
Polunetsintäalgoritmit (väliaikainen otsikko)

TkK-tutkielma
Turun yliopisto
Tietotekniikan laitos
Labran nimi
2023
Botond Ortutay

TURUN YLIOPISTO
Tietotekniikan laitos

BOTOND ORTUTAY: Polunetsintäalgoritmit (väliaikainen otsikko)

TkK-tutkielma, 22 s., 8 liites.

Labran nimi
Helmikuu 2023

Tähän abstrakti

Asiasanat: tähän, lista, avainsanoista

UNIVERSITY OF TURKU
Department of Computing

BOTOND ORTUTAY: Polunetsintäalgoritmit (väliaikainen otsikko)

Bachelor's Thesis, 22 p., 8 app. p.

Laboratory Name

2 2023

English abstract here

Keywords: here, a, list, of, keywords

Sisällys

1	Johdanto	1
1.1	Tutkielman tarkoitus	1
1.2	Tutkimuskysymykset	1
1.3	Tiedonhakumenetelmät	2
1.4	Tutkielman rakenne	2
2	Taustoitus	3
2.1	Polunetsintä ongelmana	3
2.2	Algoritmeista	4
2.3	Esimerkkejä sovelluskohteista	5
3	Joitain polunetsintäalgoritmeja	7
3.1	Leveyssuuntainen läpikäynti (BFS)	7
3.2	Syvyysuuntainen läpikäynti (DFS)	8
3.3	Dijkstran algoritmi	9
3.4	A*-algoritmi	11
3.5	Hierarkkinen polunetsintä	12
4	Algoritmien sovelluskohteita	14
4.1	Videopelit	14
4.2	Karttaohjelmat	16

4.3	Robotiikka	17
5	Eräiden algoritmien tehokkuuden tarkastelu esimerkkiongelmassa	20
6	Yhteenveto	22
	Lähdeluettelo	23
	Liitteet	
A	Polunetsintäalgoritmien pesudokooditoteutukset	A-1

Kuvat

3.1	BFS ja DFS referenssigraafissa	8
3.2	Dijkstran algoritmi ja A* referenssigraafissa	10

Termistö

A* Eräs polunetsintäalgoritmi. Luetaan A-star tai A-tähti

BFS Breadth First Search, leveyssuuntainen läpikäynti, leveyshaku

CDSA* Context Dependent Subgoalng A*, kontekstiriippuvaisesti osatavoitteistava A*

CH Contraction Hierarchy, sopimushierarkia

DAO Dragon Age: Origins

DFS Depth First Search, syvyysuuntainen läpikäynti, syvyyshaku

ECBS Enhanced Conflict Based Search, tehostettu konfliktipohjainen haku

HNA* Hierarchical pathfinding for Navigation Meshes A*, navigaatioverkkojen hierarkisen polunetsinnän A*

HPA* Hierarchical Pathfinding A*, hierarkisen polunetsinnän A*

NPC Non-Playable Character, ei-pelattava hahmo

PBS Priority Based Search, prioriteettipohjainen haku

RTS Real-Time Strategy, reaaliaikainen strategiapeli

RHCR Rolling Horizon Collision Resolution, rullaavan horisontin törmäystenratkaisu

WCHA* Windowed Hierarchical Cooperative A*, ikkunoitu hierarkinen yhteis-
toimintakykyinen A*

1 Johdanto

1.1 Tutkielman tarkoitus

Suunnitelma kappaleelle 1.1:

- *Harkittu ja kiinnostava aloitus*
- *Käy lyhyesti ja yksinkertaisesti läpi seuraavat asiat:*
 - *Polunetsintäalgoritmejä tarvitaan kun...*
 - *Polunetsintä käsitteenä*
 - *Miski kirjoitin kandidityön juuri tästä aiheesta? (tutkielman perustelu)*
- *Päätä kappale jotenkin näin:*

"Tutkielman tarkoitus on esitellä lukijalle erilaisia polunetsintäalgoritmeja, sekä verrata niiden toimintaa jossakin esimerkkiympäristössä"

1.2 Tutkimuskysymykset

Tutkielmassa pyritään vastaamaan seuraaviin kysymyksiin:

1. **Tutkimuskysymys:** Minkälaisia polunetsintäalgoritmeja on kehitetty?
2. **Tutkimuskysymys:** Miten niitä voidaan käyttää käytännön sovelluksiin?
3. **Tutkimuskysymys:** Miten niiden tehokkuutta voidaan mitata?

1.3 Tiedonhakumenetelmät

Tietoa tämän tutkielman tekoon on haettu IEEE:n Xplore Digital Center-tietokannasta, Web of Science-tietokannasta, sekä Google Scholar-hakupalvelusta. Hakutuloksia rajattiin julkaisuaajan mukaan niin, että suurin osa hakutuloksista on julkaistu vuona 2018 tai sen jälkeen. Myös aihepiirirajausta on käytetty. Hakusanoissa on käytetty osuvempien tulosten löytämiseksi Boolean operaattoreita, sekä sanakatkaisua. Alla on muutama esimerkki käytetyistä hakusanoista:

```
pathfinding AND (grid based OR graph theory) AND "map*"
"pathfinding"AND "video gam*"
comparing AND "pathfinding algorithms"
```

1.4 Tutkielman rakenne

Tutkielman luku 2 taustoittaa seuraavia lukuja. Tarkoitus on, että luvun 2 lukemisen jälkeen lukijalle tulisivat tutuksi polunetsintään liittyvät peruskäsitteet ja taustaihteet, jotta seuraavien lukujen ymmärtäminen helpottuisi. Luvussa 3 käydään läpi muutaman tunnetun polunetsinnän toiminta ja täten pyritään vastaamaan tutkimuskysymykseen **1**. Luvussa 4 käydään läpi joitakin polunetsintäalgoritmien yleisiä käyttökohteita ja pyritään vastaamaan tutkimuskysymykseen **2**. Luvussa 5 taas mitataan useiden eri polunetsintäalgoritmien tehokkuus eräässä esimerkkiongelmassa ja vertaillaan niitä tämän avulla toisiinsa. Lopussa olevassa yhteenvetokappaleessa 6 tulokset kootaan vielä yhteen ja esitetään helpommin luettavassa muodossa.

2 Taustoitus

2.1 Polunetsintä ongelmana

Polunetsintä tarkoittaa tietotekniikan kontekstissa ongelmaa, jossa halutaan koneellisesti löytää, sekä mahdollisesti myös piirtää sallittu polku kahden etukäteen määritellyn pisteen väliin. Useissa polunetsinnän liittyvissä ongelmissa halutaan myös, että löydetty polku olisi jollain tavalla optimaalinen. Optimaalinen polku voisi esimerkiksi olla lyhyempi tai nopeampi kuin muut mahdolliset polut. [1]

Jotta polunetsintäongelmia voitaisiin ratkaista koneellisesti, tarvitsee ne ensin muuttaa matemaattiseen esitysmuotoon. Tämän vuoksi polunetsintäongelmat voidaan jakaa kahteen osaongelmaan: graafigeneraatioon ja polunetsintäalgoritmin käyttöön. Graafigeneraatiota tarvitaan, koska polun löytämiseen käytetyt algoritmit toimivat tyypillisesti graafeissa. Siinä muutetaan polunetsintäongelman alueena toimiva maasto- tai kartta-alue graafimuotoiseksi rakentamalla siitä esimerkiksi luurankomalli (skeletonization). [2] Tämä tutkielma on keskittynyt polunetsintäalgoritmeihin ja niiden käyttöön, eikä graafigeneraatiota tämän vuoksi käsitellä tutkielmassa laajemmin.

