A*-algoritmi ja siihen pohjautuvat muistirajoitetut heuristiset reitinhakualgoritmit

Oskari Torikka

Pro gradu -tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Helmikuu 2015

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Kuopio
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Opiskelija, Oskari Torikka: A*-algoritmi ja siihen pohjautuvat muistirajoitetut heuristiset reitinhakualgoritmit
Pro gradu -tutkielma, 55 s.
Pro gradu -tutkielman ohjaajat: FT Matti Nykänen
Helmikuu 2015

A*-algoritmi on optimaalinen reitinhakualgoritmi, mutta sen ongelmana on liiallinen muistin käyttö. On olemassa A*-algoritmiin pohjautuvia reitinhakualgoritmeja, jotka käyttävät vähemmän muistia kuin A*-algoritmi. Tässä tutkielmassa käydään läpi A*-algoritmin lisäksi IDA*-, IEA*- ja SMA*-algoritmit. Edellä mainitut algoritmit käyttävät heuristista arviota reitinhaussa. Heuristinen arvio reitinhaussa tarkoittaa arvioitua etäisyyttä tietystä pisteestä maaliin. Heuristisen arvion voi toteuttaa eri tavalla eri ongelmia varten. Tutkielmassa käydään läpi kolme erilaista heuristiikkaa koskien 15-puzzlea. 15-puzzle on yleisesti käytetty testausympäristö tietojenkäsittelytieteissä esimerkiksi reitinhakualgoritmeille.

Tutkielman tulokset on saatu simuloimalla algoritmien toimintaa Java-pohjaisella ohjelmalla, jonka muisti on rajoitettu. Simuloinnin tuottamista tuloksista käy hyvin ilmi A*-algoritmin liiallinen muistin käyttö, ja kuinka A*-algoritmi ei selviydy vaikeimmista 15-puzzlen tapauksista sen takia. Tehokkuudeltaan A*-algoritmi on tässä tutkielmassa käsitellyistä algoritmeista paras, mutta muistin käytön kannalta toisessa ääripäässä oleva IDA*-algoritmi on lähes yhtä hyvä. IEA*-algoritmi parantaa IDA*-algoritmin tehokkuutta, mutta vain marginaalisesti. SMA*-algoritmi ei toimi lähellekään niin tehokkaasti kuin muut algoritmit, vaikka teoriassa idea onkin nerokas.

Avainsanat: A*-algoritmi, IDA*-algoritmi, IEA*-algoritmi, SMA*-algoritmi, reitinhaku

ACM-luokat (ACM Computing Classification System, 1998 version): F.2.2, I.2.1, I.2.8

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Kuopio
School of Computing
Computer Science

Student, Oskari Torikka: The A* search algorithm and memory-bounded heuristic pathfinding algorithms based on it
Master's Thesis, 55 p.
Supervisors of the Master's Thesis: PhD Matti Nykänen
February 2015

The A* search algorithm is an optimal pathfinding algorithm, but it suffers from excessive use of memory. There are algorithms based on A* which use less memory than A*. This thesis reviews A* and three memory-bounded algorithms IDA*, IEA* and SMA*. All aforementioned algorithms use a heuristic estimate. This heuristic estimate in pathfinding means the distance between a point and the goal. There are many different problems and many different ways in which a heuristic estimate can be calculated; this thesis mentions three methods concerning the 15-puzzle. The 15-puzzle is a common testing environment used for example in computer science for pathfinding algorithms.

The results presented in this thesis have been gathered by simulating the algorithms with a Java based application with limited memory. The results gathered by the simulation show the excessive use of memory by A*, and how A* cannot solve the more difficult cases of the 15-puzzle because of that. A* is the most efficient algorithm of the ones reviewed in this thesis. IDA*, which is the opposite of A* in respect of memory use, was almost as good as A*. IEA* improves the efficiency of IDA*, but only slightly. SMA* is not nearly as efficient as the other algorithms, even though the algorithm is ingenious in theory.

Keywords: A*, IDA*, IEA*, SMA*, pathfinding

CR Categories (ACM Computing Classification System, 1998 version): F.2.2, I.2.1, I.2.8

Esipuhe

Ensimmäiseksi haluaisin kiittää professori Matti Nykästä mielenkiintoisesta tutkielmaaiheesta sekä motivoivasta ohjauksesta. Hienoa tämän tutkielman kirjoittamisessa oli se, että pystyin toteuttamaan myös erikoistyöni tämän tutkielman ohella. Haluaisin myös kiittää professori Pekka Kilpeläistä tarkastajana toimimisesta, sekä Mari Miettistä yleisenä tukihenkilönä olemisesta.

Kuopiossa 24.2.2015

Oskari Torikka

Lyhenneluettelo

FIFO	First In, First Out; Jonon kaltainen tietorakenne
IDA*	Iterative-Deepening A*
IEA*	Iterative-Expansion A*
LIFO	Last In, First Out; Pinon kaltainen tietorakenne
MPD	Manhattan pair distance (Manhattan-parietäisyys); Eräs heuristinen metodi
SMA*	Simplified Memory-Bounded A*

Sisältö

1 Johdanto **1**

2 Heuristinen etsintä **3**

2.1 Hammingin etäisyys 6

2.2 Manhattan-etäisyys 7

2.3 Manhattan-parietäisyys 7

3 Ongelma (15-puzzle) **9**

4 Algoritmit **11**

4.1 A* 11

4.1.1 Toiminta 12

4.1.2 Muistin käyttö 15

4.1.3 Algoritmin toteuttaminen käytännössä 15

4.2 IDA* 17

4.2.1 Toiminta 17

4.2.2 Muistin käyttö 21

4.2.3 Algoritmin toteuttaminen käytännössä 21

4.3 IEA* 21

4.3.1 Toiminta 22

4.3.2 Muistin käyttö 25

4.3.3 Algoritmin toteuttaminen käytännössä 25

4.4 SMA* 25

4.4.1 Toiminta 26

4.4.2 Algoritmin toteuttaminen käytännössä 31

5 Tulosten yhteenveto **37**

5.1 A* ja IDA* 37

5.2 IDA* ja IEA* 41

5.3 SMA* 44

6 Pohdinta ja yhteenveto **48**

Viitteet **54**

1 Johdanto

Reitinhaku tarkoittaa useimmiten lyhimmän reitin etsimistä paikasta A paikkaan B. Joskus voidaan myös etsiä nopeinta reittiä, joka ei välttämättä ole lyhin. Reitinhakualgoritmit ovat siis algoritmeja, jotka etsivät annetun datan perusteella joko lyhimmän, nopeimman, tai muulla tavalla määritellyn reitin lähtöpaikasta määränpäähän. Joku voi esimerkiksi haluta löytää mahdollisimman helposti kuljettavan reitin, toinen voisi haluta kulkea mahdollisimman pitkään pieniä polkuja pitkin, ja kolmas voisi mahdollisesti haluta välttää moottoriteitä omalla reitillään. Nykyisin esimerkiksi autonavigaattoreissa sekä älypuhelimien GPS-sovelluksissa käytetään jonkinlaisia reitinhakualgoritmeja halutun reitin löytämiseksi. Mobiililaitteiden rajoitettu laskentateho vaatii tehokkaita ja vähän resursseja käyttäviä reitinhakualgoritmeja.

Suurin osa arkipäiväisistä reitinhakuongelmista pystytään nykyään laskemaan lähes millä tahansa kotitietokoneella, varsinkin kun tietokoneiden laskentateho ja muistikapasiteetti kasvavat jatkuvasti. On kuitenkin olemassa laskennallisesti erittäin vaikeita ongelmia, joihin reitinhakualgoritmeja voidaan soveltaa. Näiden ongelmien hakuavaruuksien koot voivat kasvaa jopa eksponentiaalisesti ongelman syötteen suhteen, jolloin suurinkaan muistikapasiteetti ei välttämättä riitä niiden ratkaisemiseen. Toisaalta on olemassa myös monimutkaisia reitinhakuongelmia (Toth & Vigo, 2001), joita perinteiset reitinhakualgoritmit eivät pysty ratkaisemaan ilman suuria muutoksia. Esimerkiksi logistiikkayritys, joka tekee tavarakuljetuksia ympäri Suomea, joutuu laskemaan muutakin kuin vain lyhimmät reitit eri toimipisteiden välille. Yrityksen on myös otettava huomioon kuinka monta rekkaa vaaditaan tavaroiden kuljettamiseen, kuinka paljon polttoainetta matkoihin vaaditaan, voiko sama rekka toimittaa tavaraa useampaan paikkaan saman matkan varrella ja niin edelleen. Tässä tutkielmassa pysyttäydytään kuitenkin vain lyhimmän reitin etsimisessä eri algoritmien avulla.

Jatkossa käytetään termejä *verkko*, *solmu* ja *kaari*. Verkko G koostuu solmujen ja kaarien joukoista V ja E (Tarjan, 1983). Kaari määrittelee parin solmuja, joiden välillä verkossa voi kulkea. Jokaiselle kaarelle on myös määriteltävä *paino*. Painolla tarkoitetaan kaaren kulkemisen kustannusta, joka on yleensä matka tai aika. Reitinhaussa verkko kuvaa esimerkiksi tieverkostoa, jossa kaaret kuvaavat teitä, ja solmut risteyksiä sekä päätepisteitä. Tällä tavoin voidaan eri algoritmeissa esittää haluttu data, sekä laskea siitä käyttäjän haluamat reitit. Yksi vaihe reitinhaussa on siis muuttaa kartat algoritmien ymmärtämiksi verkoiksi. Verkot voivat olla hyvinkin kattavia, esimerkiksi

kaarille voidaan asettaa niiden pituuksien lisäksi mahdollisia rajoituksia tai muuta tärkeää tietoa. Jokin kaari voidaan määritellä moottoritieksi, toinen kaari pyörätieksi ja niin edelleen. Solmuihin voidaan esimerkiksi tallettaa tietoa risteyksen liikennemerkeistä. Kaikissa verkoissa tulisi kuitenkin vähintään olla tieto siitä, mitkä solmut on yhdistetty toisiinsa ja mitkä ovat eri kaarien painot. Mikäli kaarille ei ole määritelty erikseen painoa, voidaan kaikille verkon kaarille määritellä sama vakiopaino.

Verkko voi olla myös abstrakti. Abstrakti verkko ei välttämättä koskaan ole olemassa kokonaisuudessaan, mutta sitä voidaan silti käyttää. Tällainen verkko voidaan luoda tiettyjen sääntöjen mukaan, ja ne voivat olla jopa äärettömän kokoisia. Esimerkiksi ristinolla rajattomalla pelialueella sisältää äärettömän monta erilaista pelitilannetta, kun taas shakissa on olemassa äärellinen määrä eri tiloja. Shakissa on kuitenkin olemassa ääretön määrä erilaisia siirtojonoja, mikäli toistuvat tilat ovat sallittuja. Siirroista voidaan kuitenkin luoda abstrakti verkko rakentamalla verkkoa siirto kerrallaan lähtötilanteesta eteenpäin, jolloin jokainen tilanne on verkon solmu, ja jokainen siirto on kaari uuteen solmuun. Tämän tutkielman tulokset on saatu simuloimalla juuri tällaista abstraktia verkkoa eri algoritmien suoritusten mittaamiseksi. Simulaattorin ohjelmoinnissa on käytetty ohjelmointikielenä Javaa (Oracle, 2014).

Luvussa 2 selitetään heuristiikan käyttäminen reitinhaussa, sekä esitetään muutama esimerkki sen toteuttamisesta. Luku 3 syventyy 15-puzzleen ja sen kiinnostaviin ominaisuuksiin tietojenkäsittelytieteen kannalta. Luku 4 esittelee A*-algoritmin sekä siihen pohjautuvat muistirajoitetut reitinhakualgoritmit. Luvussa 5 on esitetty simuloinnin avulla saadut tulokset sekä algoritmien vertailu. Luku 6 päättää tutkielman tulosten pohdinnalla.

2 Heuristinen etsintä

Heuristiikan (Pólya, 1957 s. 112) käsitettä ei ole tarkasti määritelty. Se on kuulunut osana niin logiikkaan, filosofiaan kuin psykologiaankin. Heuristiikka määritellään usein löytämiseksi tai keksimiseksi. Heuristisesti johdetun päätelmän ei tarvitse olla täysin tarkka, vaan se voi olla yksi mahdollinen vaihtoehto, kunnes täydellinen vastaus saavutetaan. Heuristiikkaa voisikin pitää eräänlaisena valistuneena arvauksena, joka johdattaa esimerkiksi ongelmanratkaisussa eteenpäin, mutta ei välttämättä anna täysin oikeaa vastausta.

Perinteiset epäheuristiset reitinhakualgoritmit (Cormen & al., 2009) kuten leveyshaku, syvyyshaku tai Dijkstran algoritmi ovat "epätietoisia", eli ne tuntevat vain ne solmut, jotka on jo käyty läpi, sekä niihin kuuluvat kaaret ja niiden painot. Ne käyvät solmuja läpi tiettyjen sääntöjen mukaan, kunnes maalisolmu löytyy. Tästä syystä kyseiset algoritmit tekevät paljon turhaa työtä, sillä ne eivät tiedä maalisolmun sijainnista mitään. Esimerkiksi lyhimmän reitin löytämiseksi Kuopiosta Helsinkiin olisi aivan turhaa lähteä etsimään polkua pohjoisesta, sillä Helsinki sijaitsee Kuopiosta etelään. Epäheuristiset reitinhaut joutuisivat käymään läpi ison osan Suomen tieverkosta ennen kuin löytäisivät lyhimmän reitin.

Tietojenkäsittelytieteissä on paljon erilaisia ongelmia, joita ei voida ratkaista perinteisin keinoin mielekkäässä ajassa. Täydellisen ratkaisun, eli kaikkien vaihtoehtojen läpikäyminen ja niistä parhaan valitseminen, tai edes yhden ratkaisun löytäminen voi joissakin tapauksissa viedä tehokkaaltakin tietokoneelta useita vuosia. Tällaiset ongelmat tulisi kuitenkin pystyä ratkaisemaan paljon nopeammin. Heuristiikan käyttäminen voi nopeuttaa ratkaisun löytymistä huomattavasti, mutta ratkaisu ei tällöin välttämättä ole kaikkein paras. Esimerkiksi kauppamatkustajan ongelmassa ahne algoritmi löytää usein kelvollisen ratkaisun lyhyessä ajassa, mutta optimaalisen ratkaisun löytäminen on NP-vaikea ongelma (Cook, 2012). Joissakin tapauksissa heuristiikka tarjoaa periaatteessa tasapainoa ratkaisun optimaalisuuden sekä sen löytämiseen käytetyn ajan väliltä. Monissa tapauksissa täydellistä tai parasta ratkaisua ei välttämättä tarvita, vaan riittävän hyvä ratkaisu kelpaa. Usein on mielekkäämpää löytää lähes lyhin reitti muutamassa sekunnissa, kuin löytää lyhin reitti usean tunnin jälkeen. On kuitenkin tärkeää huomioida, että epäoptimaalisen ratkaisun tuottava algoritmi ei välttämättä aina ole heuristinen, ja että heuristinen algoritmi ei aina tuota epäoptimaalista ratkaisua. On olemassa ns. kelvollisia heuristiikkoja, jotka nopeuttavat algoritmin toimintaa, mutta

säilyttävät kuitenkin ratkaisun optimaalisuuden. Heuristiikan kelvollisuuteen palataan hieman myöhemmin tässä luvussa. Tärkeää kuitenkin on, että ratkaisun löytämiseen ei menisi turhan kauan aikaa.

Heuristiikka ei välttämättä ole aina itsenäinen prosessi ratkaisun löytämiseksi, vaan sitä voidaan myös käyttää yhdessä jonkin algoritmin kanssa tehokkuuden parantamiseksi. Esimerkiksi reitinhaussa heuristiikkaa hyödynnetään arvioimalla jäljellä olevaa matkaa tietystä pisteestä maaliin. Kyseessä olevan arvion ei tarvitse olla tarkka algoritmin toiminnan kannalta, mutta tarkempi heuristiikka parantaa algoritmin tehokkuutta enemmän kuin epätarkka heuristiikka. Matkan arviointia ei voida kuitenkaan suoraan käyttää esimerkiksi reppuongelman kohdalla, vaan siihen tulisi kehittää täysin oma heuristiikka, tai ongelmaa pitäisi muokata heuristiikalle sopivaksi. Heuristiikkaa voidaankin pitää ongelmakohtaisena tietona, jota voidaan käyttää hyväksi.

Eräs tärkeä vaatimus (Russell & Norvig, 2009 ss. 94–95) heuristiikalle reitinhaussa on se, ettei se saa koskaan arvioida jäljellä olevaa matkaa mistään solmusta maalisolmuun liian pitkäksi, vaan aina mieluummin liian lyhyeksi, mikäli tarkkaa arvoa ei ole tiedossa. Mikäli jäljellä oleva matka on liioiteltu, ei voida olla varmoja, että löydetty polku on lyhin. Koska kahden pisteen lyhin mahdollinen etäisyys on suora linja, se ei voi koskaan olla liian suuri arvio. Tästä syystä on mahdollista käyttää muun muassa maantieteellisiä etäisyyksiä arvioinnin perusteena. Heuristisen funktion sanotaan olevan *kelvollinen* (*admissible*), mikäli se täyttää tämän vaatimuksen. Toinen heuristiikan tärkeä vaatimus, joka vaikuttaa algoritmin nopeuteen vahvasti, on sen *johdonmukaisuus* (*consistency*, *monotonicity*). Tätä vaatimusta selkeyttää kaava (1), jossa funktio $h(x)$ tarkoittaa solmun x heuristista arviota ja funktio $d(x, y)$ solmujen x ja y välistä etäisyyttä.

$$h(x) \leq d(x, y) + h(y) \tag{1}$$

Kaava (1) kuvaa *kolmioepäyhtälöä* (Cormen & al., 2009 s. 671), joka tarkoittaa sitä, että mikään kolmion sivun pituus ei voi olla pidempi kuin kahden muun sivun pituuden summa. Kolmio muodostuu tässä tapauksessa solmuista x ja y sekä niille yhteisestä maalisolmusta. Käytännössä tämä tarkoittaa sitä, että kuljetun matkan sekä arvioidun jäljellä olevan matkan summa ei voi pienentyä kuljettaessa solmusta toiseen, sillä se olisi ristiriidassa heuristiikan kelvollisuuden kanssa. Mikäli heuristiikka on kelvollinen, mutta ei johdonmukainen, algoritmin toiminta voi olla erittäin hidasta. Esimerkiksi A*-algoritmi saattaa reitinhaun aikana vierailla useasti samoissa solmuissa, jotka on käyty jo läpi. Tämä voi hidastaa algoritmin toimintaa pahimmissa tapauksissa jopa

eksponentiaalisesti.

Kaikki tässä tutkielmassa läpikäydyt algoritmit käyttävät solmujen painottamiseen kaavassa (2) esitettyä $f(x)$-funktiota, joka määrittelee kuinka lupaava solmu on. Mitä pienempi solmun $f(x)$-arvo on, sitä lupaavampi se on. Funktio koostuu $g(x)$-arvosta, joka tarkoittaa matkaa lähtösolmusta solmuun x, sekä $h(x)$-arvosta, joka on heuristinen arvio siitä, kuinka pitkä matka solmusta x on maalisolmuun. Solmun $f(x)$-arvoa voikin kuvailla siten, että mikäli reitti kulkee solmun x kautta, sen pituus on vähintään $f(x)$-arvon verran. Vaatimuksena kuitenkin on, että heuristinen arvio $h(x)$ on kelvollinen.

$$f(x) = g(x) + h(x) \tag{2}$$

Eräs yksinkertainen tapa nopeuttaa reitinhakualgoritmien toimintaa on heuristisen arvion painottaminen (Thayer & Ruml, 2008). Tämä tarkoittaa sitä, että funktio $f(x)$ onkin kaavassa (3) esitetyssä muodossa. Kaavassa (3) oleva muuttuja w painottaa heuristista arviota.

$$f(x) = g(x) + w * h(x) \tag{3}$$

Koska kaikki kelvolliset heuristiset arviot ovat aina pienempiä tai yhtä suuria kuin todellinen etäisyys, painotettu heuristiikka antaa mahdollisesti tarkemman arvion todellisesta etäisyydestä, mikäli $w > 1$. Tässä kuitenkin on vaarana se, että painotettu heuristiikka ylittää todellisen etäisyyden maaliin, jolloin se ei ole enää kelvollinen. Joissakin tapauksissa kaikkein lyhimmän reitin löytyminen ei ole tärkeintä, kunhan jokin kelvollinen reitti löytyy nopeassa ajassa. Löydetty reitti on korkeintaan w kertaa huonompi kuin lyhin mahdollinen reitti (Pearl, 1984).

Uusien heuristiikkojen suunnittelussa tavoitteena on saada tarkempi arvio jäljellä olevasta etäisyydestä maaliin. Tämän tulisi onnistua käyttämättä liikaa prosessointiaikaa heuristisen arvioin laskemiseen, sillä se voi hidastaa reitinhakua huomattavasti, koska heuristinen arvo lasketaan jokaiselle solmulle erikseen. Huomattavasti tarkempi heuristiikka kuitenkin vähentää läpikäytyjen solmujen määrää jopa eksponentiaalisesti, jolloin heuristisen arvion laskemiseen voi uhrata enemmän aikaa. Mikäli olisi olemassa ns. "täydellinen heuristiikka", reitinhakualgoritmien tehokkuus paranisi huomattavasti. Yksi tapa toteuttaa täysin tarkka heuristiikka olisi suorittaa uusi reitinhaku jokaisen solmun kohdalla, jolloin palautuva heuristinen arvo olisi aina tarkka. Tässä tapauksessa käytetty aika heuristisen tarkkuuden parantamiseen ei kuitenkaan olisi mielekästä.

