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, 17 s., 1 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, 17 p., 1 app. p.
Laboratory Name
2 2023

English abstract here

Keywords: here, a, list, of, keywords

Sisällys

Kuvat

Termistö

BFS Breadth Fist Search, leveyssuuntainen läpikäynti, leveyshaku

DFS Depth Fist Search, syvyyssuuntainen läpikäynti, syvyyshaku

NPC Non-Playable Character, ei-pelattava hahmo

RTS Real-Time Strategy, reaaliaikainen strategiapeli

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 kandityön 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 julkaisuajan mukaan niin, että suurin osa hakutuloksista on julkaistu vuona 2018 tai sen jälkeen. Myös aihepiirirajausta on käytetty. Hakusanoissa on käytetty osuvempien tulosten löytämiseksi Boolen 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 taustaaiheet, jotta seuraavien lukujen ymmärtäminen helpottuisi. Luvussa 3 käydään läpi muutaman tunnetun polunetsinnän toiminta ja täten pyritään vastaamaan tutkimuskysymykseen **1.** Luvussa 4 käydään läpi joitakin polunetsintäaldoritmien 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 osaonglemaan: 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. Moniagentillisisssa ongelmissa alueella liikkuvia agentteja 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 toisin mainita. Nämä algoritmit voidaan jakaa niiden toimintaperjaatteen 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 niinkutsuttua heuristista funktiota. Näin ollen jokaiselle graafissa olevalle solmukohdalle $\mathbf{n}$ saadaan laskettua sen kautta kulkevan polun hinta $\mathbf{F(n)}$ käyttämällä hintafunktiota $\mathbf{F(n) = G(n) + H(n)}$, jossa $\mathbf{G(n)}$ on solmulle $\mathbf{n}$ asti kuljettu matka lähtösolmusta ja $\mathbf{H(n)}$ 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 implementoida 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 automatisaatiossa 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 laskentatehokkaita. 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 rakenstaa 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 polunetsintäalgoritmejä 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.

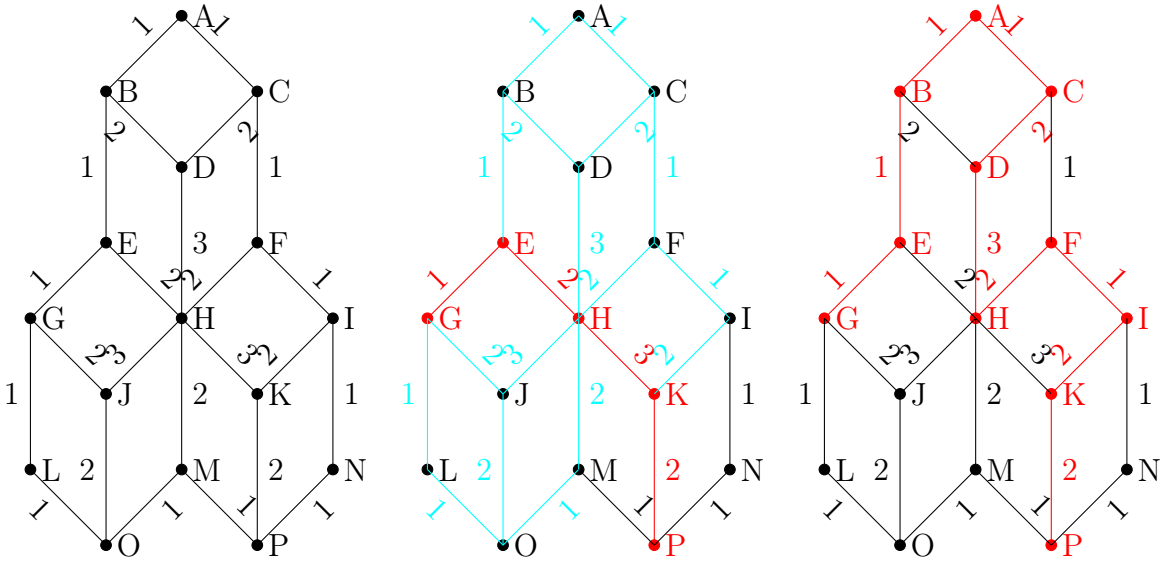
Polunetsintää on tutkittu myös paljon robotiikan näkökulmasta. Robotiikassa polunetsintää joudutaan soveltamaan muun muassa kun kehitetään itseajavia ajoneuvoja ja tehtaassa liikkuvia teollisuusrobotteja.[3] Esimerkiksi Poggenhans ja Janosovits [8] pohtivat artikkelissaan muunmuassa sitä, miten itseajaville 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 epäinformoitu hakualgoritmi, joka perustuu sokeaan hakuun.[4] Siinä graafin solmut ryhmitellään eri tasoihin sen mukaan monenko kaaren kautta pitää kulkea lähtösolmusta, jotta niihin päästään. Lähtösolmu on siis tasolla 0, siihen yhdistyneet solmut tasolla 1, tason 1 solmuihin yhdistyneet solmut tasolla 2 ja niin edelleen. Leveyshaussa 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, ohjelmalistauksen 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.



(a) Referenssigraafi (b) BFS referenssigraafissa (c) DFS referenssigraafissa

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 syaaninväristen kaarten suuresta määrästä.

