Tampereen yliopisto

Pekka Oinas

POLUNETSINTÄ RUUDUKKOKARTOILLA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Huhtikuu 2020

TIIVISTELMÄ

Pekka Oinas: Polunetsintä ruudukkokartoilla
Kandidaatintyö
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2020

Tässä kirjallisuuskatsauksessa tutustutaan säännöllisillä ruudukkokartoilla tapahtuvaan polunetsintään. Ruudukkokartat ovat yleisiä monilla polunetsinnän sovellusalueilla, kuten robotiikassa ja videopeleissä, ja niiden sisältämien polkujen korkea symmetria muodostaa haasteellisen ongelman perinteisille polunetsinnän ratkaisuille, kuten A*-algoritmille, hidastaen sen toimintaa; tästä syystä on olemassa monia ruudukkokartoille optimoituja algoritmeja, jotka suoriutuvat A*:ä nopeammin.

Tutkielmassa perehdytään ensin polunetsinnän relevantteihin perusteisiin, Dijkstran algoritmiin ja sen jälkeläiseen, A*-algoritmiin, tarkastelemalla niiden pseudokooditoteutusta sekä suoritusaikaominaisuuksia. Tämän jälkeen esitellään kolme ruudukkokartoille erikoistunutta algoritmia: Hierarchical Path-Finding A*, Rectangular Symmetry Reduction ja Jump Point Search. Esiteltyjen algoritmien toiminta kuvaillaan ja niiden suorituskykyä vertaillaan suhteessa A*:een ja muihin algoritmeihin.

Tutkielman lopussa esitetään johtopäätös, jonka mukaan tarkastelluista algoritmeista Jump Point Search vaikuttaa suorituskykynsä ja muiden ominaisuuksiensa puolesta olevan paras ratkaisu ongelmaan. Se suoriutuu jopa monikymmenkertaisesti lähtökohtana käytettyä A*-algoritmia nopeammin ja on kaikissa tarkastelluissa tilanteissa yhtä nopea tai nopeampi kuin muut vertaillut algoritmit, eikä sillä ole mainittavia varjopuolia. Lisäksi ehdotetaan kahta mahdollista suuntaa jatkotutkimukselle.

Avainsanat: polunetsintä, A*, A*-optimisaatio, ruudukkokartta

ABSTRACT

Pekka Oinas: Pathfinding on Grid Maps
Bachelor Thesis
Tampere University
Degree Programme in Computer Sciences
April 2020

This literary review examines pathfinding on regular grid maps. Regular grid maps are common in many applications of pathfinding, such as robotics and video games, and they pose a challenging problem to traditional solutions such as the A* algorithm, slowing down its execution. For this reason there are many algorithms optimized for grid maps to yield better performance than A*.

In this paper we first go over the relevant basics of pathfinding — Dijkstra's algorithm and its extension, A* — by examining their pseudocode representations and running time properties. We then introduce three algorithms optimized for pathfinding on grid maps: Hierarchical Path-Finding A*, Rectangular Symmetry Reduction and Jump Point Search. We describe the operation of these algorithms and compare their performance to A* and other algorithms.

We conclude that out of the presented algorithms, Jump Point Search appears to be the best choice considering its performance and other properties. It performs an order of magnitude faster than the A* algorithm used as the baseline, it is faster or at least as fast as the other examined algorithms, and it has no appreciable downsides. We also suggest two possible directions for further research.

Keywords: pathfinding, A*, A* optimization, grid map

SISÄLLYSLUETTELO

1 JOHDANTO

Polunetsintä (*pathfinding*) eli lyhimmän polun ongelma (*shortest path problem*) on yleinen ja perinteinen ongelma tietojenkäsittelytieteissä, ja sen perusteet ovat varmasti tutut lähes jokaiselle alaan perehtyneelle. Intuitiivisesti polunetsintää voi ajatella lyhimmän kahden pisteen välisen reitin löytämisenä. Nämä pisteet voivat olla fyysisiä sijainteja, kuten esimerkiksi sokkelon sisään- ja uloskäynnit tai kaksi osoitetta tieverkossa, mutta ne voivat esittää myös abstraktimpia käsitteitä, kuten shakkilaudan tilaa. Saattaapa pisteitä olla useampikin, kuin kaksi; reittiä voidaan etsiä vaikkapa useista päätepisteistä lähimpään.

Polunetsinnän historia on pitkä ja sen alkupistettä on hankala määritellä. Tunnetuin nimenomaan lyhimmän polun ongelman ratkaiseva algoritmi lienee kuitenkin vuonna 1959 esitetty Dijkstran algoritmi [5]. Algoritmin maine selittyy kenties sen yksinkertaisuudella. Dijkstra kertoi vuonna 2001 kehittäneensä algoritmin esimerkkinä tietokoneella ratkaistavasta ongelmasta, jonka kuka tahansa voi helposti ymmärtää; mikä on lyhin reitti Rotterdamista Groningeniin [6]?