** Tässä voisi olla jokin kuva kartan muuttamisesta graafiksi **

Polunetsintäongelmat voidaan myös jakaa yksiagentillisiin (single-agent) ja moniagentillisiin (multi-agent) polunetsintäongelmiin. Yksiagentillisissa ongelmissa ongelma-

alueella liikkuu vain yksi agentti, jolle pyritään löytämään optimaalinen polku jostain lähtöpisteestä johonkin päämäärään. Moniagentillisissa ongelmissa alueella liikkuvia agenteja on useita. Näissä kaikilla agenteilla on oma lähtöpiste ja oma päämäärä ja kaikille tulee löytää optimaaliset polut niin, että agentit eivät törmäile toisiinsa ja niiden välille ei synny reititykseen liittyviä konflikteja. [3] Näistä ongelmatyypeistä tämä tutkielma käsittelee pääasiassa yksiagentilliseen polunetsintään kehitettyjä polunetsintäalgoritmeja.

2.2 Algoritmeista

Tässä tutkielmassa tarkasteltavat algoritmit ovat kaksiulotteisissa ajoaikana muuttumattomissa graafeissa toimivia yksiagenttisia polunetsintäalgoritmeja, ellei toisinkin mainita. Nämä algoritmit voidaan jakaa niiden toimintaperiaatteen mukaan epäinformoituihin (uninformed), informoituihin (informed) ja metaheuristisiin (metaheuristic) hakualgoritmeihin (search algorithms). [4]

Epäinformoidut hakualgoritmit ovat yksinkertaisia algoritmeja, jotka eivät ole tietoisia ongelma-alueensa yksityiskohdista. Näin ollen epäinformoidut hakualgoritmit perustuvat toimintamalliin, jossa kuljetaan graafissa solmukohdalta toiseen kaaria pitkin niin kauan kunnes ollaan löydetty polku lähtösolmusta maalisolmuun. Tätä toimintamallia sanotaan joskus myös sokean haun konseptiksi (blind search). [4]

Informoidut hakualgoritmit sen sijaan käyttävät ongelma-alueesta laskettuja tietoja hyväksi nopeuttaakseen ajoaikaa. Tyypillisesti tämä tehdään laskemalla seuraavaksi läpikäytävien solmukohtien etäisyys maalisolmusta käyttämällä niin kutsuttua heuristista funktiota. Näin ollen jokaiselle graafissa olevalle solmukohdalle n saadaan laskettua sen kautta kulkevan polun hinta $F(n)$ käyttämällä hintafunktiota $F(n) = G(n) + H(n)$, jossa $G(n)$ on solmulle n asti kuljettu matka lähtösolmusta $H(n)$ on heuristisen funktion palauttama arvo. Korkeahintaisia solmukohtia ei tutkita algoritmin ajon aikana, jonka takia tutkittavia solmuja on yhteensä vähemmän

ja algoritmi on nopeampi ajaa.[4]

A^* (luetaan A-tähti tai A-star) on yksi esimerkki informoidusta hakualgoritmista. A^* on yksi suosituimmista polunetsintäalgoritmeista käytännön sovelluskohteista. Tämä johtuu siitä, että A^* on yksinkertainen toteuttaa ja se palauttaa aina optimaalisen polun, mikäli käytetään sopivaa heuristista funktiota.[1] Sen pohjalta on myös kehitetty muita polunetsintäalgoritmeja.[4]

Metaheuristiset algoritmit eivät perustu heuristisiin funktioihin, tai solmukohtien tutkimiseen, vaan ne käyttävät muunlaisia keinoja polkujen etsimiseen.[4] Niitä ei käydä tässä tutkielmassa tarkemmin läpi.

2.3 Esimerkkejä sovelluskohteista

Polunetsintäongelmia joudutaan ratkomaan muun muassa videopeleissä, karttaohjelmissa, erilaisissa simulaatioissa ja robotiikan alalla, [2] sekä esimerkiksi logistiikan alan automaatiassa ja robotisoitujen autojen kehityksessä.[3] Merkittävän suuri osa polunetsintäongelmiin liittyvistä tutkimuksista tehdään nimenomaan videopelien[1][2][5] ja robotiikan[2][6] näkökulmasta.

Polunetsintäongelmia esiintyy videopeleissä, koska niissä on usein tarve simuloida ei-pelattavien hahmojen (non-playable character, NPC) liikkeitä niin, että niille on määritelty lähtö- ja maalipisteet, sekä sallitut ja kielletyt liikkumisalueet. Kun polunetsintäalgoritmeja sovelletaan videopeleihin, on tärkeää, että algoritmit olisivat laskennallisesti tehokkaita. Esimerkiksi reaaliaikaisissa strategiapeleissä (real-time strategy, RTS) pelaaja liikuttaa useita eri yksiköitä eri puolille karttaa merkitsemällä niille pisteitä joiden kautta kulkea. Näissä tilanteissa on pelikokemuksen kannalta tärkeää, ettei eri yksiköille suoritettavat polunetsintäalgoritmit vaikuttaisi pelin sulavuuteen.[1] Permana, Bintoro, Arifitama et al. [5] tutkivat eri polunetsintäalgoritmien toimimista videopelissä, jossa pelaaja rakentaa valmiiksi annetuista palikoista labyrintin ja algoritmi yrittää ratkaista sen. Tutkimuksessa A^* osoittau-

tui tutkituista algoritmeista parhaaksi. Vastaavankaltainen tutkimus käydään yksityiskohtaisemmin läpi tutkielman luvussa 5.

Videopelien lisäksi polunetsintäalgoritmeja sovelletaan myös karttaohjelmissa. Niissä polunetsintää käytetään reitinhakuun, joka on karttaohjelmien eräs päätoiminnallisuus. Reitinhakuun käytetään usein Dijkstran algoritmia,[7] josta puhutaan tarkemmin tutkielman luvussa 3.3. Karttaohjelmissa polunetsintä on kuitenkin melko erilaista kuin videopeleissä. Ensinnäkin on tärkeää, että yksittäisen optimaalisen reitin sijasta reitinhaku palauttaisi käyttäjälle useita vaihtoehtoisia ajoreittejä, joita generoitaisiin lisää ajon aikana siltä varalta, että liikenteen olosuhteet muuttuvat.[8] Poggenhans ja Janosovits [8] käyvät artikkelissaan läpi heidän suunnittelemansa digitaalisen kartta-alustan Lanelet2:n lähestymistapaa reitinhakuun. Polunetsintäalgoritmien käyttö karttaohjelmien reitinhaussa käydään tarkemmin läpi tutkielman luvussa 4.2.