3.2 Syvyyssuuntainen läpikäynti (DFS)

Syvyyssuuntainen läpikäynti, eli syvyyshaku (Depth Fitst 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-

Ohjelmalistaus 1 Esimerkki BFS-algoritmista

```
procedure BFSEsimerkki(graafidata, lahtosolmu, maalisolmu)
    vieraillut ← tyhjaLista
    kasiteltavat ← tyhjaLista
    kasiteltavat.lisaaLoppuun(nykyinen = lahtosolmu; vanhempi = NULL)
    while kasiteltavat ≠ tyhjaLista do
        kasiteltava ← kasiteltavat.poistaEnsimmainen()
        vieraillut.lisaaLoppuun(kasiteltava.nykyinen)
        if kasiteltava.nykyinen = maalisolmu then
            polku ← tyhjaLista
            while kasiteltava.vanhempi ≠ NULL do
                polku.lisaaAlkuun(kasiteltava.nykyinen)
                kasiteltava ← kasiteltava.vanhempi
            end while
            polku.lisaaAlkuun(lahtosolmu)
            return polku
        else
            for all n ∈ graafidata.naapurit(kasiteltava.nykyinen) do
                ((n ∉ vieraillut)?kasiteltavat.lisaaLoppuun
                (nykyinen = n; vanhempi = kasiteltava.nykyinen) :
                continue
            end for
        end if
    end while
end procedure
```

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]

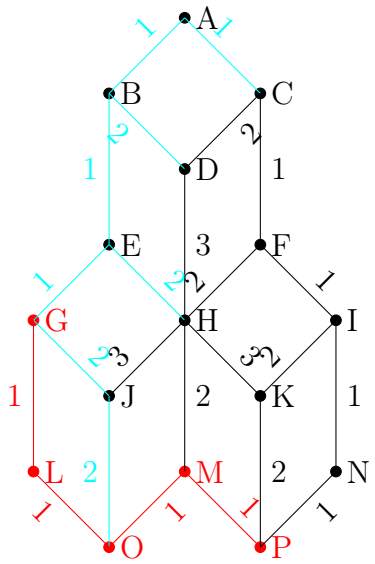
3.3 Dijkstran algoritmi

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 rakennetussa graafissa näkyisikään. Voi tulla tilanteita, joissa oikeasti lyhyempi reiti 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 kappaleessa 3.1. Algoritmin ajo johtaa kuvan 3.2

Ohjelmalistaus 2 Esimerkki DFS-algoritmista

```
procedure DFSEsimerkki(graafidata, lahtosolmu, maalisolmu, gVieraillut)
    vieraillut ← gVieraillut
    vieraillut.lisaaLoppuun(lahtosolmu)
    for all n ∈ graafidata.naapurit(lahtosolmu) do
        if n ∉ vieraillut then
            if n ≠ maalisolmu then
                polku = DFSEsimerkki(graafidata, n, maalisolmu, vieraillut)
            else
                return [n]
            end if
        end if
    end for
    if polku ≠ NULL then
        polku.lisaaAlkuun(lahtosolmu)
    end if
    return polku
end procedure
```



Kuva 3.2: Dijkstran algoritmi 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 ongelmassa 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 (Dijksta 12 verrattuna DFS 10). Eri algoritmeja vertaillaan enemmän luvussa 5.

3.4 A*-algoritmi

3.5 Hierarkinen polunetsintä

Ohjelmalistaus 3 Esimerkki Dijkstran algoritmista (osa 1)

```
procedure DijkstraEsimerkki(graafidata, lahtosolmu, maalisolmu)
    kasiteltava ← (nykyinen = lahtosolmu; vanhempi = NULL)
    vieraillut ← tyhjaLista
    hintaLog ← tyhjaSanakirja
    hintaLog.lisaa(AVAIN = lahtosolmu; ARVO = 0)
    while vieraillut.pituus ≠ graafidata.pituus do
        vieraillut.lisaaLoppuun(kasiteltava)
        minHinta ← (solmulta = NULL; solmulle = NULL;
        hinta = graafidata.maxHinta + 1)
        for v ∈ vieraillut do
            for n ∈ graafidata.naapurit(v) do
                if (nykyinen = n; vanhempi = v) ∉ vieraillutAND
                hintaLog[v] + graafidata.hinta(v, n) < minHinta.hinta then
                    minHinta ← (solmulta = n; solmulle = v;
                    hinta = hintaLog[v] + graafidata.hinta(v, n))
                end if
            end for
        end for
        hintaLog.lisaa
        (AVAIN = minHinta.solmulle; ARVO = minHinta.hinta)
```

Ohjelmalistaus 4 Esimerkki Dijkstran algoritmista (osa 2)

```
        kasiteltava ←
        (nykyinen = minHinta.solmulle; vanhempi = minHinta.Solmulta)
        if kasiteltava.nykyinen = maalisolmu then
            polkuOK ← tosi
            break
        end if
    end while
    if polkuOK then
        polku ← tyhjaLista
        while kasiteltava.vanhempi ≠ NULL do
            polku.lisaaAlkuun(kasiteltava.nykyinen)
            kasiteltava ← kasiteltava.vanhempi
        end while
        polku.lisaaAlkuun(lahtosolmu)
        return polku
    else
        return NULL
    end if
end procedure
```

4 Algoritmien sovelluskohteita

4.1 Videopelit

4.2 Karttaohjelmat

4.3 Robotiikka

Tähän mahdollisesti muitakin sovelluskohteita

5 Eräiden algoritmien tehokkuuden tarkastelu esimerkkiongelmassa

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.

Liite A Liitedokumentti placeholder

Placeholder liitedokumenteille