Yksinkertaisuudestaan huolimatta Dijkstran algoritmi levisi laajaan suosioon, ja sen käyttökohteet paljastuivat paljon Rotterdamin ja Groningenin välisen reitin etsimistä moninaisemmiksi. Sitä ja siihen perustuvia algoritmeja sovelletaan muun muassa keinoälyyn, tietoverkkojen reitittämiseen, liikenneverkoissa navigointiin, robotiikkaan, videopeleihin ja moneen muuhun tarkoitukseen.

Tässä kirjallisuuskatsauksessa perehdymme ensin lyhyesti polunetsinnän perusteisiin Dijkstran algoritmin ja siitä kehitetyn A*:n muodossa, ja tutustumme sitten pieneen mutta tietyillä aloilla tärkeään polunetsinnän osa-alueeseen: säännöllisillä ruudukkokartoilla tapahtuvaan polunetsintään. Ruudukkokartat ovat yleisesti käytössä esimerkiksi robotiikan ja videopelien aloilla, ja perinteiset ratkaisut kuten A* suoriutuvat niillä verrattain kehnosti.

Tarkastelemme kolmen ruudukkokarttoihin erikoistuvan algoritmin — HPA*:n [2], RSR:n [8] ja JPS:n [9] — toimintaa ja suorituskykyä. Lopuksi tarjoamme lyhyen johtopäätöksen algoritmeista ja esitämme mahdollisia suuntia jatkotutkimukselle.

2 POLUNETSINNÄN PERUSTEET

2.1 Dijkstran algoritmi

Edsger W. Dijkstran vuonna 1959 esittelemä Dijkstran algoritmi [5] lienee naiiveja ja epäoptimaalisia lähestymistapoja lukuunottamatta polunetsintäalgoritmeista yksinkertaisin. Sitä voidaan pitää eräänlaisena polunetsintäalgoritmien esivanhempana, sillä lähes jokaisen modernin polunetsintäalgoritmin voidaan katsoa pohjautuvan tavalla tai toisella siihen.

Dijkstran algoritmi löytää lyhimmän polun kahden positiivisilla arvoilla painotetun graafin solmun, P:n ja Q:n, välillä etsimällä lyhimmän polun solmusta P graafin jokaiseen solmuun kasvavassa etäisyysjärjestyksessä, kunnes löydetään lyhin polku solmuun Q.

Huomattakoon, että alla esitetyssä muodossaan Dijkstran algoritmi palauttaa vain yhden polun; on mahdollista, että graafissa on vaihtoehtoisia yhtä lyhyitä polkuja, joita algoritmi ei palauta. Tätä kutsutaan symmetriaksi.

Algoritmin toiminta

Olkoon P painotetun graafin G solmu, josta polunetsintä aloitetaan, ja Q solmu, johon polkua etsitään. Määritellään kaksi listaa, joissa avaimina toimivat G:n solmut s:

- Etäisyysarvio d, jossa ylläpidetään kunkin s:n ja P:n välisen lyhimmän polun pituuden ylärajaa. Algoritmin edetessä d:n arvo kullekin solmulle tarkentuu, ja jokaiselle vieraillulle solmulle sen arvo on tarkasti P:n ja s:n välisen lyhimmän polun pituus.
- Vanhempi π, jossa ylläpidetään edellistä solmua, jonka kautta lyhin polku solmujen P ja s välillä kulkee. Vanhemman arvo päivitetään aina, kun löydetään aiempaa lyhempi polku solmujen P ja s välillä.

Lisäksi määritellään kaksi solmujoukkoa:

- Vierailtujen solmujen joukko V, joka sisältää solmut, joissa algoritmi on jo vieraillut; eli toisin sanottuna solmut, joille on löydetty lyhin polku solmuun P.
- Avointen solmujen joukko A, jonka jäsenet ovat kandidaatteja seuraavaksi vierailtavaksi solmuksi. Algoritmin aikana A:sta valitaan toistuvasti solmu, jolla on alhaisin etäisyysarvio d; tästä syystä A toteutetaan minimiprioriteettijonona, joka on järjestetty solmujen etäisyysarvioiden mukaan.