Tässä tutkielmassa käytetään *15-puzzlea* algoritmien suoritusten mittaamiseksi. Se on

yksinkertainen pulmapeli, jossa on tarkoituksena siirtää 4×4-kokoisella laudalla numeroidut palat oikeille paikoilleen käyttäen tyhjää ruutua apuna siten, että jonkin viereisen ruudun voi siirtää tyhjän ruudun tilalle. Palojen nostaminen tai siirtäminen siten, että pala liikkuu vinottain tai toisen palan yli ei ole sallittua. Kuvassa 1 on kaksi erilaista asetelmaa 15-puzzlesta.

5	4	8	7
1	9	2	3
15	6	14	12
10	13	11	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Kuva 1: Vasemmalla puolella oleva 15-puzzle on epäjärjestyksessä. Oikealla puolella oleva 15-puzzle on ratkaistu.

Reitinhaku ja lyhimmän reitin löytäminen 15-puzzlen tapauksessa tarkoittavat pienimmän siirtomäärän löytämistä, jolla epäjärjestetyn 15-puzzlen saa ratkaistua. Sen voi ajatella myös siten, että mitä reittiä tyhjän ruudun täytyy kulkea laudalla, jotta palat siirtyisivät oikeille paikoilleen. Heuristinen funktio $h(x)$ pyrkii 15-puzzlen tapauksessa arvioimaan mahdollisimman tarkasti, kuinka monta siirtoa vähintään vaaditaan, jotta kaikki palat olisivat oikeilla paikoillaan.

Tästä eteenpäin heuristiikalla tarkoitetaan arviota vähintään tarvittavista siirroista 15-puzzlen ratkaisemiseksi, ja sitä kuvaa funktio $h(x)$. Luvuissa 2.1, 2.2 ja 2.3 käydään läpi erilaisia metodeja heuristisen arvion mittaamiseksi 15-puzzlesta. Luvussa 3 käydään läpi 15-puzzle tarkemmin.

2.1 Hammingin etäisyys

Hammingin etäisyys (*Hamming distance*) on erittäin yksinkertainen ja nopea heuristinen arviointimenetelmä. Sen arvo (Edelkamp & Schrödl, 2011 s. 21) määräytyy sen

perusteella, kuinka moni pala ei ole oikealla paikallaan. Tämä tarkoittaa sitä, että heuristiikka saa korkeintaan arvon 15, mikäli mikään pala ei ole oikealla paikallaan. Heuristiikka on kelvollinen, sillä yhdellä siirrolla vain yksi pala voi siirtyä oikealle paikalleen, joten jos esimerkiksi kahdeksan palaa on väärissä kohdissa, tarvitaan vähintään saman verran siirtoja niiden saamiseksi oikeille paikoilleen. Hammingin etäisyys on kuitenkin liian optimistinen heuristiikka, sillä jotkin 15-puzzlen tapaukset voivat vaatia yli 60 siirtoa.

2.2 Manhattan-etäisyys

Manhattan-etäisyys (*Manhattan distance*) perustuu Hammingin etäisyyden ideaan, mutta parantaa sen tarkkuutta huomattavasti. Sen sijaan että laskettaisiin vain väärissä paikoissa olevien palojen lukumäärä, Manhattan-etäisyys (Edelkamp & Schrödl, 2011 s. 21) laskee väärissä paikoissa olevien palojen etäisyyden oikeille paikoilleen. Palan etäisyys määräytyy sen mukaan, kuinka monta siirtoa vaaditaan sen saamiseksi oikealle paikalleen. Manhattan-etäisyys on myös kelvollinen heuristiikka, sillä yhdellä siirrolla vain yksi pala voi siirtyä lähemmäksi omaa paikkaansa, joten arvio ei ole koskaan liioiteltu. On myös todennäköistä etteivät kaikki palat pääse oikeille paikoilleen suorinta reittiä, vaan joutuvat jossain vaiheessa kiertämään toisen palan.

2.3 Manhattan-parietäisyys

Manhattan-etäisyys on yleisesti tunnettu ja melko tarkka heuristiikka. Sen tarkkuutta on kuitenkin mahdollista parantaa, ja sen *Manhattan-parietäisyys* (*Manhattan pair distance*, *MPD*) tekeekin. Manhattan-parietäisyys (Bauer, 1994) toimii kuten Manhattan-etäisyys, mutta ottaa myös huomioon palojen mahdolliset törmäykset. Törmäys tapahtuu, mikäli kaksi palaa kuuluu samalle riville, mutta ovat väärässä järjestyksessä. Tämä tarkoittaa sitä, että toisen palan on liikuttava toiselle riville, jotta palat voivat ohittaa toisensa, ja palata jälleen takaisin oikealle riville. Jokaiselle törmäävälle parille tämä tarkoittaa vähintään kahta siirtoa. Törmäykset voivat tapahtua riveittäin tai sarakkeittain. Rajoituksena kuitenkin on, että yksi pala voi törmätä vain yhden eri palan kanssa. Mikäli rivillä tai sarakkeella on kolme väärässä järjestyksessä olevaa palaa, vain yhden tarvitsee väistyä muiden tieltä, jolloin kaksi ylimääräistä siirtoa on riittävä.

Myös MPD on kelvollinen heuristiikka, sillä se perustuu Manhattan-etäisyyteen ja kasvattaa heuristista arviota vain silloin, kun on varmaa, että lisäsiirtoja on pakko tehdä. MPD on myös vähintään yhtä tarkka heuristiikka kuin Manhattan-etäisyys, sillä se laskee Manhattan-etäisyyden sekä mahdolliset törmäykset. Mikäli törmäyksiä ei tapahdu, MPD saa täysin saman arvon kuin Manhattan-etäisyys. Mitä useampia törmäyksiä MPD löytää, sitä tarkemmin se osaa arvioida vaadittavien siirtojen määrän (Manhattan-etäisyys $\leq$ MPD $\leq$ lyhin reitti). Mittauksissa käytettiin tätä heuristiikkaa, sillä se on näistä kolmesta heuristiikasta tarkin, eikä se ole tarpeettoman raskas laskea.

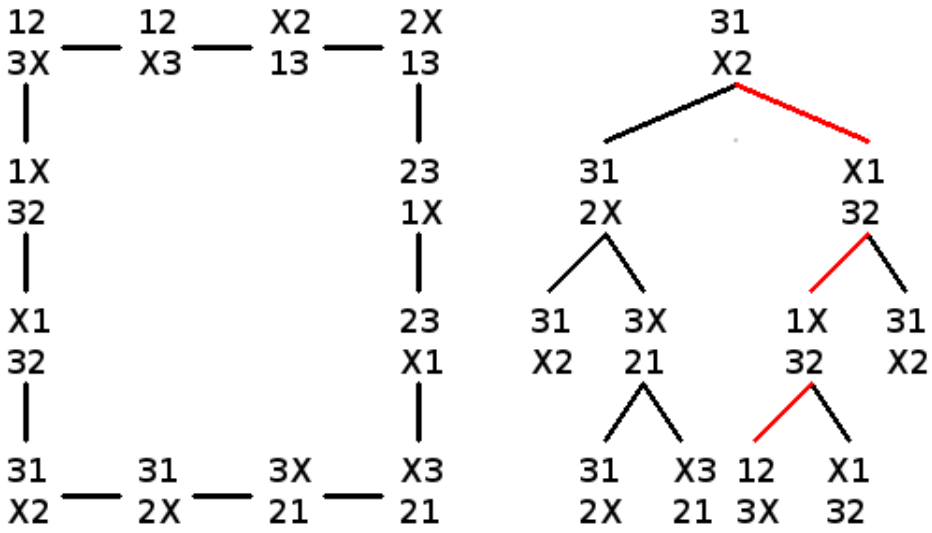
3 Ongelma (15-puzzle)

Noyes Chapmanin 1870-luvulla kehittämä 15-puzzle (Slocum & Sonneveld, 2006) on ongelma, jota käytetään vielä nykyäänkin esimerkiksi tietojenkäsittelytieteissä heuristisen reitinhakualgoritmien nopeuden mittaamiseen. Ongelmasta on myös olemassa eri kokoisia versioita, ja toinen yleinen tapaus on 3×3-kokoinen ruudukko (8-puzzle). Syyt 15-puzzlen suosioon tietojenkäsittelytieteissä ovat sen yksinkertaiset säännöt, helppo implementointi ja suuri hakuavaruus. Näistä syistä se toimii hyvänä alustana eri algoritmien tehokkuuksien mittaamiseen ja vertailuun. Vaikka itse ongelman ratkaiseminen onkin melko triviaalia, lyhimmän ratkaisun löytäminen on tunnetusti NP-vaikea ongelma (Bauer, 1994). Erilaisia alkuasetelmia 15-puzzlella on $16!$, joista kuitenkin vain puolet on ratkaistavissa, eli noin 10^{13}.

Ongelman koon kasvaessa sen vaikeustaso kasvaa eksponentiaalisesti. Nykyisillä tietokoneilla 3×3-kokoisen palapelin mikä tahansa ratkaistavissa olevista permutaatioista pystytään ratkaisemaan muutamissa millisekunneissa A*-algoritmilla, joka käyttää heuristiikkana Manhattan-etäisyyttä. Mikäli ongelman kokoa kasvatetaan 3×4-kokoiseksi taulukoksi, vaikeimpien ongelmien ratkaisemiseen menee muutamia sekunteja. 4×4-kokoisen taulukon vaikeimpien tapausten ratkaisemisessa kuluu muutamia minuutteja. Mikäli ongelman kokoa kasvatetaan tästä suuremmaksi, on hyvin todennäköistä ettei A*-algoritmi pysty enää löytämään lyhintä reittiä sen suuren muistivaatimuksen takia. Muistirajoitettujen reitinhakualgoritmien kohdalla 5×5-kokoinen taulukko (24-puzzle) on kuitenkin hyvin suosittu. Uusia heuristiikkoja suunnitellaan ja testataan nykyään pääosin 24-puzzlella (Korf & Taylor, 1996).

Jotta 15-puzzle olisi ratkaistavissa reitinhakualgoritmeilla, täytyy siitä ensin muodostaa solmuista ja kaarista koostuva verkko. Vaikka 15-puzzlella on äärellinen määrä erilaisia tiloja, niistä muodostuva verkko on erittäin suuri. Tällaista verkkoa ei kuitenkaan voi pitää muistissa kokonaisuudessaan, joten on käytettävä abstraktia verkkoa. Abstraktin verkon voi muodostaa 15-puzzlen tapauksessa siten, että jokainen taulukon permutaatio on yksi solmu, ja jokaiseen solmuun yhdistyy niin monta kaarta, kuin siinä tilassa on mahdollisia siirtoja. Kuvassa 2 on näkyvillä 2×2-kokoisesta taulukosta muodostettu verkko. Jokaisesta tilasta voi siirtyä vain kahteen muuhun tilaan johtuen taulukon pienuudesta. Reitinhakualgoritmille annetaan jokin epäjärjestyksessä oleva permutaatio, jonka jälkeen se etsii lyhintä reittiä siihen tilaan, jossa palat ovat oikeassa järjestyksessä. Lyhin reitti koostuu siis 15-puzzlen siirroista. Reitinhakualgoritmi

muodostaa abstraktia verkkoa sitä mukaa kuin on tarpeen, eli yhden siirron kerrallaan. Verkon kaaret on kaikki painotettu samanarvoisiksi, sillä jokaisesta 15-puzzlen tilasta toiseen pääsee yhdellä siirrolla. Jokainen tämän tutkielman reitinhakualgoritmi luo reitinhaun aikana hakupuun, joka on verkko ilman silmukoita. Kuvassa 2 on eräs mahdollinen tapaus hakupuusta. Punaisella merkityt kaaret kuvaavat pienintä siirtomäärää palapelin ratkaisemiseksi. Koska siirtoja voi tehdä loputtomasti, voi hakupuu teoriassa kasvaa äärettömäksi, mutta käytännössä muisti rajoittaa sen kokoa.



Kuva 2: Vasemmalla puolella on 2×2-kokoisen taulukon verkko kokonaisuudessaan, ja oikealla on yksi mahdollinen hakupuu. X kuvaa tyhjää ruutua.

Haarautumiskerroin (*branching factor*) kuvaa solmujen jälkeläisten lukumäärää, ja sen avulla esimerkiksi verkosta voidaan laskea muita tärkeitä arvoja. Solmuilla ei aina välttämättä ole samaa määrää jälkeläisiä, jolloin täytyy laskea haarautumiskertoimen keskiarvo. 15-puzzlen tapauksessa (Korf, 2000) solmuilla voi olla 2-4 jälkeläistä, tai 1-3 jälkeläistä mikäli edestakaiset liikkeet on poistettu laskuista. Naiivi tapa laskea 15-puzzlen haarautumiskerroin olisi käyttää kaavaa $(4 * 2 + 8 * 3 + 4 * 4)/16 = 3$, jossa siis lasketaan kaikkien mahdollisten ruutujen haarautumiskertoimet yhteen ja otetaan siitä keskiarvo. Mikäli edestakaiset liikkeet on poistettu, naiivi haarautumiskerroin olisi 2. Tässä on kuitenkin se oletus, että tyhjä ruutu esiintyisi laudan jokaisella ruudulla samalla todennäköisyydellä. Näin ei käytännössä ole, sillä keskimmäiset palat ovat paremmin edustettuna laudalla. Tästä syystä tarkempi asymptoottinen haarautumiskerroin on 2,13.

4 Algoritmit

Tässä luvussa käydään läpi A*-algoritmi sekä kolme siihen perustuvaa muistirajoitettua reitinhakualgoritmia. Kaikki esiteltävät algoritmit ovat "tietoisia", eli niillä on jokin hyödynnettävä tieto haettavasta reitistä. Tämä tieto on heuristinen arvio jäljellä olevasta matkasta maaliin. Kaikki algoritmit painottavat läpikäydyt solmut funktion $f(x)$ avulla, joka on esitelty kaavassa (2) sivulla 5.

Kukin algoritmi löytää lyhimmän reitin lähtöpisteestä maaliin, mikäli niiden välillä on olemassa yksi tai useampi reitti, ja mikäli algoritmin käyttämä heuristiikka on kelvollinen. Algoritmit toimivat staattisessa eli muuttumattomassa ympäristössä. Luvussa 4.1 käydään läpi A*-algoritmi, jonka jälkeen luvut 4.2, 4.3 ja 4.4 vastaavasti käsittelevät muistirajoitettuja algoritmeja IDA*, IEA* ja SMA*.

4.1 A*

A-algoritmia* pidetään tällä hetkellä yhtenä parhaista paras-ensin-reitinhakualgoritmeista, mikäli tarkoituksena on etsiä lyhin reitti pisteiden A ja B välille. A*-algoritmi on optimaalinen (Russell & Norvig, 2009 ss. 95–98), mikä tarkoittaa sitä, ettei ole olemassa toista algoritmia, joka löytäisi lyhimmän reitin pisteiden A ja B välille käymällä läpi vähemmän solmuja kuin A*-algoritmi, kun molemmilla on käytössään sama heuristiikka. Tämä johtuu siitä, että A*-algoritmi käy läpi kaikki solmut, joiden $f(x)$-arvo on pienempi kuin lyhimmän reitin pituus, sekä osan niistä solmuista, joiden $f(x)$-arvo on yhtä suuri kuin lyhimmän reitin pituus. Tällä tavoin voidaan olla varmoja, ettei ole olemassa lyhyempää reittiä kuin se, joka on löydetty. Jos jokin toinen algoritmi jättäisi käymättä läpi joitain solmuja, olisi mahdollista, että löydetty reitti ei olisi lyhin. Esimerkiksi A*-algoritmi saattaa löytää reitinhaun aikana pisteiden välille useita reittejä, mutta jatkaa kuitenkin etsimistä kunnes voidaan olla varmoja, että löydetty reitti on lyhin. A*-algoritmin toimintaa kuvaa tarkemmin luku 4.1.1 sekä sivulta 13 löytyvä pseudokoodi (algoritmi 1).

4.1.1 Toiminta

A*-algoritmi aloittaa etsimisen lähtösolmusta, joka on ensimmäinen laajennettava solmu. Laajentaminen tarkoittaa sitä, että jokainen laajennettavan solmun naapurisolmu lisätään A*-algoritmin ylläpitämään prioriteettijonoon, jota kutsutaan *avoimeksi listaksi*. Yleensä kun sanotaan A*-algoritmin käyvän läpi jonkin solmun, käytännössä tarkoitetaan sen solmun laajentamista. A*-algoritmi myös tallettaa kaikki läpikäydyt solmut. Läpikäytyjen solmujen joukkoa kutsutaan usein *suljetuksi listaksi*, vaikka sen toteutus ei käytännössä ole lista. Laajentamisen yhteydessä jokainen uusi solmu painotetaan käyttäen heuristista funktiota $f(x)$, joka määrittää solmujen järjestyksen prioriteettijonossa. Dijkstran algoritmi (Cormen & al., 2009 s. 658) käyttää myös prioriteettijonoa seuraavan laajennettavan solmun valintaan, mutta se painottaa solmut pelkästään $g(x)$-arvon mukaan. A*-algoritmin käyttämä $f(x)$-arvo ohjaa reitinhakua oikeaan suuntaan, kun taas Dijkstran algoritmi käy läpi verkkoa tasaisesti joka suuntaan.

Mikäli laajennettavan solmun naapurisolmu on jo lisätty suljettuun listaan, ei sitä tarvitse enää lisätä avoimeen listaan, sillä se on jo käyty läpi. Mikäli A*-algoritmin käyttämä heuristiikka on kelvollinen ja johdonmukainen, voidaan olla varmoja, ettei suljetussa listassa oleviin solmuihin voi löytyä enää lyhyempiä reittejä. Dijkstran algoritmi ei käytä suljettua listaa, mutta se toimii niin, ettei jo läpikäytyyn solmuun voi löytyä lyhyempää reittiä. Jos naapurisolmu on jo lisätty avoimeen listaan, verrataan sen vanhaa $f(x)$-arvoa uuteen. Mikäli nykyisestä solmusta laskettu $f(x)$-arvo on pienempi kuin aikaisempi arvo, voidaan avoimessa listassa olevan solmun arvo päivittää. Myös Dijkstran algoritmi päivittää solmujen painotuksia, mikäli lyhyempiä reittejä solmuihin tulee vastaan. Kun aloitussolmun kaikki viereiset solmut on lisätty avoimeen listaan, siirretään aloitussolmu suljettuun listaan, ja siirrytään tutkimaan seuraavaksi solmua, jolla on pienin $f(x)$-arvo. Solmujen läpikäymistä jatketaan kunnes avoimen listan ensimmäisenä alkiona on maalisolmu.

A*-algoritmin suorituksen aikana avoimeen listaan saatetaan lisätä maalisolmu (Edelkamp & Schrödl, 2011 s. 73), mutta reitinhaku ei kuitenkaan vielä keskeydy siinä vaiheessa. Mikäli avoimeen listaan lisätyllä maalisolmulla ei ole pienin $f(x)$-arvo, on mahdollista, että jokin pienemmän $f(x)$-arvon omaava solmu sisältää lyhyemmän reitin listassa olevaan maalisolmuun tai johonkin toiseen maalisolmuun. Useiden maalisolmujen tapauksissa (Russell & Norvig, 2009 s. 98) A*-algoritmin on vaikeampi määritellä lupaavimmat solmut, ja se joutuu siten todennäköisesti laajentamaan useampia

Algoritmi 1 A*-algoritmi, jonka parametreina ovat alku- ja maalisolmu

```
1:  procedure A*(alku, maali)
2:      suljettu ← ∅
3:      avoin ← {alku}
4:      while avoin ≠ ∅ do
5:          solmu ← POLL(avoin)                    ▷ Poista ja palauta paras solmu
6:          if solmu = maali then
7:              return reitti
8:          end if
9:          suljettu ← suljettu ∪ {solmu}
10:         for each solmun naapuri do
11:             if naapuri ∈ suljettu then
12:                 continue
13:             end if
14:             /*Lasketaan uusi g-arvo, jossa d on etäisyys solmujen välillä*/
15:             g_arvo ← g(solmu) + d(solmu, naapuri)
16:             if naapuri ∉ avoin then
17:                 g(naapuri) ← g_arvo
18:                 f(naapuri) ← g(naapuri) + h(naapuri)
19:                 avoin ← avoin ∪ {naapuri}
20:             else if g_arvo < g(naapuri) then          ▷ naapuri ∈ avoin
21:                 /*On löytynyt lyhyempi reitti samaan solmuun*/
22:                 g(naapuri) ← g_arvo
23:                 f(naapuri) ← g(naapuri) + h(naapuri)
24:                 Päivitä uusi reitti naapuriin
25:             end if
26:         end for
27:     end while
28:     return NOT FOUND
29: end procedure
```

solmuja kuin yhden maalisolmun tapauksessa. Tämä tutkielma kuitenkin keskittyy 15-puzzleen, jolla on vain yksi maalisolmu, joten useiden maalisolmujen tapaukset sivuutetaan. Vasta sitten kun maalisolmu on valittu laajennettavaksi, voidaan olla varmoja, että löytynyt reitti on lyhin mahdollinen. On kuitenkin mahdollista, että on olemassa vielä muitakin yhtä pitkiä reittejä lähtösolmusta maalisolmuun, mutta A*-algoritmi lopettaa etsinnän ensimmäisen sellaisen reitin löytyessä. Mikäli avoin lista tyhjenee ennen kuin maalisolmu on löytynyt, voidaan olla varmoja, ettei ole olemassa reittiä lähtösolmusta maalisolmuun. Maalisolmun $f(x)$-arvosta voidaan suoraan nähdä lyhimmän reitin pituus lähtösolmusta maalisolmuun. Mikäli on tarvetta saada selville koko polku, voidaan reitinhaun aikana jokaiselle solmulle määritellä viite edelliseen solmuun, josta nykyiseen solmuun on saavuttu. Tätä viitettä päivitetään suorituksen aikana, jolloin lopuksi viitteitä seuraamalla maalisolmusta päästään lähtösolmuun. On tärkeää muistaa päivittää viite uuteen solmuun, mikäli avoimessa listassa olevan solmun $f(x)$-arvoa päivitetään.