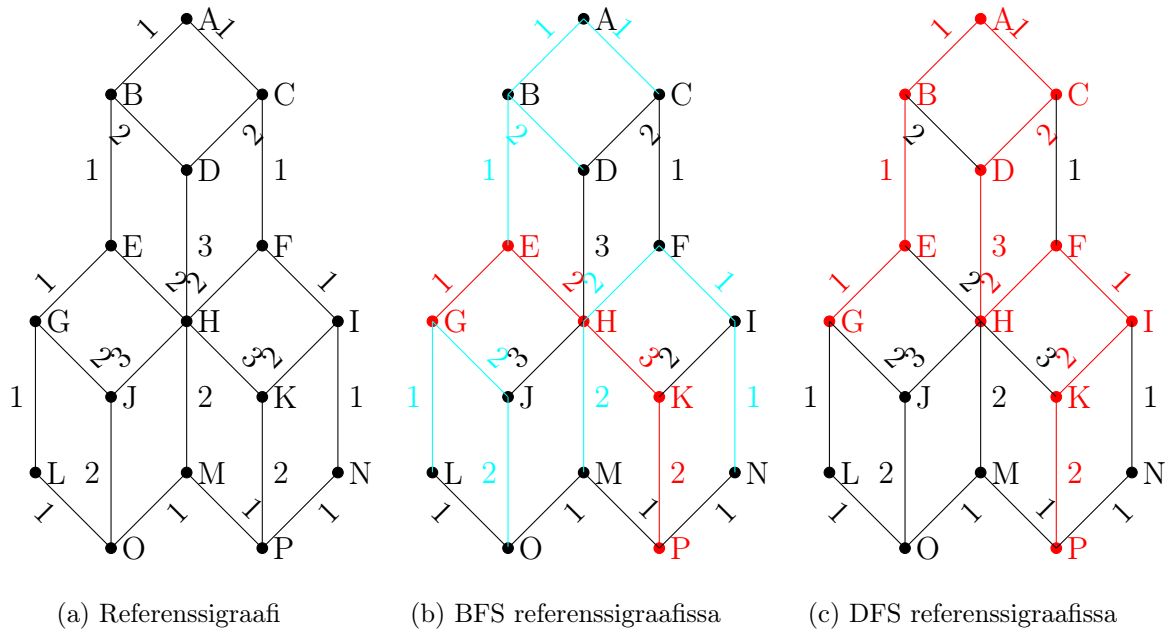
Polunetsintää on tutkittu myös paljon robotiikan näkökulmasta. Robotiikassa polunetsintää joudutaan soveltamaan muun muassa kun kehitetään itseajavia ajoneuvoja ja tehtaassa liikkuvia teollisuusrobotteja.[3] Esimerkiksi Poggenhans ja Janosovits [8] pohtivat artikkelissaan muun muassa sitä, miten itseajaville autoille voitaisiin kehittää karttapalveluita joita ne voisivat soveltaa. Polunetsinnän soveltaminen robotiikassa on hyvin monipuolista, koska ongelmat ovat keskenään erilaisia ja lähestymistavatkin voivat olla erilaisia. Esimerkiksi jos polunetsintää käytetään tilanteessa, jossa tehtaassa liikkuu kymmeniä teollisuusrobotteja eri kohteisiin, niin polunetsintäalgoritmeja voidaan ajaa jokaisessa robotissa erikseen, tai joku keskustietokone voi välittää jokaiselle robotille polun, jotka se ratkaisee moniagentillisesta polunetsintäongelmasta. Liu, Liu, Lu et al. [6] esittävät artikkelissaan Delaunay-kolmiomittaukseen perustuvaa graafigeneraatiota ja tehostetun A*-algoritmin ajoa liikkuvassa robotissa. Polunetsintäalgoritmien soveltamista robotiikan sovelluksiin käydään tarkemmin läpi tutkielman luvussa 4.3.

3 Joitain polunetsintäalgoritmeja

3.1 Leveyssuuntainen läpikäynti (BFS)

Leveyssuuntainen läpikäynti, eli leveyshaku (Breadth First Search, BFS) on epäinformatiivinen hakualgoritmi, joka perustuu sokeaan hakuun.[4] Siinä graafin solmut ryhmitellään eri tasoihin sen mukaan monenko kaaren kautta pitää kulkea lähtösolmista, jotta niihin päästään. Lähtösolmu on siis tasolla 0, siihen yhdistyneet solmut tasolla 1, tason 1 solmuihin yhdistyneet solmut tasolla 2 ja niin edelleen. Leveyshakussa graafin kaikki solmut käydään läpi niin, että tarkistetaan, onko solmussa jo käyty, onko solmu maalisolmu ja mihin solmuihin sillä on yhteys. Sitten tallennetaan solmu läpikäytyjen solmujen listalle ja tieto siitä, mitä kautta solmulle ollaan tultu.[9]

Ajetaan Lawande, Jasmine, Anbarasi et al. [4] perusteella kirjoitetun, ohjelmallistuksen 1 mukaista BFS-algoritmia graafissa 3.1a. Asetetaan lähtösolmuksi G ja maalisolmuksi P. Oletetaan, että algoritmille syötetty graafidata on aakkosjärjestyksessä, jonka takia algoritmi käy läpi solmuja aakkosten mukaan. Tasoksi 0 asetetaan lähtösolmu G. Tason 1 muodostavat siihen yhteydessä olevat solut E, J ja L. Tason 2 soluja ovat tason 1 soluissa kiinni olevat solut, eli B, H ja O. Solmua G ei huomioida, koska siellä on jo käyty. Joitain solmuja voidaan kuitenkin käydä läpi useasti, jos sinne on useampi polku. Algoritmin ajo graafissa 3.1a johtaa kuvan 3.1b lopputulokseen, jossa käytiin läpi syaaninväriset kaaret ja löydettiin punaisella merkitty polku G-E-H-K-P.



Kuva 3.1: BFS ja DFS referenssigraafissa

Ohjelmalistauksen 1 algoritmi lopettaa etsinnän löydettyään yhden polun, mutta jos algoritmiin ei lisätä tätä lopetusehdoksi, niin algoritmi käy läpi kaikki solmut ja kaaret ja löytää kaikki mahdolliset polut.[9] Haittapuoliin kuuluu suuri muistinkulutus tallennettujen polkujen lukumäärän takia,[9] sekä pitkä ajoaika.[5]. Nämä voidaan huomata myös syääninväristen kaarten suuresta määrästä.

3.2 Syvyysuuntainen läpikäynti (DFS)

Syvyysuuntainen läpikäynti, eli syvyysshaku (Depth First Search, DFS) on polunetsintäalgoritmi, joka on myös epäinformoitu sokeaan hakuun perustuva polunetsintäalgoritmi, niin kuin BFS.[4] Algoritmit eroavat keskenään siinä, missä järjestyksessä ne käyvät graafin solmut läpi. Siinä, missä BFS muodostaa eri tasoja ja käy graafin läpi taso kerrallaan, DFS valitsee jokaisella solmulla yhden haaran, jota se lähtee seuraamaan lehtisolmuihin asti.[10] Lehtisolmuja ovat ne, jossa minkään haaran kaari ei etene läpikäymättömään solmuun.[10] Tällöin palataan lehdistä pois-

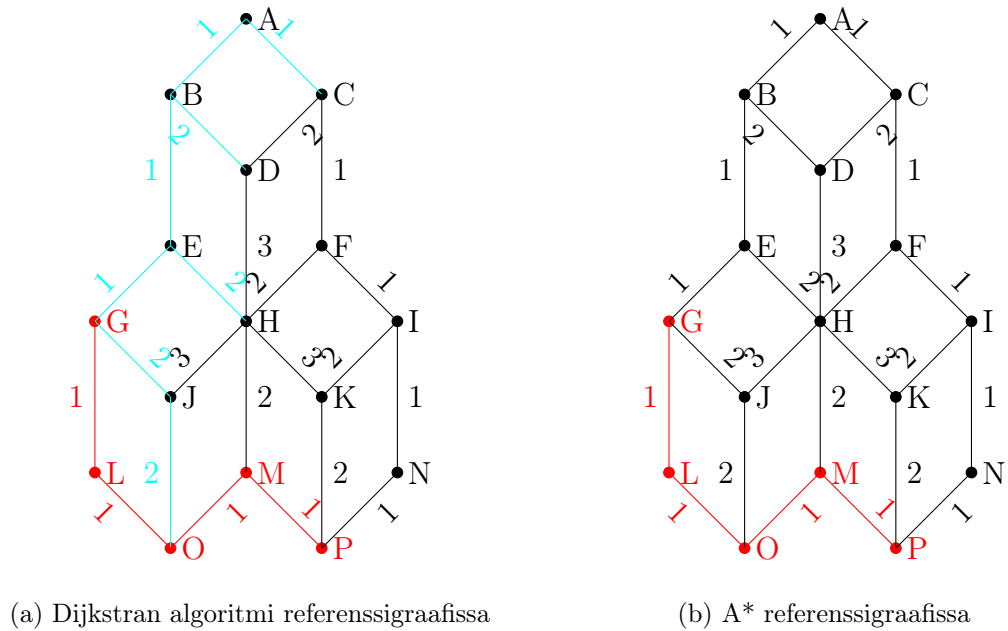
päin niin kauan, että löytyy haara, missä on läpikäymättömiä solmuja, jolloin käydään sen haaran pohjalla. Prosessia toistetaan, kunnes polku löytyy.