Algoritmin alussa aloitussolmu P lisätään avointen solmujen joukkoon A, sen vanhemmaksi $\pi[P]$ asetetaan tyhjä arvo nil, ja sen etäisyysarvioksi $d[P]$ asetetaan 0. Tämän jälkeen algoritmi toimii seuraavasti:

1. Valitaan avointen solmujen joukosta A solmu, jolla on pienin etäisyysarvio d, poistetaan se A:sta ja asetetaan se nykyiseksi solmuksi s. Mikäli $s = Q$, lyhin polku P:stä Q:n on löydetty ja algoritmi päättyy; lyhin polku muodostuu seuraamalla polkujen vanhempia π aloittaen Q:sta, kunnes saavutetaan solmu P. Toisaalta mikäli A on tyhjä joukko, ei P:n ja Q:n välillä ole polkua ja algoritmi päättyy; näin voi tapahtua, mikäli graafi ei ole kytketty.
2. Tarkastellaan nykyisen solmun s naapureita, jotka eivät ole joukon V jäseniä. Lasketaan jokaisen naapurin n etäisyys P:hen s:n kautta kuljettuna lisäämällä yhteen s:n ja P:n välisen polun pituus $d[s]$ sekä s:n ja n:n välisen kaaren pituus. Jos tulos on pienempi, kuin $d[n]$ (tai jos $d[n]$ on tyhjä arvo, eli solmun etäisyyttä ei ole aiemmin arvioitu), on löydetty aiempaa lyhempi polku solmuun n, ja asetetaan $d[n] \leftarrow tulos$ ja $\pi[n] \leftarrow s$. Lisätään n avointen solmujen joukkoon A prioriteetilla $d[n]$.
3. Kun solmun s jokainen vierailematon naapuri on käsitelty, lisätään s vierailtujen solmujen joukkoon V. Vierailtuihin solmuihin on löydetty jo lyhin polku, joten niitä ei tarvitse tarkastella enää uudestaan.

Kirjallisuudessa käytetään usein termiä **laajentaa** (*expand*), kun viitataan prosessiin, jossa solmu poistetaan A:sta ja sen vierailemattomat naapurit lisätään A:han. Solmun lisäämistä A:han kutsutaan usein **generoinniksi** (*generate*) [19, kpl. 3.3]. Tässä tutkielmassa käytämme näihin operaatioihin viitatessamme näitä termejä.

Dijkstran algoritmin pseudokoodi

Algoritmin 2.1 pseudokoodi on mukailtu *Introduction to Algorithms*:sta [3] ja Amit Patelilta [15] hakemaan polku kahden solmun välille. Algoritmi kutsuu prioriteettijonon metodeja `extract-min`, joka poistaa ja palauttaa prioriteettijonosta jäsenen, jolla on alhaisin prioriteetti, ja `put`, joka lisää prioriteettijonoon uuden jäsenen annetulla prioriteetillä, tai uudelleenpriorisoi jo olemassa olevan jäsenen. Lisäksi käytetään funktiota `length(p, q)`, joka palauttaa kahden solmun välisen kaaren pituuden.

```
1  function Dijkstra(P, Q):
2      V = empty set //Vierailtujen solmujen joukko
3      A = min-priority queue //Avointen solmujen prioriteettijono
4      d = empty list //Etäisyysarviot
5      p = empty list //Vanhemmat
6      A.put(P, 0)
7      p[P] = null
8      d[P] = 0
9
10     while A is not empty:
```

```
11          s = A.extract-min()
12          if s == Q:
13              break
14          V.add(s)
15          for each vertex n adjacent to s:
16              new_length = d[s] + length(s, n)
17              if (n not in d) or (d[n] > new_length):
18                  d[n] = new_length
19                  p[n] = s
20                  A.put(n, new_length)
```

Algoritmi 2.1. *Dijkstran algoritmin pseudokoodi.*

Algoritmin päätyttyä onnistuneesti voidaan lyhin polku hakea iteroimalla takaperin kohdesolmusta aloitussolmua kohti:

```
1  function BuildPath(Q):
2      J = empty list
3      s = Q
4      if p[s] != null or s == P:
5          while s != null:
6              J.insert-at-beginning(s)
7              s = p[s]
8      return J
```

Algoritmi 2.2. *Lyhimman polun hakeminen Dijkstran algoritmin suoriuduttua.*

Dijkstran algoritmin suoritusajasta

Dijkstran algoritmin laskennallisesti monimutkaisin osuus on algoritmin käyttämän prioriteettijonon ylläpitäminen. Tämän vuoksi algoritmin aikakompleksisuus riippuu prioriteettijonon toteutustavasta [3, s. 598–599], ja useimmat tavat nopeuttaa algoritmin toimintaa perustuvat aikatehokkaampaan prioriteettijonon toteutukseen, sekä monien algoritmin varianttien kohdalla prioriteettijonoon kohdistuvien operaatioiden määrän minimoimiseen.