Mikäli A*-algoritmin $h(x)$-arvo on tilanteesta riippumatta aina 0, toimii A*-algoritmi samoin kuin Uniform-cost search -algoritmi (Russell & Norvig, 2009 s. 97), koska silloin vain $g(x)$-arvolla on merkitystä solmujen painotuksessa. Uniform-cost search on toiminnaltaan lähes täysin samanlainen kuin Dijkstran algoritmi. Ainoa ero näillä algoritmeilla on se, että Uniform-cost search -algoritmille määritetään maalisolmu, jonka löytyessä algoritmin toiminta loppuu, kun taas Dijkstran algoritmi etsii lyhimmät reitit verkon jokaiseen solmuun lähtösolmusta, ellei sille erikseen määritetä lopetuskriteeriä (esim. maalisolmu).

Toinen A*-algoritmin erikoistapaus toteutuu silloin, kun $h(x)$-arvo on aina äärettömän suuri, jolloin funktio $f(x)$ painottaa kaikki solmut saman arvoisiksi. A*-algoritmin toiminta samojen $f(x)$-arvojen tapauksessa riippuu toteutuksesta, joista yleisimmät ovat *FIFO* (*First In, First Out*) tai *LIFO* (*Last In, First Out*), eli jono- tai pinomenetelmät. Mikäli kaikki solmut omaavat saman $f(x)$-arvon, A*-algoritmi toimii kuten leveys- tai syvyyshaku toteutuksesta riippuen. Yleensä ei ole juurikaan merkitystä kumpaa menetelmää A*-algoritmi käyttää, sillä saman $f(x)$-arvon omaavia solmuja on usein suhteellisen vähän verrattuna solmujen kokonaismäärään. On myös mahdollista lajitella saman $f(x)$-arvon omaavat solmut niiden $g(x)$-arvon perusteella, mikä on marginaalisesti tehokkain tapa tasa-arvoisten solmujen läpikäymiseen.

Jos olisi olemassa heuristiikka, joka pystyisi arvioimaan täydellisesti jäljellä olevan matkan maalisolmuun, A*-algoritmi kävisi läpi vain ne solmut, jotka ovat lyhimmäl-

lä reitillä. Mikäli maalisolmuun olisi useita yhtä pitkiä reittejä, A*-algoritmin toiminta riippuisi sen toteutuksesta. Mikäli A*-algoritmi olisi toteutettu FIFO-periaatetta käyttäen, kävisi se läpi kaikkien lyhimpien reittien solmut, kun taas LIFO-periaatteella A*-algoritmi kävisi läpi vain yhden lyhimmän reitin solmut.

4.1.2 Muistin käyttö

A*-algoritmin tehokkuus perustuu osittain siihen, että se muistaa kaikki läpikäydyt solmut, jolloin sen ei tarvitse käydä samoja solmuja useaan kertaan läpi. Tämä (Russell & Norvig, 2009 ss. 98–99) on kuitenkin myös A*-algoritmin suurin heikkous, sillä suuren hakuavaruuden omaavissa ongelmissa on todennäköisempää, että käytettävä muisti loppuu kauan ennen kuin A*-algoritmi on löytänyt ratkaisun. A*-algoritmin tallettamien solmujen lukumäärä kasvaa eksponentiaalisesti lyhimmän reitin pituuden suhteen. Tästä syystä pienikin muutos reitin pituudessa vaikuttaa huomattavasti A*-algoritmin muistin käyttöön.

Heuristiikalla on myös suuri vaikutus A*-algoritmin muistin käyttöön. Täydellinen heuristiikka vähentäisi tarvittavan muistin määrää huomattavasti, jolloin A*-algoritmi pitäisi muistissa vain $d \times b$ solmua, kun d on lyhimmän reitin pituus ja b haarautumiskerroin. Mitä suurempi heuristisen arvion ero on todelliseen matkaan verrattuna, sitä enemmän solmuja A*-algoritmi joutuu käymään läpi, ja siten säilyttämään ne myös muistissa. On myös mahdollista, että ongelmalle on olemassa useita mahdollisia reittejä, joista osa on optimaalisia, ja osa lähes optimaalisia. Tämä tarkoittaa sitä, että A*-algoritmin hakupuu ei lähesty vain yhtä mahdollista kohdetta, vaan se levittyy useaan potentiaaliseen kohteeseen aiheuttaen huomattavan kasvun läpikäytävien solmujen määrässä.

4.1.3 Algoritmin toteuttaminen käytännössä

Suuren suosionsa ansiosta A*-algoritmista löytää hyvin helposti tietoa alan kirjallisuudesta sekä Internetistä. A*-algoritmia käytetään usein esimerkkinä heuristisen etsinnän, paras-ensin-algoritmin tai yleensäkin reitinhaun tapauksissa. Tästä syystä algoritmin toteuttaminen käytännössä oli lähes triviaalia. Vaikka A*-algoritmista on useita selkeitä pseudokoodeja sekä valmiita toteutuksia eri ohjelmointikielillä tarjolla, ei sen kirjoittaminen itse vaadi kovinkaan edistyneitä ohjelmointitaitoja. Ainoaksi ongelmak-

si muodostui tietorakenteen valitseminen avointa listaa varten. Javan tarjoamat tietorakenteet olivat aina jollakin tapaa puutteelliset prioriteettijonon toteuttamiseen. Javan eri luokkien puutteita käydään läpi myöhemmin tässä luvussa. Tutkielmassa päädyttiin lopulta käyttämään SMA*-algoritmia varten toteutettua tasapainotettua binääripuuta prioriteettijonon toteuttamiseen, jolloin pystyttiin välttämään Javan tietorakenteissa esiintyvät puutteet. A*-algoritmin käyttämää suljettua listaa varten käytettiin Javan omaa HashSet-luokkaa.

A*-algoritmin useimmin käyttämät avoimen listan operaatiot ovat solmun lisääminen, parhaan solmun palauttaminen ja poistaminen, sekä tietyn solmun etsiminen. Javan PriorityQueue-luokka oli nimensä ansiosta ensimmäinen ehdokas avoimen listan toteutukseen. Valitettavasti sen luokan contains-metodi oli hieman puutteellisesti toteutettu A*-algoritmin tarpeisiin. PriorityQueue-luokan contains-metodi palautti vastaukseksi kyllä tai ei, riippuen siitä oliko etsittävä solmu avoimessa listassa vai ei. Mikäli solmu on jo lisätty listaan, A*-algoritmin tarvitsisi tietää sen $g(x)$-arvo, jotta uutta solmua voitaisiin verrata listassa jo olevaan solmuun. Itse asiassa PriorityQueue-luokka ei tarjoa suoraa metodia tietyn alkion palauttamiselle ollenkaan. Tämä johtunee siitä, että Javan PriorityQueue-luokka perustuu kekorakenteeseen, jossa alkioiden sijainti ei ole tarkasti määritelty. Tärkeintä kekorakenteessa on pitää pienin (tai suurin) alkio aina ensimmäisenä. Luokasta puuttuvan metodin olisi voinut tehdä myös itse, mutta sen aikavaativuus olisi ollut vähintään lineaarinen, mikä on liian hidas A*-algoritmin tarpeisiin.

PriorityQueue-luokan puutteita yritettiin kiertää käyttämällä Javan HashMap-luokkaa samanaikaisesti. Tarkoituksena oli pitää yllä kahta avointa listaa, johon molempiin talletettaisiin kaikki solmut. Koska toteutettavan simulaattorin tarkoituksena oli mitata vain algoritmien nopeus sekä laajennettujen solmujen määrä, ei kahden avoimen listan ylläpitäminen vaikuttaisi saatuihin tuloksiin. Javan HashMap-luokka lupaa solmujen lisäämiseen sekä hakemiseen tietorakenteesta vakioaikaista aikavaativuutta. HashMap-luokan avulla pyrittiin kiertämään PriorityQueue-luokan puutteet, mutta käytännössä kahden tietorakenteen käyttäminen rinnakkain oli liian hidasta, vaikka HashMap-luokalla toteutetun avoimen listan operaatioiden pitikin toimia vakioajassa.

Javan TreeMap-luokka (Oracle, 2011) perustuu punamustaan puuhun (Cormen & al., 2009 s. 308), joka tarjoaa kaikki A*-algoritmin tarvitsemat metodit logaritmisessa ajassa. Ongelmaksi TreeMap-luokan kanssa muodostui sen käyttämä avain-arvo-kartoitus. Koska avoimeen listaan lisättyjen solmujen tulisi olla järjestetty $f(x)$-arvon mukaan,

olisi järkevää käyttää avaimena solmun $f(x)$-arvoa. Tämä ei kuitenkaan ole mahdollista, sillä Javan Map-luokasta perityt luokat kuten TreeMap eivät salli duplikaatteja. A*-algoritmin suorituksen aikana on kuitenkin erittäin todennäköistä, että monella solmulla on sama $f(x)$-arvo, joten TreeMap-luokkaa ei voi käyttää sellaisenaan avoimen listan toteutukseen. Toinen tapa olisi käyttää 15-puzzlen tilaa yhdessä $f(x)$-arvon kanssa luomaan yksikäsitteisiä avaimia. Tällöin solmut voitaisiin järjestää $f(x)$-arvon mukaan, eikä TreeMap-luokan rajoitus olisi enää este. Tässä kuitenkin ilmenee sama ongelma kuin aiemmin PriorityQueue-luokan kanssa, eli TreeMap-luokan tarjoama containsKey-metodi palauttaa vastaukseksi vain kyllä tai ei. Eräs mahdollinen tapa ohittaa TreeMap-luokan rajoitus olisi käyttää samaa $f(x)$-arvon ja 15-puzzlen tilan yhdistelmää sekä alkion avaimena että arvona. Tällöin TreeMap-luokan get-metodi palauttaisi A*-algoritmin tarvitseman tiedon. Valitettavasti tämä lähestymistapa keksittiin vasta ohjelman toteuttamisen jälkeen, eikä sitä ole kokeiltu käytännössä.

4.2 IDA*

Päinvastoin kuin A*-algoritmi, *IDA*-algoritmi* (*Iterative-Deepening A**) pyrkii käyttämään minimaalisen määrän muistia haun aikana. IDA*-algoritmi perustuu pääasiallisesti rajoitettuun syvyyshakuun (Edelkamp & Schrödl, 2011 s. 204), mutta käyttää rajan määrittämisessä hyväksi A*-algoritmin $h(x)$-funktiota. Koska $h(x)$-arvo on lähes aina pienempi kuin lyhimmän reitin pituus, joutuu IDA*-algoritmi kasvattamaan *hakurajaa* reitinhaun aikana. Hakurajan kasvattamisessa käytetään A*-algoritmin $f(x)$-funktiota. Vähäisen muistin käytön ja yksinkertaisen toteutuksen takia IDA*-algoritmi käy läpi solmuja erittäin nopeasti, eikä sen tarvitse käyttää monimutkaista tietorakennetta toimiakseen. IDA*-algoritmin toimintaa kuvaa tarkemmin luku 4.2.1 sekä sivulta 18 löytyvät pseudokoodit (algoritmit 2 ja 3).

4.2.1 Toiminta

IDA*-algoritmi (Korf, 1985) aloittaa etsimisen määrittämällä alustavan hakurajan käyttäen aloitussolmun heuristista $h(x)$-arvoa. Tämän jälkeen käynnistyy rajoitettu syvyyshaku (algoritmi 3). Kun perinteinen syvyyshaku käyttää syvyyden mittana kuljettua matkaa, IDA*-algoritmi ottaa huomioon myös heuristisen arvion. IDA*-algoritmin syvyyden mittana toimiikin siis solmun $f(x)$-arvo, sillä se sisältää jo kuljetun matkan

sekä arvioidun matkan maalisolmuun. Koska heuristinen arvio ei saa koskaan olla liian suuri, voidaan $f(x)$-arvoa käyttää syvyyden mittana, sillä mikäli solmun $f(x)$-arvo ylittää hakurajan, voidaan olla varmoja, ettei polkua maalisolmuun voida löytää ylittämättä hakurajaa. Jokaisen läpikäydyn solmun kohdalla tarkistetaan, onko se maalisolmu. Mikäli maalisolmu on löydetty, voidaan se palauttaa, ja muistissa olevat solmut määrittävät lyhimmän polun lähtösolmusta maalisolmuun.

Algoritmi 2 IDA*-algoritmi, jonka parametreina ovat alku- ja maalisolmu

```
1:  procedure IDA*(alku, maali)
2:      hakuraja ← h(alku)
3:      while hakuraja kasvaa do
4:          t ← ETSI(alku, 0, hakuraja, maali)          ▷ Ks. algoritmi 3
5:          if t = FOUND then
6:              return reitti
7:          end if
8:          hakuraja ← t
9:      end while
10:     return NOT FOUND
11: end procedure
```

Algoritmi 3 IDA*-algoritmin etsi-metodi

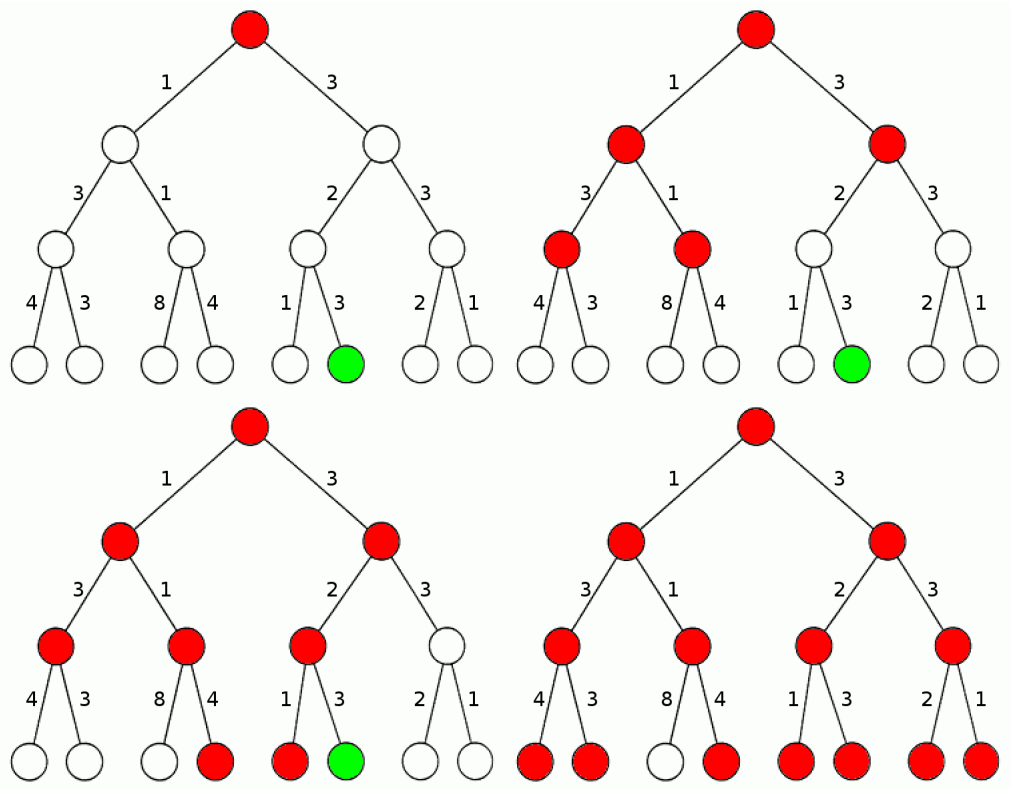
```
1:  procedure ETSI(solmu, g_arvo, hakuraja, maali)
2:      f_arvo ← g_arvo + h(solmu)
3:      if f_arvo > hakuraja then
4:          return f_arvo
5:      else if solmu = maali then
6:          return FOUND
7:      end if
8:      minimi ← ∞
9:      for each solmun naapuri do
10:         t ← ETSI(naapuri, g_arvo + d(solmu, naapuri), hakuraja, maali)
11:         if t = FOUND then
12:             return FOUND
13:         else if t < minimi then
14:             minimi ← t
15:         end if
16:     end for
17:     return minimi
18: end procedure
```

Mikäli lyhin polku on pidempi kuin hakuraja, ei IDA*-algoritmi löydä polkua ensimmäisellä haulla. Nimensä mukaisesti IDA*-algoritmi joutuu kasvattamaan hakurajaa jokaisen kierroksen jälkeen, jotta hakuraja olisi tarpeeksi suuri kattaakseen myös maa-

lisolmun. Hakurajaa kasvatetaan valitsemalla pienin $f(x)$-arvo niiden solmujen kesken, jotka ylittivät aiemman hakurajan edellisen kierroksen aikana. Hakurajan kasvattamisen jälkeen aloitetaan uusi kierros lähtösolmusta uudella hakurajalla. IDA*-algoritmi jatkaa näin kunnes hakuraja on kasvanut tarpeeksi suureksi, ja se pystyy löytämään lyhimmän polun lähtösolmusta maalisolmuun. Mikäli ei ole olemassa reittiä lähtösolmusta maalisolmuun, IDA*-algoritmi jatkaa reitinhakua kunnes hakuraja on tarpeeksi suuri kattaakseen kaikki verkon solmut. Tämän jälkeen ei ole enää solmuja joiden $f(x)$-arvo olisi suurempi kuin hakuraja, jolloin hakuraja ei enää kasva. Mikäli siis kierroksen jälkeen hakuraja ei kasva, voidaan olla varmoja siitä, ettei ole olemassa reittiä lähtösolmusta maalisolmuun, ja reitinhaku voidaan lopettaa. Koska IDA*-algoritmi käy läpi kaikki solmut joiden $f(x)$-arvo on pienempi kuin hakuraja, joudutaan jotkin solmut käymään läpi useaan kertaan ennen maalisolmun löytymistä. Viimeisellä kierroksella, eli silloin kun hakuraja on yhtä suuri kuin lyhimmän reitin pituus, reitinhaku lopetetaan heti kun maalisolmu on löytynyt. On siis mahdollista, että viimeisellä kierroksella ei tarvitse käydä läpi kaikkia solmuja, jotka kuuluvat syvyysrajoitettuun hakupuuhun. Ainoastaan silloin, jos heuristinen arvio on tarpeeksi tarkka, voi IDA*-algoritmi löytää lyhimmän polun vain yhdellä kierroksella. Tämä kuitenkin vaatisi täydellisen heuristiikan ollakseen mahdollinen.

Vaikka IDA*-algoritmi joutuukin käymään hakunsa aikana samoja solmuja useaan kertaan läpi kuten kuva 3 osoittaa, se ei välttämättä tarkoita että IDA*-algoritmi olisi hidas. IDA*-algoritmin etuna on sen yksinkertaisuus ja nopeus, eikä sen tarvitse kuluttaa aikaa erilaisten tietorakenteiden ylläpitämiseen. On myös tärkeää huomioida, että ensimmäisten kierrosten syvyysrajoitetut hakupuut ovat huomattavasti pienempiä kuin myöhempien kierrosten, sillä hakurajan kasvaessa IDA*-algoritmin läpikäymä syvyysrajoitettu hakupuu kasvaa eksponentiaalisesti. Tästä syystä ensimmäiset kierrokset on nopeampi käydä läpi, ja suurempi työ tehdään vasta viimeisten kierrosten aikana.

Eräs yksinkertainen tapa (Dillenburg & Nelson, 1993) nopeuttaa IDA*-algoritmin toimintaa on poistaa silmukat sen hakupuusta. Tämä tarkoittaa sitä, että jos jokin solmu on jo sen hetkisellä polulla, ei ole enää tarpeen käydä sitä uudestaan läpi, vaan se voidaan jättää tutkimatta. Eräs triviaali tapaus on yksinkertainen edestakainen liike kun solmusta A siirrytään solmuun B, jonka jälkeen solmusta B palataan takaisin solmuun A. Yksinkertaisin tapa poistaa tällainen silmukka on olla generoimatta solmulle jälkeläistä, mikäli laajennettavan solmun jälkeläinen on myös sen solmun isäsolmu. Solmun poistaminen tällä tavoin poistaa myös tarpeen tutkia koko solmun jälkeinen alipuu, mi-



Kuva 3: IDA*-algoritmin alkutila sekä sen laajentamat solmut kierroksittain, kun hakuraja saa arvot 4, 6 ja 8. Maalisolmu on merkitty vihreällä.

kä vähentää läpikäytävien solmujen määrää huomattavasti. Tutkielmassa käytetty simulaattori ottaa tällaisen triviaalin tapauksen huomioon, ja jättää generoimatta edestakaisen liikkeen aiheuttavat solmut. Suurempien silmukoiden poistamista ei ole kuitenkaan toteutettu IDA*-algoritmin kohdalla. Dillenburg & Nelson (1993) ovat tutkineet metodin tehokkuutta käytännössä, mutta sen tarkempi läpikäyminen sivuutetaan tässä tutkielmassa.

4.2.2 Muistin käyttö

Koska IDA*-algoritmi perustuu syvyyshakuun, pitää se muistissaan korkeintaan yhden polun verran solmuja. Mikäli tutkittava polku päätyy umpikujaan, ja IDA*-algoritmi alkaa käymään läpi seuraavaa mahdollista polkua, unohdetaan entisen polun solmut. Koska hakurajaa kasvatetaan asteittain, ja haku lopetetaan kun maalisolmu löytyy, IDA*-algoritmin käyttämä muisti on korkeintaan lyhimmän polun solmujen verran. IDA*-algoritmin käyttämä muisti on siis pienin mahdollinen, mikä vaaditaan lyhimmän reitin löytämiseen, mikä on muistin käytön kannalta toisessa ääripäässä A*-algoritmiin verrattuna.