Ajetaan Zhao, Xu, Li et al. [10] perusteella kirjoitetun, ohjelmalistauksen 2 mukaista DFS-algoritmia graafissa 3.1a. Ratkaistaan sama ongelma ja tehdään samat lähtöoletukset kuin aliluvussa 3.1. Algoritmin ajo johtaa kuvan 3.1c mukaiseen lopputulokseen, jossa algoritmi löysi polun G-E-B-A-C-D-H-F-I-K-P. Kuten kuvista näkyy, DFS-algoritmin löytämä polku on huomattavasti pidempi kuin BFS-algoritmin löytämä polku. BFS käy kuitenkin läpi useamman kaaren kuin DFS. DFS-algoritmin edut BFS:ään verrattuna ovatkin muistin ja suoritusajan säästyminen, koska harvempi kaari käydään läpi.[4] Toisaalta DFS ei löydä kaikkia mahdollisia reittejä, toisin kuin BFS.[4]

3.3 Dijkstran algoritmi

Dijkstran algoritmi on Edsger Dijkstran vuonna 1956 löytämä epäinformoitu polunetsintäalgoritmi.[4] Dijkstran algoritmi perustuu ahneuden periaatteeseen (the principle of greedy), joka tarkoittaa että joka suorituskerralla valitaan halvin eli kevyin saatavilla oleva solmu.[5] Toisin kuin sokea haku, ahneuden periaate ei pidä jokaista kaarta yhtä hyvänä vaihtoehtona polunetsintäprosessissa. Jokaiselle kaarelle merkitään jokin hinta eli paino, jonka ahneuden periaate huomio. Jos oletetaan, että graafin solmut ovat risteyksiä ja kaaret teitä, on suotavaa ajatella, että eri tiet eri risteyksien välissä ovat eri pituisia, vaikka tämä ei tieverkostosta rakennetuksa graafissa näkyisikään. Voi tulla tilanteita, joissa oikeasti lyhyempi reitti kulkee useamman solmun läpi. Tämän takia graafit painotetaan. Referenssigraafissa 3.1a painot näkyvät jokaisen kaaren kohdalla numeroilla.

Ajetaan Lawande, Jasmine, Anbarasi et al. [4] perusteella kirjoitetun, ohjelmalistauksen 3 mukaista Dijkstran algoritmia graafissa 3.1a. Ratkaistaan sama ongelma ja tehdään samat lähtöoletukset kuin aliluvussa 3.1. Algoritmin ajo johtaa kuvan 3.2a



Kuva 3.2: Dijkstran algoritmi ja A* referenssigraafissa

mukaiseen lopputulokseen. Algoritmi löysi polun G-L-O-M-P. Tämä kulkee viiden solmukohdan läpi ja on näin mitattuna yhtä pitkä kuin reitti jonka BFS löysi referenssigraafista (kuva 3.1b) ja huomattavasti lyhyempi kuin reitti, jonka DFS löysi referenssigraafista (kuva 3.1c). BFS ja DFS eivät kuitenkaan huomioi painotusta, jonka mukaan Dijkstran algoritmin löytämän polun hinta on 4, BFS-algoritmin 8 ja DFS-algoritmin 16. Dijkstran algoritmin löytämä polku on siis annetussa ongelmasa optimaalisin. Dijkstran algoritmia käytetäänkin usein graafiopin lyhimmän polun ongelman ratkaisemiseen.[7] Sitä ei kuitenkaan voi käyttää negatiivisilla painoarvoilla.[4] Dijkstran algoritmi on kuitenkin referenssiongelmassa muisti- ja suoritusaikaintensiivisempi kuin DFS, koska se käy läpi useampia kaaria (Dijkstra 12 verrattuna DFS 10). Eri algoritmeja vertaillaan enemmän luvussa 5.

3.4 A*-algoritmi

A* (luetaan A-tähti tai A-star) on informoitu polunetsintäalgoritmi, joka täydentää Dijkstran algoritmia soveltamalla siihen heuristista funktiota.[1] Heuristinen funktio on jokin matemaattinen funktio, joka palauttaa hinnan, joka mittaa että kuinka hyvä tutkittava solmu on ongelman ratkaisun kannalta. Kuten Dijkstran algoritmi, myös A* valitsee tutkittavakseen halvimmat polut ensin.[6] Dijkstran algoritmista solmukohdan n läpikäynnin hinnan funktio $F(n)$ on sama kuin lähtösolmusta solmulle n johtavan halvimman polun funktio $G(n)$. Siis Dijkstran algoritmista $F(n) = G(n)$. Sen sijaan A* algoritmista yksittäisen solmun n läpikäynnin hinta on $F(n) = G(n) + H(n)$, jossa $H(n)$ on heuristisen funktion palauttama arvo solmukohdalle n . [6] Dijkstran algoritmi voidaan myös käsittää A*-algoritmiksi, jossa $H(n) = 0$. [1]

A*-algoritmin kanssa voidaan käyttää eri heuristisia funktioita riippuen siitä, että mikä heuristiikka toimii parhaiten tutkittavan ongelman kanssa. Tyypillisesti heuristiset funktiot mittaavat käytännössä tutkittavan solmun etäisyyttä maalisolmusta tai liikkeen suuntaa.[11] Tyypillisiä heuristisia funktioita ovat tutkittavan solmun ja maalisolmun välinen Manhattan-etäisyys ja Euklidinen etäisyys.[1] Manhattan-etäisyys tarkoittaa kahden pisteen $P = (x_1, y_1)$ ja $Q = (x_2, y_2)$ välistä etäisyyttä, olettaen liikutaan vain x- ja y-akselien suuntaisesti.[1]

$$d_{\text{Manhattan}}(P, Q) = |x_1 - x_2| + |y_1 - y_2|$$

Ajetaan Mathew ja Malathy [1] perusteella kirjoitetun, ohjelmalistauksen 5 mukaista A*-algoritmia graafissa 3.1a ja tehdään samat lähtöoletukset kuin aliluvussa 3.1. Käytetään heuristisena funktiona janan pituuden kaavaa

$$d(P) = \sqrt{(x_{\text{Maalisolmu}} - x_P)^2 + (y_{\text{Maalisolmu}} - y_P)^2}$$

, jossa P on (x, y) muotoinen piste. Algoritmin ajo johtaa kuvan 3.2b mukaiseen lopputulokseen. Algoritmin löytämä polku G-L-O-M-P on täysin sama kuin Dijkstran algoritmin löytämä polku kuvassa 3.2a. Tähän päädyttiin kuitenkin huomattavasti nopeammin, koska heuristinen funktio ohjasi A*-algoritmia etsimään reittiä oikeasta suunnasta.

A* on hyvin suosittu algoritmi käytännön sovelluksissa, koska se on monipuolinen, helposti toteutettava ja tehokas. Se on useimmiten se algoritmi, josta lähdetään liikkeelle kun polunetsintää pitää käytännössä soveltaa oikean elämän ongelmiin, esimerkiksi videopelikehityksessä tai robotiikassa.[11] A* toimii myös pohjana monelle muulle algoritmille, koska sitä on erittäin helppo muokata ja parannella.[11] Esimerkiksi D*-algoritmi on A*-algoritmiin perustuva polunetsintäalgoritmi, joka on dynaaminen, eli se kykenee reagoimaan etsintäalueen muuttumiseen.[4] Hierarkkisen polunetsinnän A* (Hierarchical Pathfinding A*, HPA*) on hierarkkiseen polunetsintään kehitetty A*-variantti. Lisää hierarkkisesta polunetsinnästä aliluvussa 3.5.[4]

Polunetsintään liittyvissä tieteellisissä tutkimuksissa A* on myös suosittu lähtökohhta. Usein nämä tutkimukset kehittävät jonkun A* variantin johonkin spesifiseen käyttötarkoitukseen.[11] Esimerkiksi Mathew ja Malathy [1] käyttävät tavallista A*-algoritmia, mutta kehittävät siihen uudenlaisen heuristisen funktion ja Liu, Liu, Lu et al. [6] kehittävät A*-pohjaisen polunetsintäalgoritmin käytettäväksi robotiikassa.

