heuristic Search Algorithms* Advances in Artificial Intelligence – SBIA 2010 https://www.luisrios.eti.br/public/en_us/publications/downloads/classes.pdf

[24] Dave Ferguson and Anthony Stentz *Field D*: An Interpolation-based Path Planner and Replanner* Robotics Research. Springer Tracts in Advanced Robotics, vol 28. Springer, Berlin, Heidelberg http://robots.stanford.edu/isrr-papers/draft/stentz.pdf

[25] Sven Koenig and Xiaoxun Sun *Comparing Real-Time and Incremental Heuristic Search for Real-Time Situated Agents* Auton Agent Multi-Agent Syst 18, 313–341 (2009) http://idm-lab.org/bib/abstracts/papers/jaamas09.pdf

[26] Maxim Likhachev and Sven Koenig *D*Lite* AAAI-02 Proceedings http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf

[27] Maxim Likhachev and Sven Koenig *Fast Replanning for Navigation in Unknown Terrain* IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. XX, NO. Y, MONTH 2002 http://idm-lab.org/bib/abstracts/papers/tro05.pdf

[28] Maxim Likhachev, Geoff Gordon and Sebastian Thrun *ARA*: Anytime A* with Provable Bounds on Sub-Optimality* Proceedings of the 16th International Conference on Neural Information Processing Systems (NIPS'03). MIT Press, 767–774 https://papers.nips.cc/paper/2003/file/ee8fe9093fbbb687bef15a38facc44d2-Paper.pdf

[29] Amit Patel *Amit's Thoughts on Pathfinding* Stanford University https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

[30] *NASA* https://www.nasa.gov/mission_pages/mer/images/sol1162B-20080102a.html

[31] *8-Puzzle* https://i.ytimg.com/vi/sXgvdmzTiug/hqdefault.jpg