4.2.3 Algoritmin toteuttaminen käytännössä

Vaikka IDA*-algoritmi ei nautikaan niin suurta suosiota kuin A*-algoritmi, löytää siitä kuitenkin yhtä helposti tietoa kuin A*-algoritmistakin. IDA*-algoritmi on paljon yksinkertaisempi toteuttaa kuin A*-algoritmi, sillä siinä ei tarvitse huolehtia esimerkiksi monimutkaisista tietorakenteista ollenkaan. Solmujen läpikäyminen ja lyhimmän polun palauttaminen onnistuu rekursion avulla. IDA*-algoritmin toiminnan ymmärtämisen jälkeen sen toteuttaminen Javalla oli triviaalia.

4.3 IEA*

IEA-algoritmi* (*Iterative-Expansion A**) perustuu IDA*-algoritmiin. IEA*-algoritmi pyrkii parantamaan IDA*-algoritmin tehokkuutta käyttämällä enemmän muistia hyväkseen, mutta pitämällä muistin käytön kuitenkin reilusti A*-algoritmin muistin käyttöä pienempänä. IEA*-algoritmia voidaankin pitää A*-algoritmin ja IDA*-algoritmin

välimuotona. IEA*-algoritmin toimintaa kuvaa tarkemmin luku 4.3.1 sekä sivuilta 23 ja 24 löytyvät pseudokoodit (algoritmit 4 ja 5).

4.3.1 Toiminta

IEA*-algoritmi (Potts & Krebsbach, 2012) käy läpi solmuja rajoitetun syvyyshaun tapaan, samoin kuin IDA*-algoritmi. Alustavana hakurajana toimii lähtösolmun antama heuristinen $h(x)$-arvo. Tämä hakuraja kasvaa jokaisen kierroksen jälkeen, mikäli maalisolmua ei ole vielä löydetty. Rajan kasvattaminen tapahtuu kaikkien kierroksella hakurajan ylittäneiden solmujen pienimmän $f(x)$-arvon mukaan. Hakurajan tulee kasvaa jokaisen kierroksen jälkeen, jotta algoritmi ei jää pyörimään ikuiseen silmukkaan. Mikäli hakuraja ei kasva kierroksen jälkeen, voidaan reitinhaku lopettaa, sillä polkua lähtösolmusta maalisolmuun ei tällöin ole olemassa.

IEA*-algoritmi käyttää avointa ja suljettua listaa A*-algoritmin tavoin, mutta lisää avoimeen listaan solmuja paljon valikoivammin kuin A*-algoritmi. Avoimeen listaan ei talleteta kaikkia uusia solmuja, vaan ainoastaan sellaiset solmut, joiden $f(x)$-arvo on pienempi tai yhtä suuri kuin sen hetkisen kierroksen hakuraja. Jokaisella kierroksella IEA*-algoritmi pitää yllä myös väliaikaista avointa listaa, jonne uudet solmut talletetaan. Kierros päättyy kun avoin lista on käyty läpi, jolloin tyhjä avoin lista korvataan väliaikaisella listalla. Jokainen avoimeen listaan lisätty solmu lisätään samalla suljettuun listaan. Suljetusta listasta ei koskaan poisteta solmuja, mutta avoin lista uudistuu jokaisella kierroksella. Ensimmäisellä kierroksella IEA*-algoritmin avoimessa listassa on vain lähtösolmu, jolloin se on täysin identtinen IDA*-algoritmin ensimmäisen kierroksen kanssa.

Myöhemmillä kierroksilla IEA*-algoritmi suorittaa rajoitetun syvyyshaun (algoritmi 5) jokaisesta avoimessa listassa olevasta solmusta. Haku on identtinen IDA*-algoritmin haun kanssa sillä erotuksella, että IEA*-algoritmi tarkistaa jokaisen solmun kohdalla, onko se lisätty suljettuun listaan. Jos tutkittava solmu on suljetussa listassa, ei sitä tarvitse käydä enää läpi, ja se voidaan turvallisesti jättää välistä. Tämä on tehokas metodi silmukoiden poistoon haun aikana, ja sen mahdollinen hyöty voi olla jopa eksponentiaalinen, mikäli silmukka voidaan jättää käymättä varhaisessa vaiheessa, jolloin koko alipuu voidaan jättää tutkimatta.

Algoritmi 4 IEA*-algoritmi, jonka parametreina ovat alku- ja maalisolmu

```
1:  procedure IEA*(alku, maali)
2:      hakuraja ← h(alku)
3:      avoin ← {alku}
4:      suljettu ← {alku}
5:      while hakuraja kasvaa do
6:          uusi ← ∅
7:          minimi ← ∞
8:          while avoin ≠ ∅ do
9:              solmu ← POLL(avoin)                    ▷ Poista ja palauta paras solmu
10:             t ← ETSI(solmu, hakuraja, maali)       ▷ Ks. algoritmi 5
11:             if t = FOUND then
12:                 return reitti
13:             else if t < minimi and t > hakuraja then
14:                 minimi ← t
15:             end if
16:             for each solmun naapuri do
17:                 if naapuri ∈ suljettu then
18:                     continue
19:                 else if f(naapuri) ≤ hakuraja then
20:                     uusi ← uusi ∪ {naapuri}
21:                     suljettu ← suljettu ∪ {naapuri}
22:                 end if
23:             end for
24:             if ∃solmun naapurit ∉ uusi then
25:                 /*Kaikki solmun naapurit eivät ole uudessa listassa*/
26:                 uusi ← uusi ∪ {solmu}
27:             end if
28:         end while
29:         avoin ← uusi
30:         hakuraja ← minimi
31:     end while
32:     return NOT FOUND
33: end procedure
```

Algoritmi 5 IEA*-algoritmin etsi-metodi

```
1:  procedure ETSI(solmu, hakuraja, maali)
2:      f_arvo ← g(solmu) + h(solmu)
3:      if f_arvo > hakuraja then
4:          return f_arvo
5:      else if solmu = maali then
6:          return FOUND
7:      end if
8:      minimi ← ∞
9:      for each solmun naapuri do
10:         if naapuri ∈ suljettu then
11:             continue
12:         end if
13:         g(naapuri) ← g(solmu) + d(solmu, naapuri)
14:         t ← ETSI(naapuri, hakuraja, maali)
15:         if t = FOUND then
16:             return FOUND
17:         else if t < minimi then
18:             minimi ← t
19:         end if
20:     end for
21:     return minimi
22: end procedure
```

4.3.2 Muistin käyttö

Pääidea IEA*-algoritmissa on nopeuttaa reitinhakua verrattuna IDA*-algoritmiin hyväksikäyttämällä enemmän muistia kuin IDA*-algoritmi, mutta pitämällä muistivaativuuden kuitenkin niin pienenä, ettei se törmää A*-algoritmin tavoin ongelmiin isoissa hakuavaruuksissa. IEA*-algoritmin käyttämän muistin määrän voi arvioida kaavalla (4) (Potts & Krebsbach, 2012).

$$b^{(d-h(alku))/2} \tag{4}$$

Kaavassa (4) b tarkoittaa verkon haarautumiskerrointa, d lyhimmän reitin pituutta ja $h(alku)$ lähtösolmun heuristista arviota. Koska 15-puzzlen tapauksessa haarautumiskerroin voi korkeimmillaan olla 4, sen käyttäminen kaavassa (4) laskee IEA*-algoritmin käyttämän muistin maksimiarvon. Tarkemman arvion kuitenkin saa mikäli käyttää asymptoottista haarautumiskerrointa 2,13 (Korf, 2000).

4.3.3 Algoritmin toteuttaminen käytännössä

Koska IEA*-algoritmi toimii lähes täysin samoin kuin IDA*-algoritmi, tarvitsi IDA*-algoritmia muuttaa vain hieman, jotta siitä sai toimivan IEA*-algoritmin. Käytännössä tämä tarkoitti tietorakenteiden lisäämistä nopeuden parantamiseksi. Potts & Krebsbach (2012) olivat esittäneet selkeän pseudokoodin hyvin tarkasti selitettynä, mikä helpotti algoritmin ymmärtämistä ja toteuttamista. IEA*-algoritmin vaatimukset tietorakenteelta olivat yksinkertaisemmat kuin A*-algoritmilla, joten Javan oma PriorityQueue-luokka pystyi tarjoamaan tarvittavat toiminnot avoimen listan toteuttamiseen. Algoritmin toteuttaminen ei ollut triviaalia, mutta todennäköisesti onnistuu ongelmitta kokeneelta ohjelmoijalta.

4.4 SMA*

SMA-algoritmi* (*Simplified Memory-Bounded A**) (Russell, 1992) eroaa edellä mainituista algoritmeista siten, että se on tässä tutkielmassa esitellyistä algoritmeista ainoa, jonka muistin käytön voi rajoittaa tarkasti. SMA*-algoritmi käyttää kaiken sille annetun muistin hyödyksi, ja poistaa solmuja muistista vasta silloin kun on pakko. SMA*-

algoritmi löytää mahdollisen reitin maaliin, mikäli muistikapasiteetti on vähintään reitin pituuden mittainen. SMA*-algoritmi pystyy siis toimimaan yhtä vähällä muistilla kuin IDA*-algoritmi, mutta pystyy myös hyödyntämään isompaa muistikapasiteettia. SMA*-algoritmi eroaa myös muista algoritmeista siten, että se pitää jokaista solmua kohden kaksi erilaista $f(x)$-arvoa tallessa, sekä osoittimen, joka määrittää seuraavaksi laajennettavan jälkeläisen. SMA*-algoritmin toimintaa kuvaa tarkemmin luku 4.4.1 sekä sivulta 27 löytyvä pseudokoodi (algoritmi 6).

4.4.1 Toiminta

SMA*-algoritmi aloittaa etsimisen laajentamalla ensin sille annetun lähtösolmun. Toisin kuin A*-algoritmi, SMA*-algoritmi laajentaa vain yhden uuden solmun kerrallaan. Seuraavilla kierroksilla SMA*-algoritmi valitsee A*-algoritmin tavoin laajennettavaksi pienimmän $f(x)$-arvon omaavan solmun. Mikäli tällaisia solmuja on useita, valitaan niiden kesken korkeimman $g(x)$-arvon omaava solmu, eli mahdollisimman pitkälle edennyt solmu. Tämän jälkeen valitusta solmusta laajennetaan yksi uusi solmu. Koska solmuja laajennetaan yksi kerrallaan, jokaisessa solmussa täytyy olla osoitin, joka määrittää seuraavaksi laajennettavan jälkeläisen. Mikäli jokin solmu on täysin laajennettu, eli kaikki sen jälkeläiset ovat muistissa, voidaan se poistaa avoimesta listasta tulevien hakujen nopeuttamiseksi, sillä sitä ei voida enää laajentaa, eikä sitä voida unohtaa, koska ainoastaan hakupuun lehtisolmut voidaan unohtaa. Muiden kuin lehtisolmujen unohtaminen hajottaisi hakupuun, eikä silloin olisi enää kokonaista polkua lähtösolmusta unohdetun solmun jälkeläisiin. Solmu täytyy kuitenkin pitää muistissa, eikä sen poistaminen avoimesta listasta vapauta uudelle solmulle tilaa.

Yksi tärkein piirre (Russell, 1992) SMA*-algoritmissa verrattuna muihin muistirajoitettuihin reitinhakualgoritmeihin on sen ominaisuus muuttaa solmujen $f(x)$-arvoja. SMA*-algoritmi käyttääkin $pathmax$-metodia (kaava (5)) määrittääkseen solmun todellisen $f(x)$-arvon.

$$f(x) = \max(f(vanhempi), g(x) + h(x)) \tag{5}$$

Käytännössä tämä tarkoittaa sitä, että aina uutta solmua laajennettaessa tarkastetaan, onko uuden solmun $f(x)$-arvo (kaava (2) s. 5) pienempi kuin sen isäsolmun $f(x)$-arvo. Mikäli näin on, määritetään uuden solmun $f(x)$-arvoksi sen isäsolmun $f(x)$-arvo. Mikäli heuristinen funktio on kelvollinen, $f(x)$-arvon ei tulisi koskaan olla suurempi

Algoritmi 6 SMA*-algoritmi, jonka parametreina ovat alku- ja maalisolmu

```
1: procedure SMA*(alku, maali)
2:     avoin ← {alku}
3:     while avoin ≠ ∅ do
4:         solmu ← POLL(avoin)                          ▷ Poista ja palauta paras solmu
5:         if solmu = maali then
6:             return reitti
7:         end if
8:         if f(solmu) = ∞ then
9:             /*Paras solmu on ääretön, joten ei ole olemassa reittiä*/
10:            return NOT FOUND
11:        end if
12:        naapuri ← SEURAAVA NAAPURI(solmu, avoin)      ▷ Ks. algoritmi 8
13:        if naapuri ≠ maali and naapuri on maksimisyvyydessä then
14:            /*Haku saavuttanut suurimman muistin salliman syvyyden*/
15:            f(naapuri) ← ∞          ▷ Polku ei voi kulkea tämän solmun kautta
16:        else
17:            f(naapuri) ← MAX(f(solmu), g(naapuri) + h(naapuri))
18:        end if
19:        BACKUP(solmu, avoin)                          ▷ Ks. algoritmi 7
20:        if ∀solmun naapurit ∈ avoin then          ▷ Kaikki naapurit muistissa
21:            Poista solmu avoimesta listasta, mutta ei muistista
22:            f′(solmu) ← ∞                                  ▷ Alusta f′
23:        end if
24:        if size(avoin) > LIMIT then                        ▷ Muisti täynnä
25:            poistettu ← POLL WORST(avoin)
26:            v ← vanhempi(poistettu)
27:            f′(v) ← MIN(f′(v), f(poistettu))
28:            /*Vanhempi v palautetaan avoimeen listaan, jos se oli siitä poistettu*/
29:            if v ∉ avoin then
30:                avoin ← avoin ∪ {v}
31:            end if
32:        end if
33:        avoin ← avoin ∪ {naapuri}
34:    end while
35:    return NOT FOUND
36: end procedure
```

kuin lyhimmän reitin pituus solmun x kautta. Solmun $f(x)$-arvo on siis alaraja solmun x kautta maalisolmuun kulkevan reitin pituudelle. Tästä syystä solmun x jälkeläisten $f(x)$-arvojen tulee olla vähintään yhtä suuria kuin niiden isäsolmun $f(x)$-arvon. Toinen tilanne jossa SMA*-algoritmi muuttaa solmujen $f(x)$-arvoja on se, kun solmu on maksimisyvyydessä eikä ole maalisolmu. Solmu on maksimisyvyydessä silloin, kun jokainen muistissa oleva solmu on laajentanut vain yhden jälkeläisen, eli solmut muodostavat linkitetyn listan. Koska solmu ei ole maalisolmu, ja kaikki muisti on jo käytetty, ei ole mahdollista löytää kyseisen solmun kautta reittiä annetulla muistin määrällä. Tästä syystä SMA*-algoritmi asettaa tällaisten solmujen $f(x)$-arvon äärettömän suureksi.

Kaikki muistissa olevat solmut talletetaan binääripuista koostuvaan binääripuuhun (Russell, 1992). Solmut talletetaan siten, että yhteen binääripuuhun kerätään kaikki saman $f(x)$-arvon omaavat solmut, ja nämä solmut järjestetään niiden $g(x)$-arvon perusteella. Alemman tason binääripuut siis järjestävät solmut $g(x)$-arvon mukaan, ja ylemmän tason binääripuu järjestää binääripuut niissä olevien solmujen $f(x)$-arvojen mukaan. Tällöin on mahdollista löytää nopeasti muistissa olevista solmuista paras ja huonoin solmu. SMA*-algoritmin yhteydessä puhuttaessa binääripuusta tarkoitetaan tätä binääripuista koostuvaa binääripuuta.

Muistin loppuessa SMA*-algoritmi joutuu unohtamaan solmuja tehdäkseen tilaa uusille solmuille. Poistettava solmu valitaan suurimman $f(x)$-arvon perusteella. Mikäli useilla solmuilla on sama $f(x)$-arvo, valitaan niiden joukosta sellainen solmu, jolla on pienin $g(x)$-arvo (Kaindl & Khorsand, 1994). Poistaessaan solmuja muistista SMA*-algoritmi tallettaa unohdetun solmun $f(x)$-arvon sen isäsolmuun. Isäsolmussa pidetään tallessa vain yksi $f'(x)$-arvo unohdetuille jälkeläisille, ja siihen talletetaan aina pienin unohdettu $f(x)$-arvo. Tämän unohdettujen solmujen $f'(x)$-arvon tarkoitus on ohjata SMA*-algoritmia siten, että vaikka jokin polku jouduttaisiin unohtamaan kokonaan, SMA*-algoritmi silti pystyy vertaamaan unohdettua polkua muihin solmuihin. Käytännössä tämä tarkoittaa sitä, että SMA*-algoritmi tietää minimikustannuksen sille polulle, joka kulkee tietyn solmun kautta, vaikka sen jälkeläiset olisi jouduttu jo unohtamaan. Solmun unohtamisen yhteydessä täytyy myös tarkistaa, onko unohdettavan solmun isäsolmu täysin laajennettu, eli onko se poistettu binääripuusta. Mikäli näin on, tulee isäsolmu palauttaa takaisin binääripuuhun, koska se on jälleen laajennettavissa.

Eräs SMA*-algoritmin tärkeimmistä ominaisuuksista on solmujen $f(x)$-arvojen *päi-*

vittäminen (*backup*), jonka pseudokoodi esitetään algoritmissa 7 sivulla 29. Päivittäminen tarkoittaa solmun $f(x)$-arvon kasvattamista, mikäli solmun kaikki jälkeläiset on jo käyty läpi. Päivittäminen vaatii sen, että solmu on valmis (selitetään tarkemmin luvussa 4.4.2). On erittäin tärkeää ettei solmun $f(x)$-arvoa päivitetä silloin, kun solmu ei ole valmis, koska silloin solmun $f(x)$-arvo saattaisi kasvaa liian suureksi, mikä aiheuttaisi mahdollisesti epäoptimaalisen reitin löytymisen. Kun kaikki solmun jälkeläiset on käyty läpi, solmun uusi $f(x)$-arvo määräytyy jälkeläisten pienimmän $f(x)$-arvon mukaan. Vaikka yksi tai useampi solmun jälkeläisistä on jouduttu unohtamaan, on niiden pienin $f(x)$-arvo kuitenkin tallessa isäsolmun $f'(x)$-arvossa, jota voidaan käyttää päivityksessä. Päivityksessä tuleekin ottaa huomioon solmun muistissa olevat jälkeläiset sekä solmun $f'(x)$-arvo, ja valita näistä arvoista kaikkein pienin solmun uudeksi $f(x)$-arvoksi. Päivityksen jälkeen solmun $f(x)$-arvo ei tietenkään voi koskaan pienentyä $pathmax$-metodin (kaava (5) s. 26) ansiosta. Mikäli solmun $f(x)$-arvo muuttuu päivityksessä, täytyy myös binääripuuta päivittää, jotta se ei olisi epäjärjestyksessä solmun uuden $f(x)$-arvon takia. Koska alemman tason binääripuihin talletetaan kaikki saman $f(x)$-arvon omaavat solmut, joudutaan solmu poistamaan yhdestä binääripuusta ja lisäämään se toiseen.

Algoritmi 7 SMA*-algoritmin backup-metodi, joka kasvattaa valmiin solmun $f(x)$-arvoa mikäli mahdollista

```
1:  procedure BACKUP(solmu, avoin)
2:      if valmis(solmu) then
3:          uusi_f ← MIN(Kaikkien solmun naapureiden f-arvot, f'(solmu))
4:          if f(solmu) < uusi_f then
5:              f(solmu) ← uusi_f
6:              Järjestä avoin vastaamaan uutta arvoa f
7:              BACKUP(vanhempi(solmu), avoin)
8:          end if
9:      end if
10: end procedure
```

Reitinhaun alussa useat polut voivat tuntua lupaavilta, jolloin SMA*-algoritmi joutuu vaihtelemaan näiden polkujen välillä useasti. Rajoitetun muistin takia toisen polun solmut joudutaan unohtamaan, jotta toista polkua pystytään laajentamaan. Jossain vaiheessa kuitenkin jonkin toisen polun $f(x)$-arvo näyttää jälleen lupaavammalta, jolloin SMA*-algoritmi pyrkii laajentamaan sitä. Ongelmissa joissa tällaista vaihtelua tapahtuu paljon, SMA*-algoritmi joutuu laajentamaan samoja solmuja ja jopa laajoja hakupuita useita kertoja. Tällaisissa tapauksissa SMA*-algoritmi on vaikeuksissa, kun taas A*-algoritmilla ei tällaista ongelmaa esiinny (Russell & Norvig, 2009 s. 102). SMA*-

algoritmin käyttämän solmujen $f(x)$-arvojen kasvattamisen hyödyllisyys ilmenee syvempinä hakupuina. Käytännössä tämä tarkoittaa sitä, ettei edellä mainittua polkujen vaihtelua tapahdu niin usein.

Jokainen solmu sisältää siis osoittimen, joka määrittää seuraavaksi laajennettavan solmun. Tällaista osoitinta tarvitaan, sillä koskaan ei voi olla varma missä vaiheessa solmuja joudutaan unohtamaan, jolloin ei voida turvautua sellaiseen menetelmään, että laajennettaisiin aina ensimmäinen laajentamaton solmu. Jokaisen laajennuksen yhteydessä osoitin määritetään osoittamaan seuraavaa jälkeläistä kunnes kaikki jälkeläiset on laajennettu. Kun osoitin on käynyt läpi kaikki solmun jälkeläiset, se asetetaan osoittamaan jälleen ensimmäistä jälkeläistä. Mikäli solmua laajennettaessa seuraavaksi laajennettava solmu onkin jo laajennettu, jatketaan jälkeläisten läpikäymistä kunnes löytyy seuraava laajentamaton jälkeläinen. Solmussa tulee olla vähintään yksi laajentamaton jälkeläinen tässä tilanteessa, koska muuten solmu olisi poistettu avoimesta listasta, eikä sitä voisi valita laajennettavaksi. Seuraavan laajennettavan jälkeläisen valinta on esitetty pseudokoodina algoritmissa 8.