3.5 Hierarkkinen polunetsintä

Monet perinteiset polunetsintäalgoritmit skaalautuvat huonosti etsintäalueen kasvassa ja monimutkaistuessa. Esimerkiksi Yiu ja Mahapatra [12] demonstroivat artikkelissaan, kuinka A*-algoritmi kävi läpi suuren osan etsintäalueesta eräässä esimerkkitehtävässä. Tätä varten kehitettiin polunetsintätekniikoita suurille etsintäalueille. Yksi tällainen tekniikka on hierarkkinen polunetsintä.[12]

Hierarkkisessa polunetsinnässä etsintäalue jaetaan pienimpään osa-alueisiin, ni-

in että pidetään kirjaa siitä, että mitkä osa-alueet ovat kytköksissä toisiinsa minkäkin solmujen kautta. Tämän jälkeen osa-alueista muodostetaan ylemmän tason graafi.[12] Ylemmän tason graafista etsitään ensin ylemmän tason polku, joka kertoo, että minkä osa-alueiden kautta varsinainen polku kulkee. Sitten kriittisten osa-alueiden läpi etsitään niiden kautta kulkevat polut. Nämä sitten yhdistetään lopussa varsinaiseksi poluksi. Hierarkkisella polunetsinnällä saadaan yksinkertaistettua polunetsintäongelmaa abstraktion avulla ja tämä nopeuttaa ratkaisun löytämistä merkittävästi. Näin saadut polut eivät kuitenkaan välttämättä ole optimaalisia.[12]

Hierarkkisia polunetsintäalgoritmeja on monia erilaisia, esimerkiksi hierarkkisen polunetsinnän A^* (HPA*) ja navigaatioverkkojen hierarkkisen polunetsinnän A^* (Hierarchical pathfinding for Navigation Meshes A^* , HNA*).[12] Hierarkkiset polunetsintäalgoritmit eroavat toisistaan muun muassa siinä, että mitä algoritmia käytetään jakamaan etsintäalue ja mitä polunetsintäalgoritmia käytetään polkujen löytämiseen. Yiu ja Mahapatra [12] esittelevät artikkelissaan kehittämänsä alueenlöytöalgoritmin (Regions Discovery Algorithm, RDA), jota he vertaavat HPA*-algoritmin sisäänrakennettuun tapaan jakaa ruudukkomuotoinen etsintäalue alaruudukoiksi. Esimerkkipseudokoodi hierarkkisesta polunetsinnästä löytyy ohjelmalistauksesta 7.

4 Algoritmien sovelluskohteita

4.1 Videopelit

Monet nykyaikaiset polunetsintäalgoritmeihin liittyvät tutkimukset tehdään videopeli-teollisuuden käyttöön.[1][2][5] Tämä johtuu siitä, että videopeleissä törmätään usein tilanteisiin, jossa NPC-hahmojen pitää liikkua videopelikartalla niin, että pelaaja ei suoraan itse ohjaa niitä. Tällöin pelin on itse löydettävä reitti kahden pisteen välille. Kun polunetsintäalgoritmeja toteutetaan videopeleissä, joudutaan kiinnittämään huomiota muutamiin erityisvaatimuksiin. Polunetsintäalgoritmien ajaminen ei saa olla liian resurssi-intensiivistä, eli sen pitää pyrkiä kuluttamaan tietokoneen muistia ja suoritinaikaa mahdollisimman vähän. Jotta resurssi-intensiivisyydeltä vältyttäisiin, voidaan löystää polun optimaalisuuteen liittyviä vaatimuksia, jolloin polun ei tarvitse olla enää lyhin mahdollinen.[1]

Esimerkiksi reaaliaikaisissa strategiapeleissä (real-time strategy, RTS) pelaaja komentaa usein eri yksiköitä eri puolelle pelialuetta mekitsemällä niille maalipisteitä. Näitä yksiköitä voi olla jopa satoja, joille pitää löytää polut suurella etsintäalueella huomioiden mahdollisesti muut alueella liikkuvat yksiköt, sekä pelin säännöt siitä, että miten eri yksiköt käytännössä voivat liikkua.[13][1] Nämä monimutkaiset laskutoimitukset on suoritettava pelin logiikan vaatimalla nopeudella. Esimerkiksi videopeliyhtiö BioWare on asettanut säännön, jonka mukaan kaikkien pelialueella liikkuvien agenttien polunetsintä on kyttävä suorittamaan 1-3 ms aikana.[13]

Spittlemeister ja Opdahl [13] kehittivät Pathfinding-in-Pacman-projektin, jossa he sovelsivat polunetsintäalgoritmeja Pac-Manin kaltaiseen peliin. Pelissä pelihahmo liikkuu labyrintissa yrittäen kerätä pisteitä, samalla kun sitä jahtaavat vihollishahmoina toimivat haamut. Pelaaja voittaa kerättyään kaikki pisteet, ja häviää kun haamut koskevat häntä. Peli sisältää toteutukset BFS- ja A*-algoritmeista, sekä pelin tarkoituksiin muokatusta A*-algoritmista, jota projektin raportti kutsuu Context Dependent Subgoalng A*-algoritmiksi (kontekstiriippuvaisesti osatavoitteistava A*, CDSA*) ja ohjelmakoodi kutsuu subGoalAStar-algoritmiksi. CDSA* etsii polun haamulta pelaajahahmolle A*-algoritmilla, mikäli haamu on tarpeeksi lähellä pelaajahahmoa. Muussa tapauksessa CDSA* palauttaa ainoastaan suunnan, jolla haamu pääsee lähemmäs pelaajahahmoa. Tällä tavoin säästetään järjestelmän resursseja rajoittamalla tehtävien laskujen määrää. Pelissä käytetään polunetsintäalgoritmeja haamujen ja pelaajahahmojen välisen polun löytämiseksi ja A*- ja CDSA*-algoritmit käyttävät heuristiikkana Manhattan-etäisyyttä.

Sturtevant ja Geisberger [14] vertaavat tutkimuksessaan erilaisia suurelle etsintäalueelle soveltuvia polunetsintätekniikoita Dragon Age: Origins-videopelissä (DAO) käytettyyn polunetsintään. Nämä polunetsintätekniikat ovat abstraktiahierarkien käyttö, parempien heurististen funktioiden käyttö, sekä sopimushierarkioiden (Contrarction Hierarchy, CH) käyttö. Abstraktihierarkia ja CH ovat hierarkisessa polunetsinnässä (luku 3.5) käytettäviä tapoja muodostaa ylätasen graafi. Abstraktihierarkia muodostaa ylätasen graafin niin, että varsinaisen etsintäalueen eri osa-alueet ovat ylätasen graafin solmuja, kun taas CH muodostaa ylätasen graafin laskemalla alkuperäisen graafin jokaiselle solmulle tärkeystason (importance level) ja poistamalla vähemmän tärkeät solmut graafista. Sturtevant ja Geisberger [14] osoittivat, että hierarkinen polunetsintä joko abstraktiohierarkiaa tai CH:ta käyttäen soveltui parhaiten polunetsintään DAO:n kaltaisissa videopeleissä, koska näillä menetelmillä käytettiin vähiten suoritinta ja muistia. DAO käyttää abstraktiohierarkiaa, mut-

ta Sturtevant ja Geisberger [14] mainitsevat, että pelin polunetsintää olisi voinut tehostaa entisestään käyttämällä suurempia osa-alueita ylätasoon graafin rakentamiseen.