Algoritmi 8 SMA*-algoritmin seuraavan naapurisolmun palauttava metodi

```
1:  procedure SEURAAVA NAAPURI(solmu, avoin)
2:      if osoitin(solmu) = 0 then
3:          f'(solmu) ← ∞                                   ▷ Alusta f'
4:      end if
5:      while osoitin käy läpi kaikki naapurit do
6:          /*osoitin jatkaa siitä mihin viime kerralla jäi*/
7:          if osoitin osoittaa viimeistä naapuria then
8:              kierros ← TRUE          ▷ Kierros valmis, joten päivitys turvallista
9:          end if
10:         if osoitin osoittaa laajentamatonta naapuria then
11:             if kierros then
12:                 BACKUP(solmu, avoin)                    ▷ Ks. algoritmi 7
13:                 f'(solmu) ← ∞                           ▷ Alusta f'
14:             end if
15:             return osoittimen osoittama naapuri
16:         end if
17:         osoitin ← osoitin + 1
18:         if osoitin > size(naapurit(solmu)) then
19:             osoitin ← 0
20:         end if
21:     end while
22: end procedure
```

Täydellisen heuristiikan tapauksessa SMA*-algoritmi toimii lähes samoin kuin A*-

algoritmikin. SMA*-algoritmin tapauksessa ei kuitenkaan ole väliä, käykö algoritmi tasa-arvoiset solmut FIFO- vai LIFO-periaatteella, sillä SMA*-algoritmi valitsee aina sen solmun, jolla on suurin $g(x)$-arvo. Tästä syystä SMA*-algoritmi laajentaa vain yhden reitin, eikä useaa reittiä, mikäli ne ovat samanpituisia. On myös mahdollista, että SMA*-algoritmi laajentaa pienemmän määrän solmuja kuin A*-algoritmi, sillä se laajentaa aina vain yhden solmun kerrallaan. SMA*-algoritmin laajentamien solmujen määrä riippuu naapurisolmujen järjestyksestä. Muistissa olevien solmujen määrä on SMA*-algoritmilla kuitenkin aina pienempi tai yhtä suuri kuin A*-algoritmilla, mikäli siis heuristiikka on täydellinen, ja SMA*-algoritmilla on tarpeeksi muistia.

Sellaisen verkon tapauksessa, jossa verkon kaikkien kaarien paino olisi 0, SMA*-algoritmin toiminta riippuisi täysin siitä, kuinka tasa-arvoisten solmujen käsittely olisi toteutettu. Tällaisen painottoman verkon tapauksessa $f(x)$-arvo olisi aina 0, sillä $g(x)$-arvo ei kasvaisi koskaan, eikä $h(x)$-arvo voisi koskaan olla suurempi kuin 0. FIFO-periaatteen perusteella toimiva SMA*-algoritmi kävisi solmuja läpi leveyshakumaisesti, mutta muistirajoituksen vuoksi ei pystyisi etenemään kovin syvälle hakupuussa. Tämä johtuu siitä, että muistin loppuessa algoritmi alkaa käymään läpi samoja solmuja uudestaan, eikä haku etene enää uusiin solmuihin. Koska jokaisen solmun $f(x)$-arvo on aina 0, ei SMA*-algoritmi pysty päivittämisen avulla kasvattamaan $f(x)$-arvoja, jotta haku etenisi. LIFO-periaatteella toimiessaan SMA*-algoritmi ei jumiudu paikoilleen, koska silloin SMA*-algoritmi toimii rajoitetun syvyyshaun mukaan. Tällöin jokainen hakurajan sisäpuolella oleva solmu käydään läpi. Myös silloin jos $g(x)$-arvo kasvaisi yhdellä jokaisen solmusta toiseen siirtymisen jälkeen, SMA*-algoritmi toimisi rajoitetun syvyyshaun mukaisesti.

4.4.2 Algoritmin toteuttaminen käytännössä

SMA*-algoritmin toteuttamisessa oli käytännössä kaksi pääongelmaa: tiedon vähyys ja sen epäselvyys. SMA*-algoritmin harvinaisuudesta kertoo se, ettei siitä löytynyt juurikaan tietoa kirjallisuudesta tai Internetistä. Mikäli jossain lähteessä kerrottiin jotain SMA*-algoritmista, se oli yleensä hyvin pelkistettyä tai epäselvää. Alkuperäisessä lähteessä (Russell, 1992) oli jopa selviä virheitä pseudokoodissa (Kaindl & Khorsand, 1994). Tiedon vähyydestä huolimatta algoritmin toiminta oli helppo ymmärtää yleisellä tasolla, mutta käytännön toteuttamiseen tarvittavat yksityiskohdat olivat vaikeita hahmottaa. Esimerkiksi solmun $f(x)$-arvon päivittämisen yhteydessä ei mainittu mi-

tään siitä, että myös binääripuuta tulisi muokata päivityksen jälkeen. Tämä voi tuntua itsestään selvältä asialta, mutta algoritmin monimutkaisuuden viedessä huomion saattaa tällainen pieni mutta tärkeä asia jäädä huomaamatta. Kaindl & Khorsand (1994) olivat kuitenkin omassa pseudokoodissaan maininneet tämän asian selkeästi.

Russell & Norvig (1996) ovat hahmotelleet pseudokoodin SMA*-algoritmista kirjansa ensimmäisessä painoksessa. Tämä oli erittäin hyvä apu algoritmin toiminnan hahmottamisessa, vaikka se ei ollutkaan kovin yksityiskohtainen. Myöskin siinä esiintyvät virheet oli helppo huomata. Kirjassa oli myös esitetty lyhyt esimerkkitapaus SMA*-algoritmin toiminnasta, joka selvensi algoritmin reitinhakua hyvin. Esimerkissä oli kuitenkin käytetty hakupuuta, jonka haarautumiskerroin oli tasan kaksi. Tämä tarkoittaa sitä, että jokaisella solmulla, paitsi lehtisolmuilla, oli kaksi jälkeläistä. Esimerkki jätti siis epäselväksi miten algoritmi käyttäytyisi, mikäli solmulla olisi kolme tai useampia jälkeläisiä. 15-puzzlen tapauksessa jälkeläisiä voi olla jopa neljä. Russellin & Norvigin (2009) kirjan kolmannessa painoksessa ei esiinny enää SMA*-algoritmin pseudokoodia, mutta siinä kerrotaan algoritmin toiminnasta edelleen virheellisesti. Kirjoittajat väittävät, että SMA*-algoritmi laajentaa aina parhaan lehtisolmun, ja unohtaa huonoimman lehtisolmun. Jotta SMA*-algoritmi toimisi oikein (Kaindl & Khorsand, 1994), laajennuksessa tulisi valita paras solmu kaikista avoimessa listassa olevista solmuista, eikä pelkästään lehtisolmuista. Muutoin jokaisella solmulla voisi olla vain yksi jälkeläinen muistissa, jolloin SMA*-algoritmin toiminta muistuttaisi rajoitettua syvyyshakua, jossa hakurajana olisi muistin määrä. Huonoin solmu tulee kuitenkin aina valita lehtisolmujen joukosta, koska sisäsolmun poistaminen rikkoisi hakupuun.

Binääripuista koostuva binääripuu on siis SMA*-algoritmin käyttämä tietorakenne avoimen listan toteutukseen. Java ei valitettavasti tarjoa sopivaa toteutusta binääripuulle. Javan TreeMap-luokka perustuu punamustaan puuhun, mutta se ei sellaisenaan kelvannut SMA*-algoritmin avoimen listan toteutukseen samoista syistä kun se ei ollut sopiva A*-algoritmin avoimen listan toteutukseen. Binääripuun toteuttaminen Javalla on melko triviaalia, mutta yksinkertainen tasapainottamaton binääripuu ei ole tarpeeksi tehokas. Koska SMA*-algoritmi kasvattaa solmujen $f(x)$-arvoja jatkuvasti, tasapainottamaton binääripuu muuttuu lopulta pelkäksi linkitetyksi listaksi, jonka aikavaativuus ei ole enää logaritminen. Tästä syystä tasapainotettu binääripuu on ehdoton SMA*-algoritmin toteutukselle. Tutustuin ensin *splay-puuhun* (Sleator & Tarjan, 1985), joka on itsestään tasapainottuva binääripuu. Erikoista splay-puussa on se, että se nostaa viimeksi lisätyn, poistetun tai haetun alkion puun juurialkioksi, mikä mahdollisesti

nopeuttaa tulevia hakuja. Splay-puun toiminta nopeutuu silloin, kun samoja puun alkioita käytetään usein. Tällöin ne alkiot kerääntyvät lähelle splay-puun juurta, ja ovat nopeasti löydettävissä. Oletin splay-puun olevan hyvä ratkaisu SMA*-algoritmin avoimen listan toteutukseen, sillä jokin solmu voi olla paras vaihtoehto monesti peräkkäin, riippuen sen jälkeläisten määrästä. Tällöin paras solmu on juurialkio, jolloin sen hakeminen on erittäin nopeaa. Valitettavasti splay-puu ei ollut juuri ollenkaan tasapainossa, mikä hidasti niiden alkioiden hakemista, joita ei oltu käytetty vähään aikaan. Puun epätasapainoisuus sekä ylimääräinen aika joka kuluu viimeksi käytetyn alkion asettamiseen juurialkioksi tekevät splay-puusta kelvottoman SMA*-algoritmin avoimen listan toteutukseen. Lopulta päädyin toteuttamaan avoimen listan *AVL-puun* avulla. AVL-puu (Knuth, 1973 s. 451) on itsestään tasapainottuva binääripuu, jonka tärkeimmät operaatiot toimivat huonoimmassakin tapauksessa logaritmisessa ajassa.

SMA*-algoritmin toiminnan yksi keskeisin osa on solmujen $f(x)$-arvojen päivittäminen silloin, kun solmu on *valmis* (*complete*). Se, milloin solmu käytännössä on valmis, oli epäselvästi ilmaistu Russellin (1992) artikkelissa. Hänen mukaansa solmu on valmis silloin, kun sen jälkeläisistä mikään ei ole enää tutkimaton (unexamined), ja solmu on tutkimaton silloin, kun sitä ei ole vielä laajennettu sen jälkeen, kun sen isäsolmu on laajennettu. Mikäli isäsolmu joudutaan unohtamaan ja laajentamaan uudestaan, kaikki sen jälkeläiset ovat silloin tutkimattomia. Mikäli kuitenkin valmiin solmun jälkeläinen joudutaan unohtamaan, on se jo kertaalleen laajennettu, jolloin isäsolmun tulisi edelleen olla valmis. Tämä oli kaikkein vaikein asia ymmärtää SMA*-algoritmissa, sillä se on monitulkintaisesti selitetty Russellin (1992) artikkelissa, ja kaiken lisäksi se ei pidä edes paikkaansa. Vaikka edellä mainittu selitys onkin suuntaa antava, jättää se tärkeitä yksityiskohtia kertomatta.

Kun solmu laajennetaan, se ei tietenkään voi olla valmis, sillä kaikki sen jälkeläiset ovat vielä tutkimattomia. Solmu on valmis vasta sitten, kun kaikki sen jälkeläiset on laajennettu vähintään kerran. Solmu pysyy valmiina, vaikka kaikki sen jälkeläiset jouduttaisiin jossain vaiheessa unohtamaan. Tämä selittyy sillä, että unohdettujen solmujen pienin $f(x)$-arvo on talletettu isäsolmun $f'(x)$-arvoon, jolloin isäsolmun $f(x)$-arvoa on turvallista päivittää, sillä se ei voi kasvaa liian suureksi. Vasta siinä tapauksessa, kun valmis solmu valitaan uudelleen laajennettavaksi, se ei enää välttämättä ole valmis. Solmun tila riippuu siitä, onko sillä vielä unohdettuja jälkeläisiä laajennuksen jälkeen. Mikäli solmun kaikki jälkeläiset ovat muistissa, voidaan olla varmoja siitä, että kyseinen solmu on valmis. Jos näin ei ole, joudutaan turvautumaan solmun osoittimeen.

Solmun osoitin on kierroksen lopussa silloin kun kaikki sen solmun jälkeläiset on laajennettu kerran, ja seuraavaksi laajennettava solmu olisi jälleen se solmu, joka laajennettiin ensimmäiseksi. Ainoastaan silloin kun solmun osoitin on kierroksen lopussa, tai solmun kaikki jälkeläiset ovat edelleen muistissa, voidaan solmua kutsua valmiiksi.

Missään löytämässäni artikkelissa ei otettu kantaa siihen, milloin solmun $f'(x)$-arvo tulisi alustaa, eli käytännössä muuttaa äärettömän suureksi. Mikäli sitä ei alusteta koskaan, jää SMA*-algoritmi jossain tilanteessa jumiin, sillä solmun $f'(x)$-arvoon talletetaan aina pienin unohdetun jälkeläisen $f(x)$-arvo, ja SMA*-algoritmin perusidea on kasvattaa solmujen $f(x)$-arvoja etsinnän edetessä. Mikäli solmun $f'(x)$-arvo asetetaan todella pieneksi reitinhaun alkuvaiheessa, ei kyseisen solmun $f(x)$-arvo pääse koskaan päivittymään suuremmaksi, jos $f'(x)$-arvoa ei alusteta välillä. Tällöin menetetään arvokasta tietoa, kun solmuja joudutaan unohtamaan. Triviaali tapaus on se kun solmu unohdetaan, jolloin kaikki sen tiedot joudutaan unohtamaan. Tällöin myös unohdetun solmun $f'(x)$-arvo alustetaan. Toinen triviaali tapaus on se, kun kaikki solmun jälkeläiset ovat muistissa. Tällöin voidaan $f'(x)$-arvo turvallisesti alustaa, koska päivityksen yhteydessä tarvittavat $f(x)$-arvot voidaan suoraan hakea jälkeläisiltä. Kolmas tapaus milloin solmun $f'(x)$-arvo alustetaan, on se, kun solmun kierros päättyy, eli kun kaikki jälkeläiset on kertaalleen laajennettu. Alustaminen on turvallista siksi, koska solmun $f(x)$-arvon päivittäminen tapahtuu myös kierroksen lopussa. Kun solmun $f(x)$-arvo on päivitetty, voidaan sen $f'(x)$-arvo unohtaa, sillä kaikki solmut joudutaan laajentamaan uudestaan ennen kuin päivittäminen on jälleen mahdollista. Ei ole väliä, onko solmulla muistissa olevia jälkeläisiä $f'(x)$-arvon unohtamisen aikaan, sillä mikäli jälkeläisiä joudutaan unohtamaan, niiden $f(x)$-arvo tallettuu isäsolmun $f'(x)$-arvoon seuraavan kierroksen ajaksi.

SMA*-algoritmin toteuttamisen aikana huomasin joitain heikkouksia algoritmin suunnittelussa. Suurin ongelma ilmeni huonoimman alkion poistamisen yhteydessä. Sisäkkäisten binääripuiden tarkoituksena on tarjota mahdollisimman nopeasti paras ja huonoin solmu, sillä ne ovat SMA*-algoritmin keskeisimpiä operaatioita. Käytännössä tämä ei kuitenkaan toimi, sillä SMA*-algoritmi voi poistaa muistista vain lehtisolmuja. Mikäli avoimena listana toimivan binääripuun huonoin solmu ei olekaan lehtisolmu, joudutaan puuta tutkimaan pidempään, jotta huonoin lehtisolmu löytyy. Jos lehtisolmuja on huomattavasti vähemmän kuin sisäsolmuja, voi poistettavan alkion etsiminen olla erittäin hidasta. Yksi tapa tämän ongelman korjaamiseen on toisen binääripuun luominen pelkästään lehtisolmuille. Toinen binääripuu ei kasvata SMA*-algoritmin

muistin käyttöä merkittävästi, sillä sama määrä solmuja pidetään muistissa, mutta ne ovat vain lajiteltu kahteen eri binääripuuhun. Binääripuun jakaminen kahteen erilliseen puuhun vaatii lisämuistia vain toisen puun tarvitsemien osoittimien verran. Kahden binääripuun käyttäminen tuo mukanaan omat ongelmansa, mutta ne eivät ole niin vakavia kuin alkuperäinen huonoimman solmun poistaminen yhdestä binääripuusta. Ensimmäinen ongelma on parhaan solmun etsiminen laajennettavaksi. Koska solmut on nyt jaettu kahteen eri binääripuuhun, joudutaan haku tekemään kahdesta puusta. Vaikka binääripuiden alkioiden yhteinen lukumäärä pysyy samana, on haku noin kaksinkertaisesti hitaampaa kuin yhdestä binääripuusta hakeminen. Tästä huolimatta lehtipuiden jakaminen omaan binääripuuhun on kannattavaa, sillä vaikka logaritminen aikavaativuus kaksinkertaistuukin, on se paljon nopeampaa kuin mahdollinen lineaarinen aikavaativuus, kuten pahimmassa tapauksessa voi yhden binääripuun versiossa tapahtua. Toinen ongelma on solmujen vaihteleminen puiden välillä. Kun lehtisolmun jälkeläinen laajennetaan, lehtisolmu muuttuu sisäsolmuksi, ja se täytyy siirtää toiseen binääripuuhun. Sama operaatio joudutaan tekemään, mikäli solmun kaikki jälkeläiset unohdetaan, jolloin solmu muuttuu sisäsolmusta lehtisolmuksi, ja se joudutaan siirtämään lehtisolmuille tarkoitettuun binääripuuhun. Näistä lisäoperaatioista huolimatta avoimen listan jakaminen kahdeksi binääripuuksi nopeutti SMA*-algoritmin toimintaa huomattavasti, varsinkin silloin kun käytössä oli paljon muistia.

Toinen hieman pienempi heikkous SMA*-algoritmissa on sen käyttämä tuplabinääripuu. Solmujen jakaminen eri binääripuihin niiden $f(x)$-arvojen perusteella nopeuttaa algoritmin toimintaa, sillä SMA*-algoritmi hakee useimmiten suurimman tai pienimmän $f(x)$-arvon omaavia solmuja. SMA*-algoritmin $pathmax$-metodi (kaava (5) s. 26) sekä solmujen $f(x)$-arvojen päivittäminen kuitenkin aiheuttavat sen, että suurimmalla osalla solmuista on sama $f(x)$-arvo. Testausympäristönä toimineen 15-puzzlen tapauksessa avoin lista sisälsi vain kahta eri $f(x)$-arvoa samanaikaisesti. Tämä on ymmärrettävää, sillä SMA*-algoritmi pyrkii ensiksi laajentamaan pienimmän $f(x)$-arvon omaavia solmuja, joiden $f(x)$-arvo kasvaa päivittämisen myötä. Lopulta kaikki solmut omaavat saman $f(x)$-arvon, jolloin solmujen $g(x)$-arvo määrää seuraavaksi laajennettavan solmun. Tällainen $f(x)$-arvojen homogeenisuus johtuu 15-puzzlen abstraktista verkosta. Koska jokaisesta verkon solmusta naapurisolmuun siirtymisen kustannus on vakio, eivät solmujen $f(x)$-arvojen erot pääse kasvamaan suuriksi. Tämä pätee kaikkiin verkkoihin, joissa kaikkien solmujen etäisyydet naapurisolmuihin ovat samoja. Mikäli verkon kaarien painot vaihtelevat suuresti, on todennäköistä, että SMA*-algoritmin ylläpitämä binääripuu sisältää useampia alemman tason binääripuita

kuin 15-puzzlen tapauksessa.

Russell (1992) väittää artikkelissaan SMA*-algoritmin toimivan samoin kuin A*-algoritmi, mikäli SMA*-algoritmilla on käytössään enemmän muistia kuin A*-algoritmi käyttäisi saman ongelman ratkaisemiseen. Tämä voi pitää paikkansa joissakin tapauksissa, mutta se ei päde yleisesti. SMA*-algoritmin toiminta muistuttaa syvyyshakua, mikäli solmujen $f(x)$-arvot ovat usein samoja. Jos juuri laajennetun solmun $f(x)$-arvo on sama kuin sen vanhemman $f(x)$-arvo, valitsee SMA*-algoritmi tämän uuden solmun laajennettavaksi seuraavaksi, koska sen $g(x)$-arvo on suurempi. Tällöin muistissa on useita solmuja, jotka ovat vain osittain laajennettuja. Tämä on suurin ero A*-algoritmiin verrattuna, sillä A*-algoritmi laajentaa aina solmun kaikki jälkeläiset kerralla. Koska A*-algoritmi ei ota huomioon solmun $g(x)$-arvoa ollenkaan, saattaa se käydä läpi saman $f(x)$-arvon omaavat solmut eri järjestyksessä kuin SMA*-algoritmi. Näistä syistä SMA*-algoritmin on mahdollista löytää ratkaisu ennen kuin se on käynyt läpi kaikki ne solmut, jotka A*-algoritmi käy läpi. Vaikka A*-algoritmin toteuttaisi siten, että se ottaisi huomioon myös solmujen $g(x)$-arvon, eivät A*- ja SMA*-algoritmit siltikään välttämättä kävisi solmuja läpi samassa järjestyksessä. Tämä johtuu siitä, että SMA*-algoritmi laajentaa solmut vain osittain, eikä välttämättä ole tietoinen samoista solmuista kuin A*-algoritmi. Jotkin aluksi lupaavalta vaikuttavat reitit voivat myöhemmin osoittautua huonoiksi, joten A*-algoritmi saattaa tutkia montakin tällaista reittiä, kun taas SMA*-algoritmi voi olla tietoinen vain yhdestä reitistä, joka voi osoittautua lyhimmäksi reitiksi.

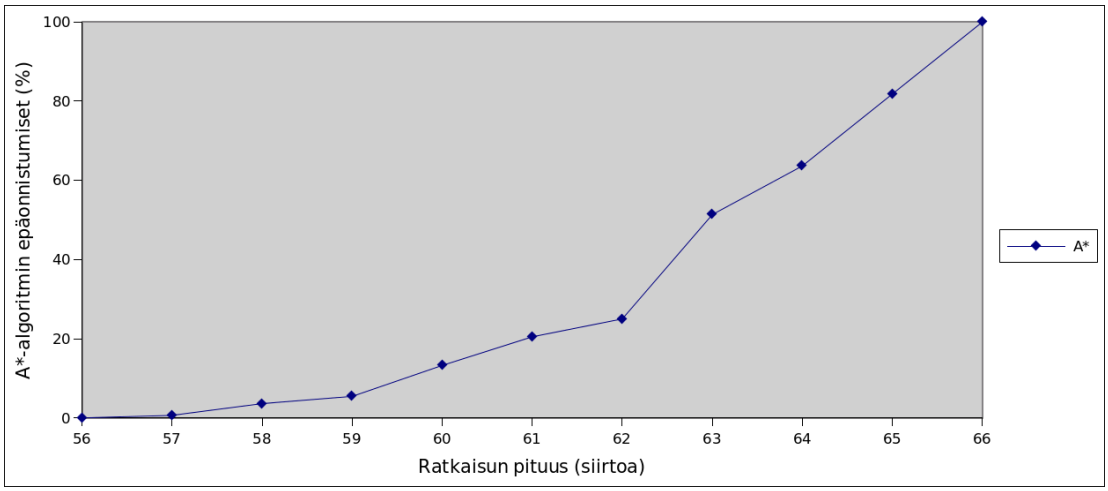
5 Tulosten yhteenveto

Tässä luvussa esitetyt tulokset on saatu reitinhakualgoritmien toimintaa simuloivalla ohjelmalla. Simulaatio suoritettiin Intel Q6600 -prosessorilla varustetulla tietokoneella, jonka käyttöjärjestelmänä toimi Ubuntu Linux. Ohjelma vaatii toimiakseen Javan version 1.7, ja samaa versiota käytettiin tulosten laskemiseen. Java-virtuaalikoneella oli käytössään 5 gigatavua muistia simuloinnin aikana. Vertailukelpoiset tulokset saatiin aikaan luomalla sattumanvarainen 15-puzzle, joka ratkaistiin erikseen jokaisella algoritmilla. Kaikki algoritmit käyttivät heuristiikkana Manhattan-parietäisyyttä. Ratkaisun löytämiseen kulunut aika sekä algoritmin läpikäymien solmujen määrä tallennettiin tietokantaan taulukointia varten. Tulokset on koostettu noin kolmen tuhannen 15-puzzlen otoksesta ja niistä lasketuista keskiarvoista. Tulevissa luvuissa olevissa kaavioissa ei esiinny 15-puzzlen tapauksia, joissa lyhin reitti on ollut 55 siirtoa tai vähemmän. Nämä tapaukset on jätetty pois kaavioiden selkeyttämiseksi, koska algoritmien erot alkavat näkyä selkeämmin vasta 56 siirron kohdalla. Luvussa 5.1 käydään läpi A*-algoritmin tuloksia, sekä verrataan sen toimintaa IDA*-algoritmiin. Luvussa 5.2 verrataan IDA*-algoritmin sekä IEA*-algoritmin tulosten eroja. Luku 5.3 käsittelee SMA*-algoritmin tuloksia erilaisilla muistikapasiteeteilla.

5.1 A* ja IDA*

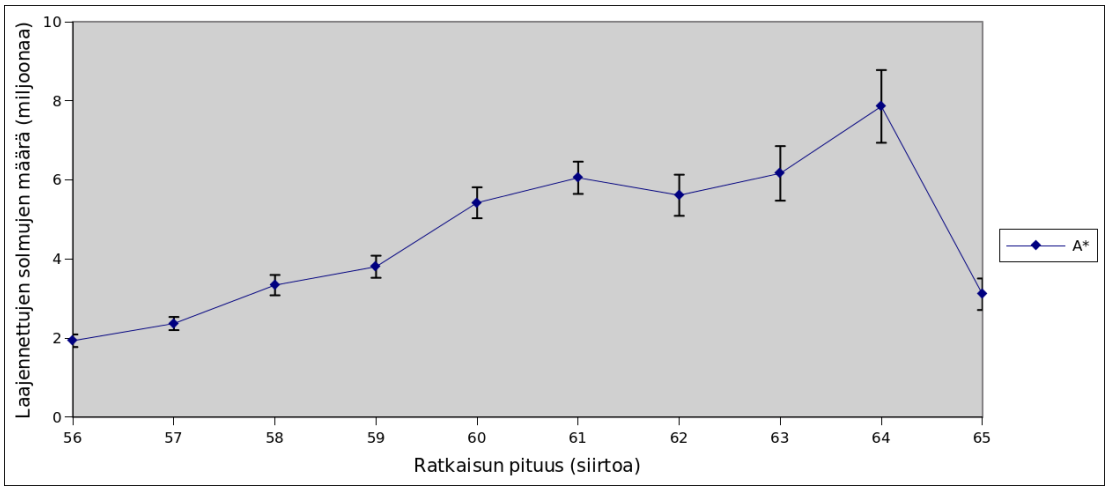
A*-algoritmi on siis optimaalinen reitinhakualgoritmi, mutta sen suurin ongelma on sen liiallinen muistin käyttö. Koska A*-algoritmi käyttää muistia eksponentiaalisesti suhteessa ratkaisun pituuteen, on todennäköistä että reitinhaku epäonnistuu muistin loppumisen takia. Nykyisten tietokoneiden nopea muistikapasiteetin kasvaminenkaan ei pysty ratkaisemaan A*-algoritmin ongelmaa, sillä vaikka muistin määrä kasvaisi moninkertaisesti, monien ongelmien hakuavaruus kasvaa eksponentiaalisesti suhteessa niiden tapauksen kokoon. Vaikka jokin hyvin varustettu tietokone pystyisikin aina ratkaisemaan 15-puzzlen A*-algoritmia käyttäen, 24-puzzle olisi todennäköisesti kuitenkin mahdotonta ratkaista.

Simuloinnin aikana osoittautui, että 5 gigatavua on liian vähän muistia, jotta A*-algoritmi pystyisi ratkaisemaan jokaisen 15-puzzlen permutaation. Kuvasta 4 ilmenee kuinka pidemmillä ratkaisun pituuksilla A*-algoritmi ei enää pystynyt ratkaisemaan ongelmaa muistirajoituksen vuoksi. A*-algoritmi pystyi ratkaisemaan kaikki 15-



Kuva 4: A*-algoritmin reitinhakujen prosentuaalinen epäonnistuminen.

puzzlet, joiden ratkaisun pituus oli 56 siirtoa tai vähemmän. Tämän jälkeen jotkin 15-puzzlen permutaatiot osoittautuvat liian vaativiksi A*-algoritmille, mikäli niiden ratkaisun pituus oli 57-65 siirtoa. Käytetystä otoksesta A*-algoritmi ei pystynyt ratkaisemaan yhtään 66 siirtoa tai enemmän vaatinutta 15-puzzlea, vaikka se teoriassa olisikin mahdollista. Tämä osoittaa hyvin sen, että ratkaisun pituuden kasvaessa A*-algoritmi vaatii enemmän muistia. Kuvasta 4 voi myös nähdä A*-algoritmin muistivaativuuden eksponentiaalisuuden.



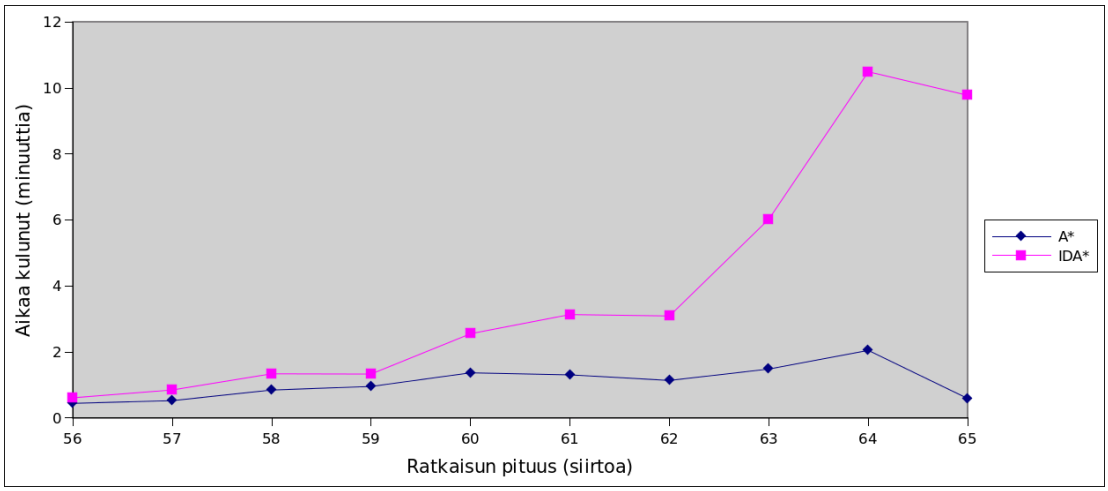
Kuva 5: A*-algoritmin laajentamien solmujen lukumäärä (keskiarvo ja sen keskivirhe) eri ratkaisun pituuksilla.

A*-algoritmin keskiarvon keskivirhe (kuva 5) kasvaa sitä suuremmaksi, mitä pidempi ratkaisun pituus on, lukuun ottamatta 65 siirron tulosta. Ensimmäinen syy kasvuun on otoksen pieneneminen ratkaisun pituuden kasvaessa. Sattumanvaraisesti luo-

dun 15-puzzlen ratkaisun pituus jakautuu melko tasaisesti pituuden 53 ympärille, ja erittäin lyhyiden sekä erittäin pitkien ratkaisujen 15-puzzlet ovat harvinaisempia kuin 15-puzzlet joiden ratkaisujen pituudet ovat lähellä 53 siirtoa. Toinen syy kasvuun on suurempi hajonta laajennettujen solmujen määrässä. Luonnollisestikin pidempi ratkaisu vaatii useampia läpikäytyjä solmuja, mutta joissain tilanteissa ratkaisu voi löytyä erittäin pienellä määrällä läpikäytyjä solmuja. Kolmas syy keskivirheen kasvuun on A*-algoritmin epäonnistuminen ratkaisun löytämisessä. Koska ratkaisemattomia tuloksia ei ole otettu huomioon keskivirhettä laskiessa, pienentää se otoskokoa entisestään. 65 siirron erikoisen tuloksen selittää se, että yli 80 prosenttia otoksesta on jouduttu hylkäämään, jolloin ainoat ratkaistut 15-puzzlet ovat olleet ratkaistavissa pienellä määrällä solmuja, eikä niiden solmujen määrissä ole ollut kovin suurta hajontaa. Kuva 5 esittää pelkästään A*-algoritmin laajentamien solmujen lukumäärän keskiarvon ja keskiarvon keskivirheen eri ratkaisun pituuksilla, mutta ei kerro mitään A*-algoritmin ajankäytöstä. Käytännössä kuva A*-algoritmin keskiarvon keskivirheestä ajankäytön suhteen on lähes identtinen kuvan 5 kanssa, joten kuva 5 on riittävä ilmaisemaan molempien muuttujien keskiarvojen keskivirheet. Keskiarvojen keskivirheiden samankaltaisuus myös viittaa siihen, että laajennettujen solmujen määrä korreloi vahvasti A*-algoritmin ajankäytön kanssa. Tästä voidaan päätellä, että yhden solmun laajentaminen vie aina lähes saman verran aikaa, eivätkä tietorakenteet hidasta algoritmin toimintaa merkittävästi kasvaessaan suuremmiksi.

Vähän muistia käyttävänä vaihtoehtona A*-algoritmille toimii IDA*-algoritmi. Siinä missä A*-algoritmi joutuu pitämään kaiken työn tallessa muistissa, IDA*-algoritmi vaatii vain minimaalisen määrän muistia. Näin suuri ero muistin käytössä voi aluksi vaikuttaa selvästi siltä, että IDA*-algoritmi ei pärjäisi A*-algoritmille ollenkaan, ja että IDA*-algoritmi joutuisi tekemään paljon turhaa työtä laajentaessaan samoja solmuja yhä uudestaan. Kuvan 6 tulokset kuitenkin osoittavat, että IDA*-algoritmi menestyy hyvin vähäisestä muistin käytöstä huolimatta. Suurin syy IDA*-algoritmin tehokkuuteen on sen yksinkertaisuus, jolloin sen ei tarvitse käyttää monimutkaisia tietorakenteita solmujen tallettamiseen, vaan se keskittyy käymään läpi solmuja mahdollisimman nopeasti.

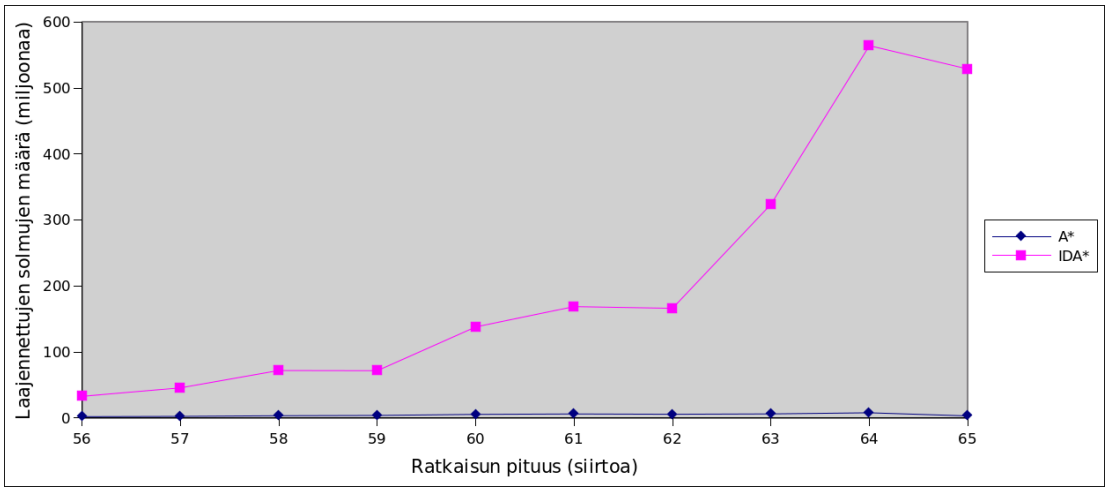
Algoritmien erot ovat erittäin vähäiset kun on kyseessä 59 tai vähemmällä siirrolla ratkaistava 15-puzzle, mutta A*-algoritmi oli kuitenkin aina keskimäärin nopeampi kuin IDA*-algoritmi. Tämän jälkeen IDA*-algoritmi joutuu käyttämään selvästi enemmän aikaa pidempien ratkaisujen löytämiseen, kuin mitä A*-algoritmi joutuu. Kuluneen



Kuva 6: A*- ja IDA*-algoritmien ajankäytön keskiarvot eri ratkaisun pituuksilla.

ajan eksponentiaalinen kasvu johtuu hakurajan kasvamisesta sekä hakuavaruuden laajenemisesta. Koska lyhimmän ratkaisun pituus kasvaa, joutuu IDA*-algoritmi tekemään yhä useampia kierroksia kasvattaakseen hakurajan tarpeeksi suureksi, jotta lyhin reitti mahtuisi hakurajan sisään. Jokainen ylimääräinen kierros on edellistä hitaampi käydä läpi, koska hakuavaruus kasvaa eksponentiaalisesti suuremmaksi jokaisen hakurajan noston yhteydessä. Kuvaa 6 tulkitessa on kuitenkin otettava huomioon kuva 4, eli A*-algoritmin epäonnistumiset. Se on tärkeää siksi, koska A*-algoritmin käyttämä aika ei kuvan 6 mukaan juurikaan kasva ratkaisun pituuden kasvaessa. Ajankäytön pienen kasvun A*-algoritmin kohdalla selittää vaikeimpien ongelmien huomiotta jättäminen muistin loppumisen vuoksi. Mikäli A*-algoritmilla olisi ollut enemmän muistia, ja se olisi pystynyt ratkaisemaan kaikki samat ongelmat kuin IDA*-algoritmikin, olisi A*-algoritmin käyrä ollut hieman nykyistä jyrkempi, mutta silti selvästi loivempi kuin IDA*-algoritmin käyrä.

Suurin ero A*- ja IDA*-algoritmien välillä on niiden laajentamien solmujen määrä, kuten kuva 7 hyvin ilmaisee. Vaikka kuvaa 7 tulkittaessa tulee myös ottaa huomioon kuvassa 4 esiintyvät epäonnistumisten määrät, pysyy A*-algoritmin laajentamien solmujen määrä erittäin pienenä verrattuna IDA*-algoritmiin. Verrattaessa kuvia 6 ja 7 voi hyvin huomata, kuinka IDA*-algoritmin käyttämän ajan sekä laajennettujen solmujen määrän käyrät ovat lähes täysin samanlaiset. Samankaltaisuus johtuu juuri siitä, ettei IDA*-algoritmi käytä minkäänlaisia tietorakenteita, jolloin ratkaisuun käytetty aika korreloi erittäin voimakkaasti laajennettujen solmujen määrän kanssa. Tämä tarkoittaa sitä, että IDA*-algoritmi käyttää jokaisen solmun laajentamiseen lähes täysin saman ajan.



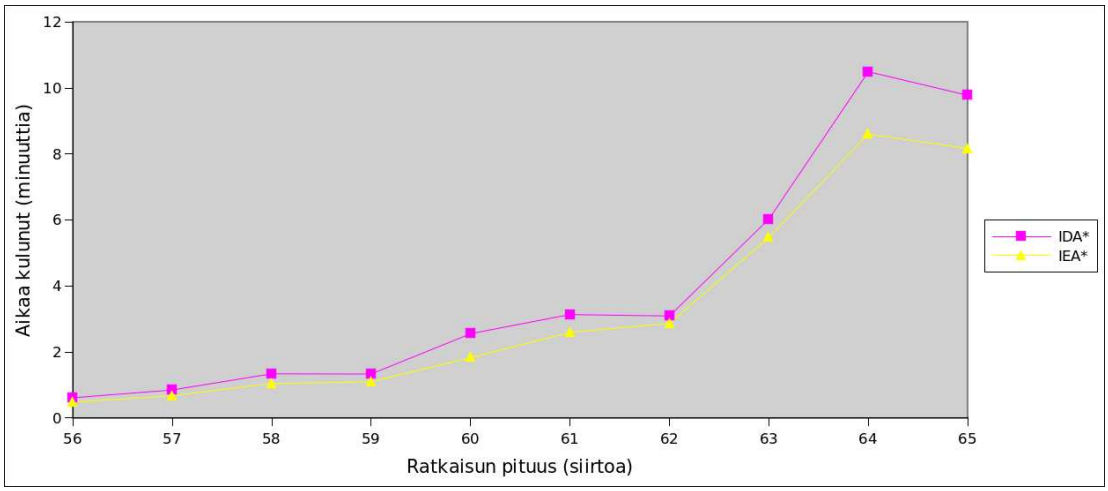
Kuva 7: A*- ja IDA*-algoritmien laajennettujen solmujen määrien keskiarvot eri ratkaisun pituuksilla.

A*- ja IDA*-algoritmit ovat kaksi ääripäätä muistin käytön suhteen, mutta tehokkuudeltaan kuitenkin yllättävän tasa-arvoiset. Siinä missä A*-algoritmi etenee hitaammin mutta älykkäämmin, IDA*-algoritmi pyrkii käymään läpi solmuja mahdollisimman tehokkaasti. A*-algoritmin tietorakenteet ja muistin hyötykäyttö ovat tehokkaampi vaihtoehto, mikäli ratkaistava ongelma ei vaadi paljon muistia. Vaikeimmissa ongelmissa A*-algoritmi epäonnistuu muistin loppuessa, mutta IDA*-algoritmi pystyy ratkaisemaan vaikeimmatkin ongelmat ilman liiallista muistin kuormittamista, mikäli aikaa ja laskentatehoa on riittävästi.

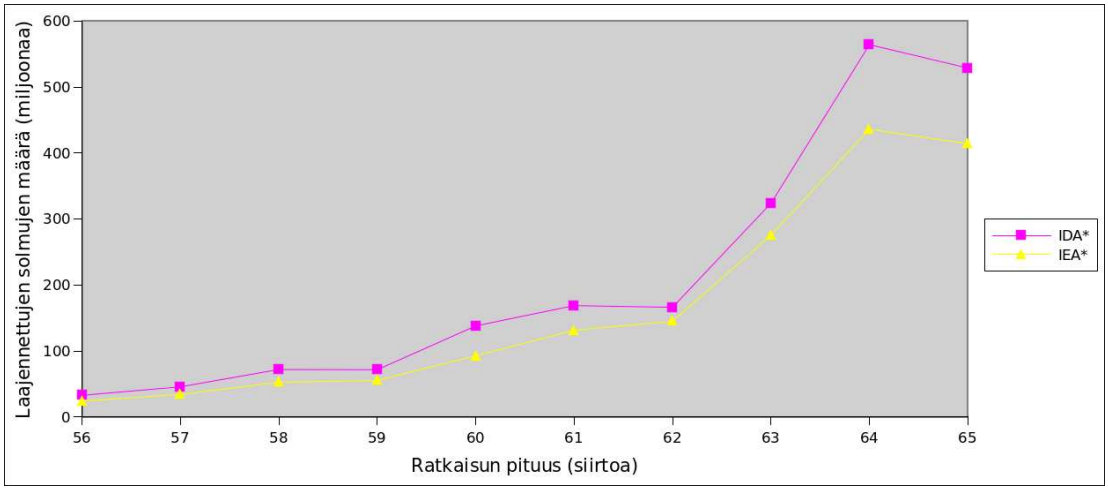
5.2 IDA* ja IEA*

Edellisessä luvussa huomattiin IDA*-algoritmin olevan pätevä vaihtoehto A*-algoritmille paljon muistia vaativissa ongelmissa. IEA*-algoritmi pyrkii parantamaan IDA*-algoritmin tehokkuutta käyttämällä enemmän muistia, mutta huomattavasti vähemmän kuin A*-algoritmi. Vaikka IEA*-algoritmi olikin keskimäärin nopeampi kuin IDA*-algoritmi, jäi näiden kahden algoritmin ero melko pieneksi, kuten kuvasta 8 voi huomata.