4.2 Karttaohjelmat

Polunetsintäalgoritmit ovat myös keskeisessä osassa karttaohjelmistoissa. Monet karttaohjelmat tarjoavat reitinhakupalveluita, joissa generoidaan kartan eri pisteiden välille reitti käyttäen polunetsintäalgoritmeja. Tämä on usein helppo toteuttaa, koska karttaohjelmat säilyttävät karttadataa usein polunetsintäalgoritmeille ideaalisessa muodossa. Esimerkiksi OpenDrive- formaatti sisältää dataa risteyksistä ja teistä, josta kootaan reititysgraafi.[8]

OpenStreetMap-projekti [15] on yhteisön ylläpitämä avoimen lähdekoodin karttatietokanta, joka tarjoaa sivustollaan myös tietokannan karttadataa käyttävää karttaohjelmaa.[16] OpenStreetMap-karttaohjelma sisältää myös reitinhakumahdollisuuden. Reitinhakumahdollisuus sisältää reitinhakuvaihtoehdot kävelylle, pyöräilylle ja autoilulle käyttäen joko GraphHopperia, OSRM:ää tai Valhallaa.[15]

Nämä ovat niin sanottuja reititysmoottoreita (routing engine), jotka ovat valmiita ohjelmistototeutuksia polunetsintäfunktioille.[17] OpenStreetMapin käyttämä GraphHopper-projekti [17] on Javalla kirjoitettu reititysmoottori. Se käyttää sisäisessä toteutuksessaan Dijkstran algoritmia, A*-algoritmia, sekä hierarkista polunetsintää käyttäen CH-menetelmää ylätasoon graafin luomiseen ja Dijkstran algoritmia polunetsintään. GraphHopper valitsee ongelmaan sopivan polunetsintämenetelmän sen asetusten perusteella.

4.3 Robotiikka

Polunetsintää tutkitaan myös robotiikan näkökulmasta, koska käytännössä kaikki robotit, jotka liikkuvat jossakin ympäristössä, tarvitsevat polunetsintäalgoritmeja tehdäkseen päätöksiä siitä, minne mennä.[11] Liikkuvien robottien polunetsintään liittyy monenlaisia haasteita. Yksi näistä on törmäyksien estäminen.[2] Robotit voivat törmätessään aiheuttaa vakavaa vahinkoa itseensä, ympäristöönsä ja alueella liikkuviin ihmisiin.[11] Esimerkiksi itseajavien autojen tapauksessa törmäykset voidaan pyrkiä estämään huomioimalla muut tienkäyttäjät paikallisessa reitinsuunnittelussa. Tämä ongelma kuitenkin monimutkaistuu kun tarvitsee huomioida erilaisia tienkäyttäjiä. Muut tienkäyttäjät huomioiva paikallinen reitinhakualgoritmi on helpompi suunnitella moottoritielle, missä voidaan odottaa liikkuvan pelkästään autoja ja voidaan odottaa että kaikilla on tarpeeksi tilaa, kuin kaupunkiin, jossa on lisäksi jalankulkijoita ja pyöräilijöitä, sekä määrällisesti enemmän tienkäyttäjiä ja vähemmän tilaa.[8]

Törmäysten estämisen lisäksi robotiikassa tärkeässä roolissa on myös graafigeneraatio. Graafigeneraatio on osa lähes jokaista polunetsintäongelmaa, koska etsintäalue pitää muuttaa graafiksi, jotta polunetsintäalgoritmia voisi käyttää. Graafigeneraatio kuitenkin korostuu robotiikan sovelluksissa, koska toisin kun esimerkiksi videopeleissä tai kartanohjelmissa, tietokone ei hallitse etsintäaluetta, eikä se välttämättä ole tietoinen esteiden sijainnista alueella, tai alueen muutoksesta. Tätä varten robotiikassa käytetään usein niin sanottuja dynaamisia polunetsintäalgoritmeja, jotka kykenevät reagoimaan etsintäalueen muutoksiin. Esimerkkejä näistä ovat A*-algoritmiin perustuva D*-algoritmi[4] ja Liu, Liu, Lu et al. [6] kehittämä DFPA-algoritmi.

Kim, Timothy, Salim et al. [18] esittelevät artikkelissaan kehittämänsä halpaa kuljetusrobotia, joka käyttää polunetsintään A*-algoritmia. Robotille syötetään ensin graafidata, jonka perusteella se sitten etsii lyhimmän polun A*-algoritmillä ja

seuraa sitä maaliin asti. Robotti ei käytä dynaamista polunetsintää, eikä se kykene reagoimaan etsintäalueella tapahtuviin muutoksiin, vaan sille syötetään alueella olleet esteet graafidatan mukana. Lisäksi robotti sijoittaa itsensä graafiin seuraamalla toiminta-alueella maahan kiinnitettyjä mustia viivoja. Kyseinen robotti ei siis mielestäni soveltuisi tosielämän kuljetusrobotiksi, koska se ei pysty reagoimaan ympäristönsä muutoksiin ja se vaatii esiasennettua infrastruktuuria toimiakseen. Tutkimus keskittyi kuitenkin sensorisijoitusten optimisointiin ja käännösmekanismien ja -algoritmien tekemiseen realistisen kuljetusrobotin polunetsintää enemmän.

Hein, Wesselhöft, Kirchheim et al. [19] vertailevat artikkelissaan polunetsintäalgoritmeja varastoissa työskenteleville teollisuusroboteille. Koska varastossa saatavaa liikkua kerralla jopa kymmeniä tai satoja robotteja, kyseessä on moniagentillinen polunetsintäongelma. Lisäksi artikkeli olettaa, että robotit liikkuvat varastossa jatkuvasti lastausalueiden ja varastointialueiden välillä niin, että kun jokin robotti saa sille asetetun tehtävän valmiiksi, sille annetaan uusi satunnainen tehtävä.

Robotin sijaintia ei siis kyetä ennustamaan nykyisen tehtävän suorittamisen jälkeen. Tämän vuoksi artikkelin polunetsintäalgoritmit ajetaan uudestaan sille asetetun aikaikkunan ω mukaan. Tällöin algoritmeille riittää, että löydetyt polut ovat törmäysvapaita ainoastaan seuraavan aikaikkunan alkuun asti, eli korkeintaan ω ajan. Tätä kutsutaan artikkelissa rullaavan horisontin törmäystenratkaisuksi (Rolling Horizon Collision Resolution, RHCR)

Artikkelin vertaamat polunetsintäalgoritmit ovat ikkunoitu hierarkinen yhteistoimintakykyinen A^* (Windowed Hierarchical Cooperative A^* , WHCA*), tehostettu konfliktipohjainen haku (Enhanced Conflict Based Search, ECBS) ja prioriteettipohjainen haku (Priority Based Search, PBS). WCHA* on hierarkinen (luku 3.5) A^* -algoritmi, joka on muokattu toimimaan moniagentillisten ongelmien kanssa ikkunoidusti. ECBS on tehostettu versio CBS-algoritmista, joka etsii polkua iteroituvasti niin, että aina kun törmäyskohta löydetään niin polunetsintä aloitetaan alusta niin, että tör-

mäysmahdollisuus huomioidaan rajoitteena. PBS on yhdistelmä moniagentilliseen ympäristöön mukautettua A*-algoritmia ja CBS-algoritmia. Artikkelin mittauksissa PBS oli marginaalisesti tehokkaampi kuin WHCA* ja ECBS oli niin hidas, että se ei pystynyt tutkimukseen asennettujen aikarajoitteiden mukaan ratkaisemaan kuin yhden kolmesta tutkitusta varastosta.