Kuvien 8 ja 9 samankaltaisuus johtuu algoritmien lähes identtisestä $etsi$-funktiosta, mikä tarkoittaa sitä, että IEA*-algoritmin ainoa etu on siinä, että se käy keskimäärin vähemmän solmuja läpi kuin IDA*-algoritmi, kuten kuva 9 näyttää. Joissakin tapauksissa voi käydä kuitenkin niin, että IEA*-algoritmi käy läpi enemmän solmuja

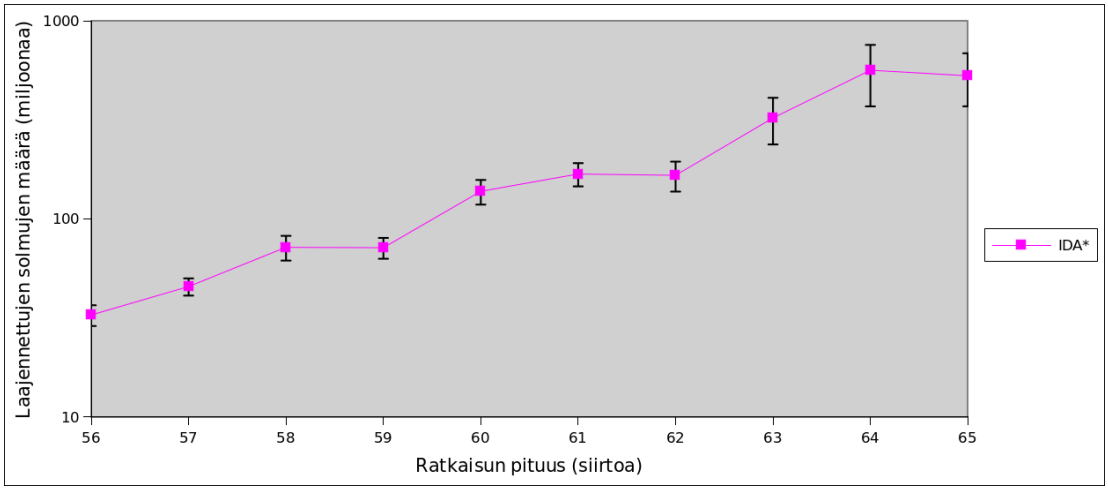


Kuva 8: IDA*- ja IEA*-algoritmien ajankäytön keskiarvot eri ratkaisun pituuksilla.



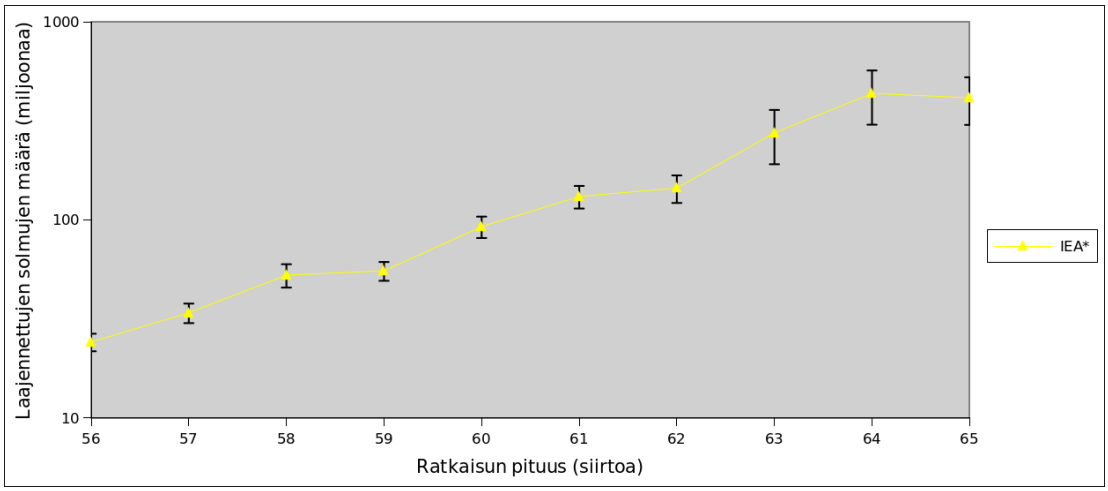
Kuva 9: IDA*- ja IEA*-algoritmien laajennettujen solmujen määrien keskiarvot eri ratkaisun pituuksilla.

kuin IDA*-algoritmi, mutta useimmissa tapauksissa näin ei ole. Koska molemmat algoritmit käyttävät samaa periaatetta rajoitetun syvyyshaun hakurajan kasvattamiseen, algoritmit käyvät läpi yhtä monta kierrosta kunnes hakuraja on tarpeeksi suuri. Viimeistä kierrosta lukuun ottamatta kierrosten aikana laajennettujen solmujen määrä on molemmilla algoritmeilla lähes sama, koska molemmat joutuvat käymään läpi koko syvyysrajoitetun hakupuun ennen kuin voidaan olla varmoja ettei lyhintä reittiä voida löytää, jolloin hakurajaa täytyy nostaa. On myös tärkeää huomata, että algoritmien erot ovat hieman pienemmät (noin 5%) kuvassa 8 kuin kuvassa 9. Tämä johtuu siitä, että IEA*-algoritmi joutuu käyttämään ylimääräistä aikaa tietorakenteiden kanssa, mikä hidastaa algoritmin toimintaa hieman. IEA*-algoritmi käyttää siis keskimäärin hieman enemmän aikaa solmun laajentamiseen kuin IDA*-algoritmi, mutta pystyy tietorakenteidensa ansiosta joissakin tapauksissa vähentämään läpikäytävien solmujen määrää huomattavasti. Keskimäärin ero jää kuitenkin melko pieneksi.



Kuva 10: IDA*-algoritmin laajentamien solmujen lukumäärä (keskiarvo ja sen keskivirhe) eri ratkaisun pituuksilla.

Kuvista 10 ja 11 voidaan huomata, että molempien algoritmien hajonta on hyvin samanlaista. Lyhimmillä ratkaisun pituuksilla keskiarvojen keskivirheet ovat pienempiä, koska niiden kohdalla otos on ollut suurempi kuin pidemmillä ratkaisun pituuksilla. Pidemmillä ratkaisun pituuksilla hajonta on myös luonnollisestikin suurempaa. Luvussa 4.1 on selitetty keskiarvon keskivirhettä tarkemmin A*-algoritmin osalta, mikä soveltuu myös kuviin 10 ja 11, lukuun ottamatta A*-algoritmin epäonnistumisten määrää. Kuvat 10 ja 11 kuvaavat vain laajennettujen solmujen lukumäärien keskiarvojen keskivirheitä, mutta kaaviot algoritmien ajankäytön keskiarvojen keskivirheistä olisivat olleet lähes identtisiä kuvien 10 ja 11 kanssa, joten ne on jätetty pois.



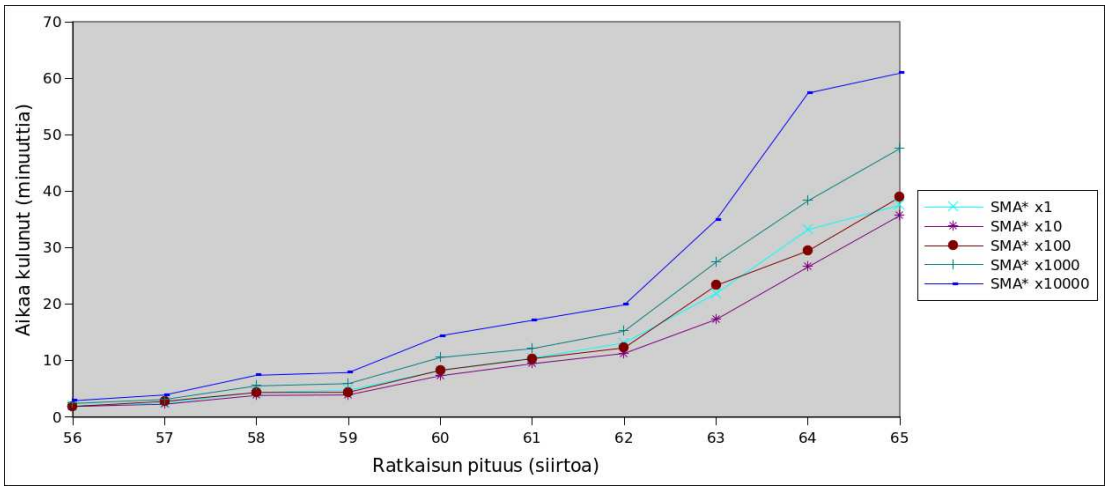
Kuva 11: IEA*-algoritmin laajentamien solmujen lukumäärä (keskiarvo ja sen keskivirhe) eri ratkaisun pituuksilla.

IEA*-algoritmin hyödyntämät tietorakenteet, eli avoin ja suljettu lista, eivät kata kovinkaan suurta osuutta läpikäydyistä solmuista, jolloin etu IDA*-algoritmiin verrattuna ennen viimeistä kierrosta jää vähäiseksi. Mikäli IEA*-algoritmi hyödyntäisi enemmän muistia, voisi se jättää käymättä läpi paljon enemmän solmuja jokaista kierrosta kohden. Suurin ero algoritmeissa esiintyy viimeisellä kierroksella, kun hakuraja on tarpeeksi suuri lyhimmän reitin löytämiseen. IDA*-algoritmi aloittaa etsimisen jokaisella kierroksella lähtösolmusta, kun taas IEA*-algoritmi aloittaa etsimisen avoimen listan lupaavimmasta solmusta. On odotettua, että IEA*-algoritmi löytää lyhimmän reitin vähemmillä solmujen laajennuksilla kuin IDA*-algoritmi, sillä IEA*-algoritmi hyödyntää muistia enemmän kuin IDA*-algoritmi. IEA*-algoritmin avoin lista todennäköisesti ohjaa etsintää oikeaan suuntaan, mutta on myös mahdollista, että avoimen listan lupaavimmat solmut eivät kuulu lyhimmälle reitille. Tällaisessa tapauksessa IDA*-algoritmi mahdollisesti löytää lyhimmän reitin maaliin nopeammin kuin IEA*-algoritmi, joka avoimen listan solmujen takia saattaa joutua tutkimaan paljon enemmän solmuja kuin IDA*-algoritmi.

5.3 SMA*

Muihin tässä tutkielmassa esitettyihin algoritmeihin verrattuna SMA*-algoritmi on ainutlaatuinen, sillä se on ainoa algoritmi jolle voi määrittää sen käyttämän muistimäärän tarkalleen. Periaatteessa SMA*-algoritmi pystyy toimimaan IDA*-algoritmin tavoin käyttämällä minimaalisen määrän muistia, mikäli lyhimmän reitin muistivaativuus on

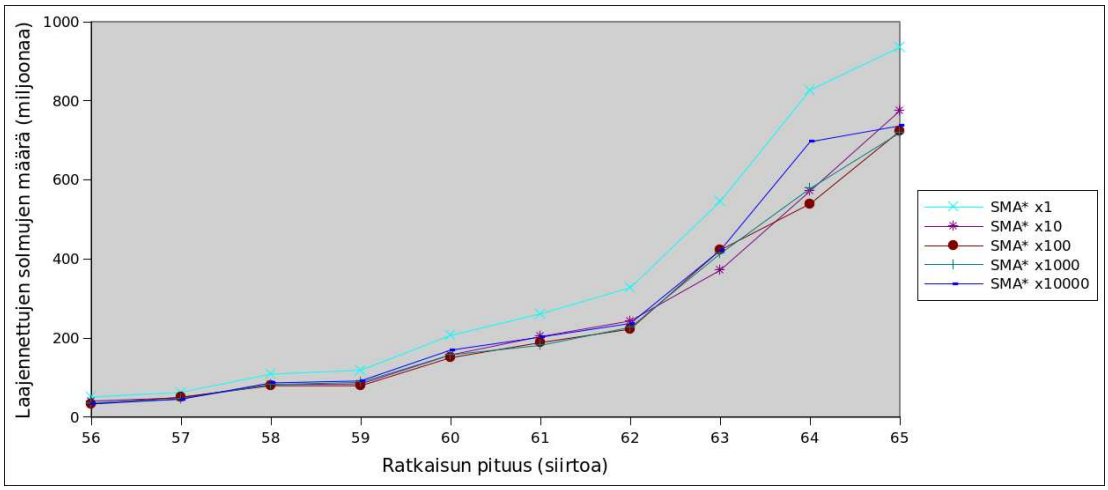
tiedossa, mutta myös A*-algoritmin tavoin, mikäli SMA*-algoritmille annetaan käyttöön kaikki mahdollinen muisti. Vaikka kaikki käytettävissä oleva muisti ei olisi riittävä ratkaisun löytymiseen, SMA*-algoritmi pystyisi jatkamaan etsintää unohtamalla huonoja solmuja, kun taas A*-algoritmin toiminta loppuisi, eikä reittiä löytyisi. Russellin (1992) väite SMA*-algoritmin toimimisesta A*-algoritmin tavoin, mikäli SMA*-algoritmilla on käytettävissään vähintään saman verran muistia kuin A*-algoritmilla, lupaa SMA*-algoritmista erittäin hyvää vaihtoehtoa A*-algoritmille.



Kuva 12: SMA*-algoritmin ajankäytön keskiarvot eri muistimäärillä.

SMA*-algoritmin toiminta vaihtelee sen mukaan, kuinka paljon sillä on käytössään muistia. Kuvassa 12 näkyy SMA*-algoritmin toiminta eri muistimäärillä. Kuvassa nimen perässä näkyvä luku kertoo kuinka monta kertaa lyhimmän ratkaisun verran SMA*-algoritmilla on käytössään muistia. SMA* x1 tarkoittaa algoritmilla olevan vain sen verran muistia, että lyhin reitti on mahdollista löytää, SMA* x10 tarkoittaa algoritmilla olevan kymmenen kertaa lyhimmän reitin verran muistia ja niin edelleen. Aiempien tulosten perusteella voisi olettaa SMA*-algoritmin toimivan nopeammin, mitä enemmän sillä on käytössään muistia, koska esimerkiksi A*-algoritmi hyödyntää paljon muistia, ja on sen takia erittäin tehokas algoritmi. Sama ei päde SMA*-algoritmin kohdalla, kuten kuvasta 12 voi huomata. Vaikuttaisi siltä, että SMA*-algoritmin toiminta hidastuu, mitä enemmän sille antaa muistia käytettäväksi tietyn rajan jälkeen. Algoritmin tehokkuus alkaa heiketä mikäli muistia on sata kertaa lyhimmän ratkaisun verran tai enemmän. Jos muistia on käytössä vain vähimmäismäärä, eli saman verran kuin IDA*-algoritmi käyttäisi, algoritmi toimii noin samalla tehokkuudella, mikäli muistia olisi käytössä sata kertaa enemmän. Parhaiten algoritmi näyttäisi toimivan silloin, kun muistia on käytettävissä noin kymmenkertaisesti lyhimmän ratkaisun verran.

Erikoiset tulokset eivät noudata yleistä normia, eli että enemmän resursseja olisi aina parempi. SMA*-algoritmin tapauksessa se tuntuisi menevän lähes täysin toisin päin. On selvää, että mitä enemmän muistia SMA*-algoritmille antaa, sitä enemmän se joutuu tekemään työtä. Algoritmin alkuperäinen idea kuitenkin oli, että mitä enemmän muistia SMA*-algoritmille antaisi, sitä vähemmän sen täytyisi tehdä työtä, sillä sen ei tarvitsisi käydä läpi unohdettuja solmuja niin usein. Voi myös olla, että ylimääräinen työ tulee tietorakenteiden ylläpidosta, vaikka SMA*-algoritmin käyttämän tietorakenteen operaatiot toimivatkin logaritmisessa ajassa.



Kuva 13: SMA*-algoritmin laajennettujen solmujen määrien keskiarvot eri muistimäärillä.

Voisi olettaa SMA*-algoritmin laajentavan sitä vähemmän solmuja, mitä enemmän muistia sillä on käytössä, koska silloin sen ei tarvitse unohtaa solmuja niin usein. Kuvan 13 osoittamat tulokset ovat kuitenkin ristiriidassa tämän oletuksen kanssa. SMA* x1 laajentaa eniten solmuja, mikä oli odotettavissa sen vähäisen muistin käytön takia. Muut muistimäärät tuntuvat laajentavan lähes saman verran solmuja ratkaisun pituudesta riippumatta, vaikka niiden järjestys vaihtelee melko paljon. On vaikeaa määrittää vähiten solmuja laajentavaa muistimäärää, mutta SMA* x100 vaikuttaisi useimmiten olevan tässä suhteessa tehokkain. On myös mielenkiintoista huomata, että suurin muistimäärä vaikuttaisi olevan toiseksi huonoin vaihtoehto SMA* x1:n jälkeen. Jos SMA*-algoritmi joutuu tekemään enemmän työtä, mitä enemmän sille annetaan muistia käytettäväksi, täytyy algoritmissa olla jokin vakava virhe, tai sitten se tarvitsee toisenlaisen tietorakenteen toimiakseen tehokkaammin.

Algoritmin nopeuteen voivat vaikuttaa monet asiat, mutta SMA*-algoritmin laajentamien solmujen määrään vaikuttavat vain heuristiikka sekä algoritmille annettu muistin

määrä. Heuristiikka on näiden mittausten aikana pysynyt samana, jolloin ainoa vaikuttava tekijä laajennettujen solmujen määrään on SMA*-algoritmille annettu muistin määrä. Mittaustulokset kuitenkin osoittavat, että muistin lisäämisestä on hyötyä vain tiettyyn pisteeseen asti, jonka jälkeen lisämuisti alkaa olemaan merkityksetöntä, tai jopa haitallista. Kuvan 13 perusteella muistin lisääminen ei vaikuta kovin haitallisesti laajennettujen solmujen määrään, mutta kuvasta 12 käy ilmi, että liiallinen muisti vaikuttaa haitallisesti lopulliseen suoritusaikaan. Tarkkaa rajaa tälle on vaikea määritellä, mutta 15-puzzlen tapauksessa se on pienempi kuin sata kertaa lyhimmän reitin pituus. Liiallisen muistin käytön raja on todennäköisesti ongelmakohtainen, mutta sellainen raja on kuitenkin olemassa aina, sillä tehokkaatkin tietorakenteet muuttuvat hitaiksi, mikäli ne kasvavat liian suuriksi, jolloin solmujen uudelleen läpikäyminen on tehokkaampaa kuin niiden muistissa pitäminen.

Tulokset myös osoittavat vääräksi Russellin (1992) väitteen SMA*-algoritmin toimimisesta samoin kuin A*-algoritmi, mikäli SMA*-algoritmilla on enemmän muistia käytössään kuin mitä A*-algoritmi käyttää saman ongelman ratkaisemiseen. Yhdessäkään tapauksessa SMA*-algoritmi ei löytänyt ratkaisua vähemmällä määrällä laajennettuja solmuja, vaikka muistia olikin käytössä paljon enemmän kuin mitä A*-algoritmilla. Niissäkin tapauksissa missä SMA*-algoritmilla oli tarpeeksi muistia käytössään, eikä sen tarvinnut unohtaa solmuja ollenkaan, kävi se läpi useampia solmuja kuin A*-algoritmi. Vaikka SMA*-algoritmin erilainen solmujen läpikäymisjärjestys teoriassa mahdollistaakin lyhimmän reitin löytymisen vähemmällä määrällä laajennettuja solmuja kuin A*-algoritmilla, käytännössä se ei kuitenkaan näytä tapahtuvan. Teoriassahan syvyyshakukin pystyisi löytämään reitin nopeammin kuin A*-algoritmi, mutta todennäköisyys siihen olisi erittäin pieni, ja isommalla otoksella A*-algoritmi olisi kuitenkin selvä voittaja.

Valitettavasti SMA*-algoritmi ei pääse lähellekään A*-algoritmin tehokkuutta. Myös IDA*- sekä IEA*-algoritmit toimivat tehokkaammin kuin SMA*-algoritmi, vaikka SMA*-algoritmilla olisi käytössään miten vähän tai paljon muistia tahansa. Tulokset ovat kuitenkin yllättäviä, sillä SMA*-algoritmin idea teoriassa on erittäin nerokas. Siinä missä IDA*-algoritmin idea tuntui aluksi toimimattomalta, mutta toimikin käytännössä erittäin tehokkaasti, SMA*-algoritmi toimii juuri päinvastoin. Teoriassa hyvä idea ei toimikaan käytännössä niin hyvin.

6 Pohdinta ja yhteenveto

A*-algoritmi oli kaikista tutkielmassa olevista algoritmeista tehokkain, mutta myös ainoa algoritmi, joka ei pystynyt ratkaisemaan kaikkia sille annettuja 15-puzzlen permutaatioita. IDA*-algoritmin yksinkertaisuus ja solmujen unohtaminen ja jatkuva uudelleen läpikäyminen vaikuttivat aluksi erittäin tehottomalta, mutta käytännössä algoritmi näytti toimivan erittäin hyvin. IEA*-algoritmi pyrki parantamaan IDA*-algoritmia vähentämällä uudelleen laajennettavien solmujen määrää. Teoriassa tehokkaalta vaikuttava idea toimi myös käytännössä, mutta ei niin hyvin kuin olisi voinut odottaa. Ero IDA*- ja IEA*-algoritmien välillä oli pettymys, sillä IEA*-algoritmi vaikutti aluksi siltä, että se olisi pystynyt vähentämään laajennettujen solmujen määrää huomattavasti enemmän. SMA*-algoritmi taas vaikutti olevan erittäin nerokas ja ainutlaatuinen idea, mutta käytännössä se ei toiminut juuri ollenkaan, vaikka sille olisi antanut enemmän muistia käytettäväksi kuin A*-algoritmille.