5 Eräiden algoritmien tehokkuuden tarkastelu esimerkkiongelmassa

Tässä tutkielmassa on verrattu kahta polunetsintäalgoritmia, Dijkstran algoritmia ja BFS:ää, ajamalla niitä koneellisesti monta kertaa satunnaisesti generoiduissa graafeissa ja tutkimalla ajautumisaikoja. Graafit on toteutettu Boost Graph Library C++ kirjaston avulla jonka kehitti Boost-järjestö [20] . Testiohjelma ajaa algoritmit kahdessa Boost Graph Libraryn eri graafitoteutuksessa. Tämän tarkoitus on demonstroida graafien toteutuksen vaikutusta algoritmeihin. Testiohjelma itsessään on kehitetty tätä projektia varten ja julkaistu kokonaisuudessaan avoimen lähdekoodin jakeluun. [21]

Testiohjelma ajaa jokaisella ajokerralla 2000 testiä, jossa jokaista varten generoidaan graafi n solmulla ja $1,25n$ kaarella. Käytetyt graafikoot olivat $n = 64$, $n = 128$ ja $n = 512$. Toteutuksen testisilmukassa oli kutsu, sekä BFS:lle, että Dijkstran algoritmille, sekä listatyypisessä-, että matriisityypisessä graafissa, mutta yksittäisissä testeissä kommentoitiin kaikki muu kuin testattava. Täten kun testattiin esimerkiksi Dijkstran algoritmia matriisityypisessä graafissa, niin kaikki listatyypisiin graafeihin ja BFS:ään liittyvät toimenpiteet oli kommentoitu pois, jolloin ne eivät vaikuta testien tuloksiin.[21]

Väliaikaista tekstiä, jotta näen miltä tulostaulukko näyttää takstin ympäröimänä:
Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry

Taulukko 5.1: Yhteenveto testituloksista

Graafin tyyppi ja solmujen määrä	BFS	Dijkstra
Lista 64	Aikakeskiarvo: 2,805 Aikakeskihajonta: 0,3972	Aikakeskiarvo: 2,975 Aikakeskihajonta: 0,1565
Matriisi 64		
Lista 128		
Matriisi 128		
Lista 512		
Matriisi 512		

turururu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry tururuu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu.

Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu. Dipi diipa diipa doudou. Diipaa didi dou. Tyryry ryryryryry ryryry ryryry ry ryryryry turururu tururuu tu.

6 Yhteenveto

Lähdeluettelo

- [1] G. E. Mathew ja G. Malathy, "Direction based heuristic for pathfinding in video games", teoksessa *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, IEEE, 2015, s. 1651–1657. DOI: 10.1109/ECS.2015.7124867.
- [2] Z. A. Algfoor, M. S. Sunar ja H. Kolivand, "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games", *Int. J. Comput. Games Technol.*, vol. 2015, tammikuu 2015, ISSN: 1687-7047. DOI: 10.1155/2015/736138.
- [3] R. Stern, N. R. Sturtevant, A. Felner et al., "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks", *CoRR*, vol. abs/1906.08291, 2019. eprint: 1906.08291.
- [4] S. R. Lawande, G. Jasmine, J. Anbarasi ja L. I. Izhar, "A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games", *Applied Sciences*, vol. 12, nro 11, 2022, ISSN: 2076-3417. DOI: 10.3390/app12115499.
- [5] S. Permana, K. Bintoro, B. Arifitama ja A. Syahputra, "Comparative Analysis of Pathfinding Algorithms A*, Dijkstra, and BFS on Maze Runner Game", *IJISTECH (International Journal Of Information System & Technology)*, vol. 1, toukokuu 2018. DOI: 10.30645/ijistech.v1i2.7.

- [6] Z. Liu, H. Liu, Z. Lu ja Q. Zeng, "A Dynamic Fusion Pathfinding Algorithm Using Delaunay Triangulation and Improved A-Star for Mobile Robots", *IEEE Access*, vol. 9, s. 20 602–20 621, 2021. DOI: 10.1109/ACCESS.2021.3055231.
- [7] D. Rachmawati ja L. Gustin, "Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem", *Journal of Physics: Conference Series*, vol. 1566, kesäkuu 2020. DOI: 10.1088/1742-6596/1566/1/012061.
- [8] F. Poggenhans ja J. Janosovits, "Pathfinding and Routing for Automated Driving in the Lanelet2 Map Framework", teoksessa *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, 2020. DOI: 10.1109/ITSC45102.2020.9294376.
- [9] R. Rahim, D. Abdullah, S. Nurarif et al., "Breadth First Search Approach for Shortest Path Solution in Cartesian Area", *Journal of Physics: Conference Series*, vol. 1019, kesäkuu 2018. DOI: 10.1088/1742-6596/1019/1/012038.
- [10] L. Zhao, H. Xu, J. Li ja Q. Cai, "A Kind of Map Coloring Algorithm Considering of Depth First Search", teoksessa *2012 International Conference on Industrial Control and Electronics Engineering*, 2012. DOI: 10.1109/ICICEE.2012.175.
- [11] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah ja E. Gunawan, "A Systematic Literature Review of A* Pathfinding", *Procedia Computer Science*, vol. 179, s. 507–514, 2021, 5th International Conference on Computer Science and Computational Intelligence 2020, ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2021.01.034>.
- [12] Y. F. Yiu ja R. Mahapatra, "Regions Discovery Algorithm for Pathfinding in Grid Based Maps", teoksessa *2020 Second International Conference on Transdisciplinary AI (TransAI)*, 2020, s. 84–87. DOI: 10.1109/TransAI49837.2020.00018.

- [13] A. Spittlemeister ja J. Opdahl, *Pathfinding-in-Pacman*, Käytetty projektin raporttia (Full-Project-Report.pdf), sekä lähdekoodia., toukokuu 2017. url: <https://github.com/AndrewSpittlemeister/Pathfinding-in-Pacman>.
- [14] N. Sturtevant ja R. Geisberger, "A Comparison of High-Level Approaches for Speeding Up Pathfinding", *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 6, s. 76–82, lokakuu 2010. DOI: 10.1609/aiide.v6i1.12400.
- [15] OpenStreetMap-projekti, toim., *OpenStreetMap*, <https://www.openstreetmap.org>, OpenStreetMap Foundation, maaliskuu 2023. (viitattu 17.03.2023).
- [16] OpenStreetMap-projekti, toim. "OpenStreetMap - About", OpenStreetMap Foundation. (maaliskuu 2023), url: <https://www.openstreetmap.org/about> (viitattu 17.03.2023).
- [17] GraphHopper-projekti, toim., *GraphHopper*, versio 7.0, GraphHopper GmbH, maaliskuu 2023. url: <https://github.com/graphhopper/graphhopper> (viitattu 17.03.2023).
- [18] J. W. Kim, N. Timothy, A. A. Salim, E. Taniara ja E. Steven, "Implementation of A* Shortest Path Finding Algorithm in a Transport Robot with Robust Turning Mechanism", teoksessa *2022 1st International Conference on Technology Innovation and Its Applications (ICTIIA)*, 2022. DOI: 10.1109/ICTIIA54654.2022.9935869.
- [19] B. Hein, M. Wesselhöft, A. Kirchheim ja J. Hinckeldeyn, "Towards Industry-Inspired Use-Cases for Path Finding in Robotic Mobile Fulfillment Systems", teoksessa *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2022. DOI: 10.1109/ETFA52439.2022.9921501.

-
- [20] Boost-järjestö, *Boost Graph Library*, versio 1.85.0, 28. huhtikuuta 2024. url: https://www.boost.org/doc/libs/1_85_0/libs/graph/doc/index.html.
- [21] B. Ortutay, *graph-tests-2*, versio b1f1d96, 28. huhtikuuta 2024. url: <https://github.com/gitond/graph-tests-2/>.

Liite A Polunetsintäalgoritmien pesudokooditoteutukset

Tämä liitedokumentti sisältää tutkielmassa käsiteltyjen polunetsintäalgoritmien pesudokooditoteutukset.