Vaikka tämä tutkielma sisältääkin huolellisesti mitattuja tuloksia algoritmien toiminnasta, ovat ne kuitenkin vain suuntaa antavia tuloksia. Algoritmien toimintaan vaikuttavat monet asiat, ja tässä tutkielmassa algoritmeista on tutkittu vain kuinka ne toimivat Javan virtuaalikoneella Manhattan-parietäisyys-heuristiikkaa käyttäen 15-puzzlen ympäristössä. Ohjelmointikieli, heuristiikka sekä ongelma johon algoritmia käytetään vaikuttavat kaikki vahvasti algoritmin toimintaan.

Java on oliopohjainen ohjelmointikieli, joka tunnetusti käyttää paljon muistia. Jokainen Javan luoma olio vaatii vaihtelevan määrän lisämuistia (Roubtsov, 2002) ohjelmoijan käyttämien muuttujien lisäksi. Tämä lisämuistin käyttäminen on Javan toiminnalle olennaista, vaikka sille ei yksinkertaisimmissa ohjelmissa olisikaan käyttöä. Tähän lisämuistiin ei ohjelmoija myöskään pääse käsiksi mitenkään, eikä sitä voi ottaa pois käytöstä, vaikka ohjelmoija tietäisikin, ettei sille ole tietyissä tapauksissa käyttöä. Javan varaama lisämuisti on jo pelkästään Object-luokassa 8 tavua oliota kohden, ja jokainen muu luokka on kyseisen luokan aliluokka. Jokainen Javan luoma olio varaa siis vähintään 8 tavua ylimääräistä muistia, mutta mahdollisesti myös enemmän. Varsinkin paljon olioita muistissa pitävä ohjelma voi kuormittua liikaa pelkästään tämän lisämuistin takia. Esimerkiksi A*-algoritmin tapauksessa, kun muistissa on useita miljoonia olioita, Javan varaaman lisämuistin osuus on merkittävä. Java on kuitenkin yksi yleisimmin käytetyistä ohjelmointikielistä, joten sen käyttäminen tässä tutkielmassa on perusteltua. Toinen mielenkiintoinen ohjelmointikieli olisi ollut C (Wikibooks,

2014), sillä siinä ohjelmoijalla on lähes täydet valtuudet muistin käytön suhteen. Silloin A*-algoritmin olisi voinut toteuttaa siten, että se olisi käyttänyt vähemmän muistia jokaista laajennettua solmua kohden, kuin mitä Javalla toteutettu A*-algoritmi käytti. Ohjelmointikieli itsessään ei kuitenkaan vaikuta läpikäytyjen solmujen järjestykseen, joten algoritmien laajentamien solmujen määrät olisivat olleet samat, mikäli simulaattori olisi toteutettu esimerkiksi C-kielellä.

Yleisesti algoritmien testaamisessa käytetty 15-puzzle on yksinkertainen toteuttaa, ja sen avulla voi helposti mitata erilaisia tuloksia. Se kuvastaa kuitenkin vain yhdenlaista ongelmaa, ja useat reitinhakuongelmat voivat olla hyvinkin erilaisia. A*-algoritmin tapauksessa 15-puzzle on hyvä, koska sen avulla A*-algoritmin muistikapasiteetti on helppo ylittää. IDA*- ja IEA*-algoritmien kohdalla 15-puzzle oli sikäli ongelmallinen, että kummankaan algoritmin hakuraja ei syventynyt missään vaiheessa huomattavasti. Manhattan-parietäisyyttä käyttäen hakuraja kasvoi aina 2 siirtoa suuremmaksi jokaisen kierroksen jälkeen. Muissa reitinhakuongelmissa hakuraja voi mahdollisesti kasvaa suuremmaksi erittäin nopeasti, jolloin tarvittavien kierrosten määrä on pienempi, mikä nopeuttaa kummankin algoritmin toimintaa huomattavasti, varsinkin jos loppupään kierroksia on vähemmän. SMA*-algoritmin tapauksessa 15-puzzle oli erittäin vaikea tapaus. SMA*-algoritmi unohtaa aina huonoimmalta vaikuttavan solmun, ja pyrkii muistamaan vain unohdettujen solmujen korkeimman $f(x)$-arvon. Tällöin SMA*-algoritmi tietää, että suuren $f(x)$-arvon omaavat solmut eivät ole lupaavalla reitillä. Solmun $f(x)$-arvon kasvattaminen siis muistuttaa SMA*-algoritmia huonosta reitistä, mikäli se joudutaan unohtamaan. SMA*-algoritmin ongelma 15-puzzlen suhteen on sama kuin IDA*- ja IEA*-algoritmeilla, eli solmujen $f(x)$-arvojen pienet erot. Tämä näkyy SMA*-algoritmin toiminnassa edestakaisena liikkeenä, eli algoritmi ensin laajentaa yhtä reittiä kunnes toinen reitti vaikuttaa lupaavammalta, jonka jälkeen ensimmäinen reitti unohtuu kun toista laajennetaan. Tämän jälkeen ensimmäinen tai jokin muu reitti vaikuttaa lupaavammalta, jolloin nykyinen reitti unohdetaan ja uusi laajennetaan. Tämähän on käytännössä SMA*-algoritmin idea, mutta 15-puzzlen tapauksessa edestakainen liike on erittäin toistuvaa, eivätkä solmujen $f(x)$-arvot kasva huomattavasti, samoin kuin IDA*- ja IEA*-algoritmien tapauksissa. Toisenlaisessa ongelmassa olisi mahdollista, että jotkin reitit ovat selkeästi huonompia, jolloin niitä ei ensimmäisen laajennuksen jälkeen enää toista kertaa käytäisi läpi.

Heuristiikka on erittäin tärkeä osa tässä tutkielmassa käsiteltyjen algoritmien toimintaa, ja se erottaa nämä algoritmit perinteisistä epäheuristisista algoritmeista. Heuris-

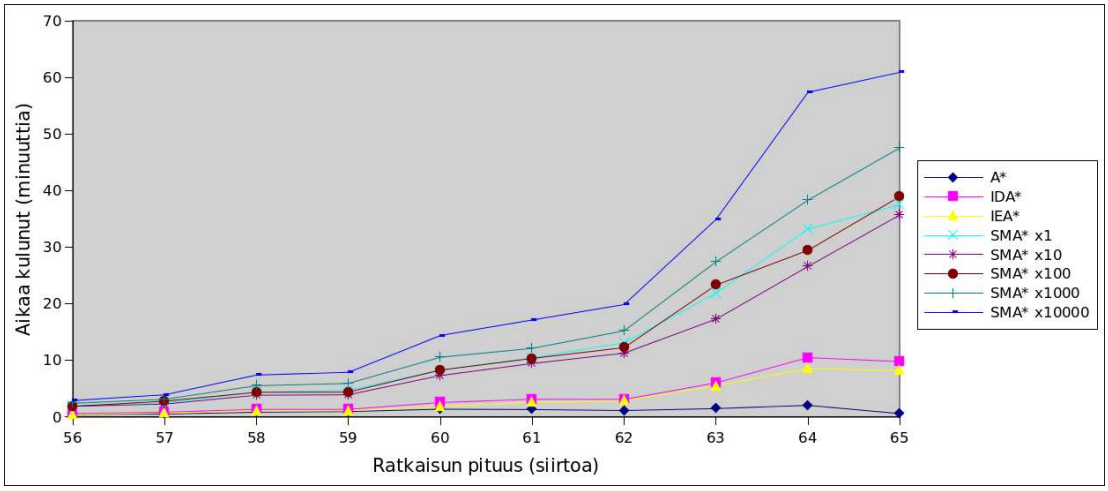
tiikan valinnalla on myös suuri vaikutus algoritmien toimintaan. Algoritmit toimivat ilman heuristiikkaakin, jolloin $h(x)$-arvo on jokaisella solmulla 0, mutta silloin algoritmien tehokkuus heikkenee huomattavasti. Ilman heuristiikkaa A*-algoritmi joutuisi laajentamaan huomattavasti enemmän solmuja, sillä heuristiikka ei ohjaisi sitä oikeaan suuntaan, jolloin sen epäonnistumisten määrä olisi suurempi kuin Manhattan-parietäisyyttä käytettäessä. IDA*- ja IEA*-algoritmit aloittaisivat yhden siirron hakurajalla, ja hakuraja kasvaisi vain yhdellä jokaisen kierroksen jälkeen ilman heuristiikkaa 15-puzzlen tapauksessa. Muuten algoritmien toiminnassa ei tapahtuisi muutoksia, sillä ne aina käyvät läpi kaikki solmut, jotka ovat hakurajan sisällä. Ensimmäiset kierrokset eivät juurikaan vaikuta algoritmien nopeuteen, sillä pienen hakurajan takia hakupuu jää erittäin pieneksi. Vasta myöhemmät kierrokset vaativat enemmän aikaa, ja mikäli heuristiikka pystyy vähentämään myöhempien kierrosten määrää, algoritmien toiminta nopeutuu huomattavasti. SMA*-algoritmin tapauksessa heuristiikka sekä ohjaa algoritmin toimintaa oikeaan suuntaan kuten A*-algoritmiakin, mutta myös vähentää mahdollista edestakaista liikettä eri polkujen välillä. Heuristiikan avulla solmujen $f(x)$-arvot ovat isompia, jolloin unohdettujen polkujenkin $f(x)$-arvot ovat luonnollisestikin isompia. Tällöin laajennettavaa polkua ei tarvitse vaihtaa niin usein. Täydellisen heuristiikan tapauksessa A*- ja SMA*-algoritmit laajentaisivat vain ne solmut, jotka ovat lyhimmällä reitillä maalisolmuun, kun taas IDA*- ja IEA*-algoritmit joutuisivat käymään läpi vain yhden kierroksen, mutta saattaisivat joutua laajentamaan erittäin paljon solmuja, mikäli hakuraja on suuri.

Kaikessa tutkimustyössä on aina mahdollisuus virheille, eikä tämä tutkielma ole poikkeus. Erilaisia merkittäviä virheitä tällaisessa tutkimuksessa voisi olla esimerkiksi algoritmien toiminnan ymmärtäminen väärin, algoritmien virheellinen toteutus, tulosten virheellinen mittaaminen ja väärä tulkinta. Algoritmien ymmärtäminen ei ollut vaikeaa, vaikka SMA*-algoritmi vaikuttikin aluksi hieman monimutkaiselta. Tämän suhteen en usko tutkielmassa esiintyvän virheitä. Algoritmien toteuttaminen käytännössä onnistui myös hyvin, ja huolellisen testaamisen jälkeen algoritmit toimivat kuten pitikin. Ainoastaan SMA*-algoritmin suhteen on mahdollista, että sen olisi voinut toteuttaa tehokkaamminkin. Vaikka algoritmi toimiikin oikein, ei se välttämättä ole toteutettu täysin samoin kuin Russell (1992) sen toteutti, mutta puutteellisen dokumentoinnin takia tästä ei ole varmuutta. Tulosten mittaamiseen käytettiin erillistä järjestelmää häiriöiden minimoimiseksi, jolloin mitatut tulokset ovat kelvollisia. Mittaamisen käytetty ohjelma on myös hyvin testattu virheiden varalta. Tässä tutkielmassa käsitellyt mittaukset ovat yksinkertaisia numeerisia arvoja, joita on hyvin helppo tulkita. Tästä syystä tulosten

väärin tulkitseminen on erittäin epätodennäköistä.

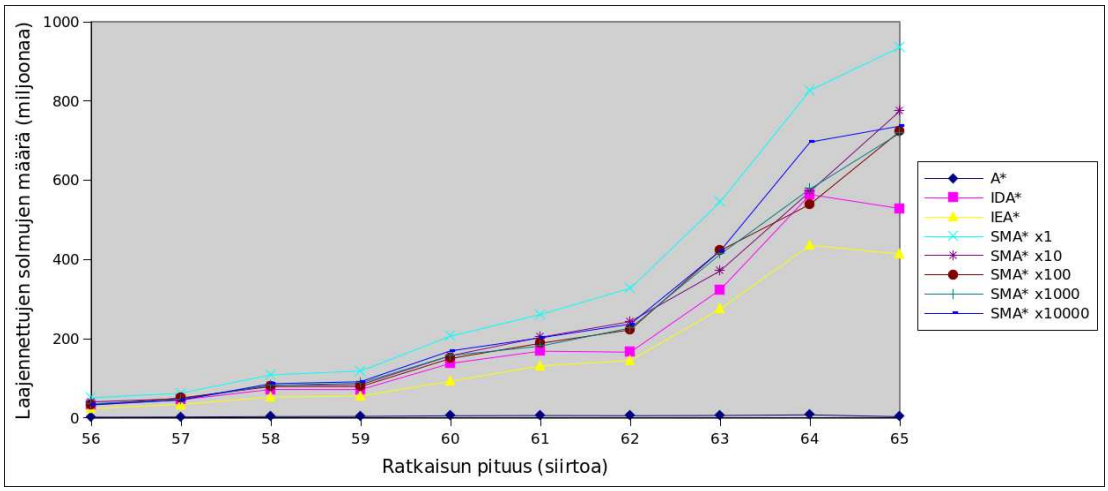
Vaikka tämä tutkielma käsitteleekin edellä olevia reitinhakualgoritmeja vain tietyltä kannalta, kuvaa se kuitenkin hyvin algoritmien suhteita toisiinsa niin tehokkuuden kuin niiden käytännön toteuttamisenkin kannalta. Tutkielmaa olisi voinut laajentaa käsittelemään eri heuristiikkoja sekä eri ohjelmointikieliäkin, mutta tällöin tutkielma olisi kasvanut liian suureksi. Jatkotutkimusta ajatellen nämä olisivat erittäin mielenkiintoisia asioita, mutta tässä tutkielmassa käytiin läpi vain Manhattan-parietäisyysheuristiikka sekä Java-ohjelmointikieli.

Lopuksi tämän tutkielman päättää kattava yhteenveto algoritmien toiminnasta sekä toteutuksesta. Vaikka luvussa 4 käytiinkin läpi algoritmien toteuttaminen käytännössä, ja luvussa 5 esitettiin tulokset algoritmien toiminnasta, suhde näiden kahden asian välillä on käsittelemättä. Kuvat 14 ja 15 ovat yhteenvetoja aiemmin esiintyneistä kaavioista algoritmien vertailun helpottamiseksi, eivätkä ne sisällä uusia tuloksia.



Kuva 14: Kaavio kaikkien algoritmien ajankäytön keskiarvoista.

Vaikka kuvissa 14 ja 15 algoritmien järjestys pysyy lähes samana, on tärkeämpää huomioida kuvan 14 tuloksia, sillä ohjelmoijaa ei välttämättä kiinnosta laajennettujen solmujen määrä vaan hakuaika. A*-algoritmi on kummassakin tapauksessa tehokkain, vaikkakin marginaalisesti verrattuna IDA*- ja IEA*-algoritmeihin. Ero olisi todennäköisesti ollut pienempi, mikäli A*-algoritmi olisi pystynyt ratkaisemaan jokaisen sille annetun 15-puzzlen. IDA*- ja IEA*-algoritmien tulokset ovat lähes samanlaiset, mutta IEA*-algoritmi on hieman IDA*-algoritmia parempi sekä ajassa että laajennettujen solmujen määrässä. Vaikka näiden kahden algoritmin laajentamien solmujen määrä ei eroa merkittävästi SMA*-algoritmin eri kierrosten laajentamien solmujen määristä,



Kuva 15: Kaikkien algoritmien laajennettujen solmujen määrien keskiarvot eri ratkaisun pituuksilla.

ajallisesti IDA*- ja IEA*-algoritmit eroavat SMA*-algoritmista huomattavasti.

Toteutuksen suhteen IDA*-algoritmi oli selkeästi helpoin ja yksinkertaisin toteuttaa, eikä sen ohjelmoinnin aikana esiintynyt juuri minkäänlaisia suurempia ongelmia. IEA*-algoritmin idea oli hieman vaikeampi ymmärtää, mutta ohjelmoinnin kannalta se ei eronnut IDA*-algoritmista muuten kuin yksinkertaisen lisätyn tietorakenteen verran. A*-algoritmin ainoa suurempi ongelma oli oikean tietorakenteen löytäminen. Koska A*-algoritmi käyttää valtavan määrän muistia, täytyy solmuja tallettavan tietorakenteen olla erittäin tehokas. SMA*-algoritmi oli kaikkein ongelmallisin algoritmi toteuttaa, ja lopulta se oli myös kaikkein heikoin algoritmi.

A*-algoritmi on erittäin hyvä algoritmi vaikeisiin ongelmiin, mikäli järjestelmällä on tarpeeksi muistia käytössään. Vaikka A*-algoritmi on hieman vaikeampi toteuttaa kuin IDA*-algoritmi, toimii se tehokkaammin useammissa tapauksissa. A*-algoritmi pystyy myös hyödyntämään tarkempia heuristiikkoja paremmin kuin IDA*-algoritmi. IDA*-algoritmi on erittäin helppo ja nopea reitinhakualgoritmi toteuttaa, mutta myös erittäin tehokas. Hyvin optimoituna se voi olla yhtä tehokas, tai jopa tehokkaampi kuin A*-algoritmi, varsinkin jos hakupuun haarautuvuus on pieni. Pienen muistin käytön takia IDA*-algoritmia voi käyttää myös vähän muistia omaavissa järjestelmissä, esimerkiksi palvelimissa, jotka tarjoavat reitinhakuja käyttäjille. Tällöin palvelimen muisti ei kuormitu, vaikka useampi reitinhaku olisi käynnissä samanaikaisesti IDA*-algoritmia käyttäen. A*-algoritmi voisi tällaisessa tapauksessa kuormittaa muistia liikaa, jolloin palvelin ei pystyisi toteuttamaan useaa reitinhakua samanaikaisesti. IEA*-algoritmi pa-

rantaa IDA*-algoritmia, mutta vain marginaalisesti. IEA*-algoritmi käyttää myös hyvin vähän muistia, jolloin sitä voi myös käyttää vähän muistia omaavissa järjestelmissä. SMA*-algoritmi vaikutti aluksi erittäin nerokkaalta idealta, joka voisi toimia lähes yhtä hyvin kuin A*-algoritmikin, mutta loppujen lopuksi se osoittautui tämän tutkielman heikoimmaksi algoritmiksi niin toteutuksen kuin toiminnankin kannalta. SMA*-algoritmin toteuttaminen on erittäin hankalaa ja monimutkaista, vaikka siihen löytäisi ohjeet, ja algoritmin toiminnan ymmärtäisi täysin. SMA*-algoritmin toiminta oli myös moninkertaisesti hitaampaa verrattuna muihin tutkielmassa käsiteltyihin algoritmeihin, eikä sen käyttämistä voi suositella.

Viitteet

Bauer, B. (1994) *The Manhattan Pair Distance Heuristic for the 15-Puzzle*. Technical Report PC^2 TR-001-94, University of Paderborn, Paderborn, Germany.

Cook, W. (2012) *In Pursuit of the Traveling Salesman*. Princeton University Press, Princeton, New Jersey, USA.

Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009) *Introduction to Algorithms (Third Edition)*. MIT Press, Cambridge, Massachusetts.

Dillenburg, J., Nelson, P. (1993) Improving the Efficiency of Depth-First Search by Cycle Elimination. *Information Processing Letters* **45**(1), 5-10.

Edelkamp, S., Schrödl, S. (2011) *Heuristic Search: Theory and Applications*. Elsevier, USA.

Kaindl, H., Khorsand, A. (1994) Memory-Bounded Bidirectional Search. *Proceedings of the twelfth national conference on Artificial intelligence*, American Association for Artificial Intelligence, Menlo Park, CA, USA, 1359-1364.

Knuth, D. (1973) *The Art of Computer Programming Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Korf, R. (1985) Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence* **27**(1), 97-109.

Korf, R., Taylor, L. (1996) Finding Optimal Solutions to the Twenty-Four Puzzle. *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2*, AAAI Press, Portland, Oregon, 1202-1207.

Korf. R. (2000) Recent Progress in the Design and Analysis of Admissible Heuristic Functions. *Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation*, SARA 2000, Horseshoe Bay, USA, 45-55.

Oracle (2011) *TreeMap (Java Platform SE 7*. WWW-sivusto, `https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html` (7.12.2014)

Oracle (2014) *Java Software*. WWW-sivusto, `https://www.oracle.com/`

java/index.html (20.12.2014)

Pearl, J. (1984) *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Pólya, G. (1957) *How To Solve It*. Princeton, Princeton University Press.

Potts, C., Krebsbach, K. (2012) Iterative-Expansion A*. *25th International FLAIRS Conference*. Marco Island, Florida, USA.

Roubtsov, V. (2002) *JavaWorld*. WWW-sivusto, http://www.javaworld.com/article/2077496/testing-debugging/java-tip-130--do-you-know-your-data-size-.html (27.10.2014)

Russell, S. (1992) Efficient Memory-Bounded Search Methods. *Proceedings of the 10th European conference on Artificial intelligence*, John Wiley & Sons, Inc. New York, USA, 1-5

Russell, S., Norvig, P. (1996) *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey.

Russell, S., Norvig, P. (2009) *Artificial Intelligence: A Modern Approach (Third Edition)*. Prentice Hall, Upper Saddle River, New Jersey.

Sleator, D., Tarjan, R. (1985) Self-Adjusting Binary Search Trees. *Journal of the ACM*, **32**(3), 652–686.

Slocum, J., Sonneveld, D. (2006) *The 15 Puzzle: How it Drove the World Crazy*. Slocum Puzzle Foundation, Beverly Hills, CA.

Tarjan, R. (1983) *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Thayer, J., Ruml, W. (2008) Faster Than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, AAAI Press, Menlo Park, California, 355-362.

Toth, P., Vigo, D. (2001) *The Vehicle Routing Problem*. SIAM, Philadelphia.

Wikibooks (2014) *C Programming*. WWW-sivusto, https://en.wikibooks.org/wiki/C_programming_language (20.12.2014)