Ohjelmalistaus 1 Esimerkki BFS-algoritmista (osa 1)

procedure BFSESIMERKKI(*graafidata*, *lahtosolmu*, *maalisolmu*)*vieraillut* \leftarrow *tyhjaLista**kasiteltavat* \leftarrow *tyhjaLista**vanhemmat* \leftarrow *tyhjaLista**polku* \leftarrow *tyhjaLista**kasiteltavat.lisaaLoppuun*(*lahtosolmu*)*vanhemmat*[*lahtosolmu*] \leftarrow -1 **while** *kasiteltavat* \neq *tyhjaLista* **do***kasiteltava* \leftarrow *kasiteltavat.poistaEnsimmäinen*()*vieraillut.lisaaLoppuun*(*kasiteltava*)**if** *kasiteltava* = *maalisolmu* **then****while** *vanhemmat*[*kasiteltava*] \neq -1 **do***polku.lisaaAlkuun*(*kasiteltava*)*kasiteltava* \leftarrow *vanhemmat*[*kasiteltava*]**end while***polku.lisaaAlkuun*(*lahtosolmu*)**return** *polku***else****for all** $n \in$ *graafidata.naapurit*(*kasiteltava*) **do****if** ($n \notin$ *vieraillut*) \wedge ($n \notin$ *kasiteltavat*) **then***vanhemmat*[n] \leftarrow *kasiteltava**kasiteltavat.lisaaLoppuun*(n)**end if****end for****end if**

Ohjelmalistaus 2 Esimerkki DFS-algoritmista

```

procedure DFSESIMERKKI(graafidata, lahtosolmu, maalisolmu, gVieraillut)
    vieraillut  $\leftarrow$  gVieraillut
    vieraillut.lisaaLoppuun(lahtosolmu)
    for all  $n \in$  graafidata.naapurit(lahtosolmu) do
        if  $n \notin$  vieraillut then
            if  $n \neq$  maalisolmu then
                polku = DFSEsimerkki(graafidata,  $n$ , maalisolmu, vieraillut)
            else
                return [ $n$ ]
            end if
        end if
    end for
    if polku  $\neq$  NULL then
        polku.lisaaAlkuun(lahtosolmu)
    end if
    return polku
end procedure

```

Ohjelmalistaus 3 Esimerkki Dijkstran algoritmista (osa 1)

procedure DIJKSTRAESIMERKKI(*graafidata*, *lahtisolmu*, *maalisolmu*)

unvisited : {} $\leftarrow (\forall V \in G : true)$

distances : {}

for all $V \in G$ **do**

$(V = S) \Rightarrow (distances[V.index] \leftarrow 0)$

$(V \neq S) \Rightarrow (distances[V.index] \leftarrow \infty)$

end for

parents : {}

for all $V \in G$ **do**

$(V = S) \Rightarrow (distances[V.index] \leftarrow -1)$

$(V \neq S) \Rightarrow (distances[V.index] \leftarrow -2)$

end for

path : {}

current $\leftarrow S$

i $\leftarrow 0$

while $i \leq G.size0$ **do**

for all $N \in current.neighbors$ **do**

if *unvisited*[*N.index*] = *true* **then**

nWeight $\leftarrow distances[current.index] + weight(current, N)$

if *distances*[*N.index*] > *nWeight* **then**

distances[*N.index*] $\leftarrow nWeight$

parents[*N.index*] $\leftarrow current$

Ohjelmalistaus 4 Esimerkki Dijkstran algoritmista (osa 2)

```
    end if

    end if

end for

 $unvisited[current.index] \leftarrow false$ 

 $minDist \leftarrow inf$ 

 $minDistInd \leftarrow -1$ 

for all  $U \in unvisited$  do

    if  $unvisited[U.index] = true$  then

        if  $distances[U.index] < minDist$  then

             $minDist \leftarrow distances[U.index]$ 

             $minDistInd \leftarrow U.index$ 

        end if

    end if

end for

 $current \leftarrow minDistInd$ 

 $(current = d) \Rightarrow ( \text{break} )$ 

 $i = i + 1$ 

end while

 $previous \leftarrow parents[current]$ 

while  $previous \neq -1$  do

     $path.beginning \leftarrow current$ 

     $current \leftarrow previous$ 

     $previous \leftarrow parents[current]$ 

end while

 $path.beginning \leftarrow S$ 

return  $path$ 

end procedure
```

Ohjelmalistaus 5 Esimerkki A*-algoritmista (osa 1)

procedure A*ESIMERKKI(*graafidata*, *lahtosolmu*, *maalisolmu*)

avoinL \leftarrow *tyhjaLista*

suljettuL \leftarrow *tyhjaLista*

gHintaLog \leftarrow *tyhjaSanakirja*

avoinL.lisaaLoppuun(*nykyinen* = *lahtosolmu*; *vanhempi* = *NULL*)

gHintaLog.lisaa(*AVAIN* = *lahtosolmu*; *ARVO* = 0)

while *avoinL* \neq *tyhjaLista* **do**

minHinta $\leftarrow \infty$

for all *solmu* \in *avoinL* **do**

if *gHintaLog*[*solmu.nykyinen*] + *heuristinenArvo*[*solmu.nykyinen*]

 < *minHinta* **then**

minHinta \leftarrow

gHintaLog[*solmu.nykyinen*] + *heuristinenArvo*[*solmu.nykyinen*]

nykyinen \leftarrow *solmu*

end if

end for

avoinL.poista(*avoinL.indeksi*(*nykyinen*))

suljettuL.lisaaLoppuun(*nykyinen*)

for *n* \in *graafidata.naapurit*(*nykyinen.nykyinen*) **do**

if *n* = *maalisolmu* **then**

nykyinen \leftarrow (*nykyinen* = *n*; *vanhempi* = *nykyinen.nykyinen*)

polku \leftarrow *tyhjaLista*

while *nykyinen.vanhempi* \neq *NULL* **do**

Ohjelmalistaus 6 Esimerkki A*-algoritmista (osa 2)

```
    polku.lisaaAlkuun(nykyinen.nykyinen)

    nykyinen  $\leftarrow$  nykyinen.vanhempi

end while

polku.lisaaAlkuun(lahtoarvo)

return polku

end if

if  $n \in \text{suljettuL.map}(jasen \rightarrow jasen.nykyinen)$  then

    continue

else

    if  $n \notin \text{avoinL.map}(jasen \rightarrow jasen.nykyinen)$  then

        avoinL.lisaaLoppuun

        ( $nykyinen = n$ ;  $vanhempi = nykyinen.nykyinen$ )

         $gHintaLog.lisaa(AVAIN = n; ARVO =$ 

         $gHintaLog(nykyinen.nykyinen) +$ 

         $graa fidata.hinta(nykyinen.nykyinen, n))$ 

    else

        if  $gHintaLog[n] > gHintaLog(nykyinen.nykyinen) +$ 

         $graa fidata.hinta(nykyinen.nykyinen, n)$  then

             $gHintaLog[n] \leftarrow gHintaLog(nykyinen.nykyinen) +$ 

             $graa fidata.hinta(nykyinen.nykyinen, n)$ 

        end if

    end if

end if

end for

end while

return NULL

end procedure
```

Ohjelmalistaus 7 Esimerkki hierarkisesta polunetsintäalgoritmista

procedure HIERARKINENESIMERKKI(*graafidata*, *lahtsolmu*, *maalisolmu*)

YlatasonGraafi = *Alueenjakoalgoritmi.Jaa(graafigdata)*

lahtoalue \leftarrow *tyhjaGraafi*

maalialue \leftarrow *tyhjaGraafi*

varsinainenPolku \leftarrow *tyhjaLista*

for all *alue* \in *YlatasonGraafi* **do**

if *lahtsolmu* \in *alue* **then**

lahtoalue \leftarrow *alue*

end if

if *maalisolmu* \in *alue* **then**

maalialue \leftarrow *alue*

end if

end for

YlatasonPolku = *PolunetsintaAlgoritmi*(*YlatasonGraafi*, *lahtoalue*, *maalialue*)

for all *alue* \in *YlatasonPolku* **do**

varsinainenPolku.lisaaLoppuun(

PolunetsintaAlgoritmi(*alue*, *Alueenjakoalgoritmi.KriittisetPisteet*(*alue*)))

end for

Return *varsinainenPolku*

end procedure