Mikäli prioriteettijono on toteutettu yksinkertaisena taulukkona, jossa jokainen solu vastaa yhtä graafin solmua, algoritmin huonoimman tilanteen suoritusajaksi tulee $O(V^2)$, jossa V on graafin solmujen lukumäärä.

Harvoille graafeille (*sparse graph*), eli graafeille, joissa on merkittävästi vähemmän kuin $|V|^2$ kaarta, prioriteettijono on käytännöllistä toteuttaa binäärikekona. Tällöin algoritmin huonoimman tilanteen suoritusajaksi saadaan $O((E + V)\ log\ V)$, jossa E on graafin kaarten lukumäärä.

Suoritusaika pienenee entisestään mikäli prioriteettijono toteutetaan Fibonacci-kekona. Tällä toteutuksella huonoimman tilanteen suoritusajaksi saadaan $O(V\ log\ V + E)$.

2.2 Heuristiset haut

Dijkstran algoritmin kaltaisten formaalejen ratkaisujen rinnalla on olemassa erilaisia heuristisia ratkaisuja, jotka pyrkivät soveltamaan ongelmakohtaista tietoa kahden solmun välisen polun pituuden arviointiin. Polunetsinnän tapauksessa tämä toteutetaan useimmiten **heuristisen arviointifunktion** (*heuristic evaluation function*) muodossa [1, kpl. 22.4].

Heuristisen arviointifunktio voi hyödyntää oikeastaan mitä tahansa ongelma-alueelle tyypillistä tietoa, mutta tärkeää on, että se arvioi polun todellista pituutta, ja että se on tehokkaasti laskettavissa. Monissa käytännön sovelluksissa, joissa graafin solmut vastaavat pisteitä avaruudessa, euklidinen etäisyys on yleinen heuristiikka. 4-naapurista neliöruudukkoa (eli ruudukkoa, jossa jokainen ruutu on yhteydessä ortogonaalisiin naapureihinsa) vastaavassa graafissa niinkutsuttu Manhattan-etäisyys, eli kahden ruudun välinen etäisyys vain pysty- ja vaakasuorassa liikkuen, olisi toimiva heuristiikka.

Arviointifunktiota, jonka tulos vastaa lyhintä mahdollista etäisyyttä kahden solmun välillä (eli toisin sanottuna funktiota, joka ei koskaan yliarvioi polun pituutta), kutsutaan **luvalliseksi** (*admissible*). Esimerkiksi tieverkossa nopein mahdollinen reitti kahden kaupungin välillä olisi kulkea tieverkon suurimman nopeusrajoituksen mukaisesti linnuntietä, sillä suora viiva on lyhin etäisyys kahden pisteen välillä; euklidinen etäisyys jaettuna suurimmalla nopeusrajoituksella olisi siis tähän ongelmaan luvallinen heuristiikka (mutta tieverkon rakenteesta riippuen se ei välttämättä ole tarkin mahdollinen heuristiikka).

Arviointifunktion luvallisuus on joillekin käyttökohteille tärkeä vaatimus, mutta toisaalta myös ei-luvalliset arviointifunktiot ovat tietyissä tilanteissa hyödyllisiä. Esimerkiksi A*-algoritmi on taattu löytämään lyhin polku vain luvallisella arviointifunktiolla, mutta monet sen variantit hyödyntävät ei-luvallista arviointifunktiota nopeuttaakseen algoritmin suoritusaikaa löydettyjen polkujen optimaalisuuden hinnalla [12, s. 107].

2.2.1 Täysin heuristinen haku

Heuristisista hauista yksinkertaisin on **täysin heuristinen haku** (*pure heuristic search*) tai **ahne paras ensin -haku** (*greedy best-first search*) [1, kpl. 22.4.3]. Se toimii Dijkstran algoritmin kaltaisesti, mutta sen sijaan, että se arvioisi etäisyyttä lähtösolmuun P, se käyttää heuristista funktiota arvioidakseen kunkin solmun etäisyyttä päätesolmuun Q. Se on niin kutsuttu paras ensin -haku, sillä toisin kuin Dijkstran algoritmi, joka tarkastelee solmuja sokean tasa-arvoisesti, täysin heuristinen haku vierailee ensimmäisenä lupaavimmalta näyttävissä solmuissa.

Täysin heuristinen haku löytää polun alku- ja päätesolmun välillä yleensä merkittävästi Dijkstran algoritmia nopeammin, sillä se vierailee pienemmässä määrässä solmuja polkua etsittäessä. Dijkstran algoritmista poiketen sen löytämät polut eivät kuitenkaan ole taattuja olemaan optimaalisia lyhimpiä polkuja.

Koska graafin rakenteesta riippuen täysin heuristisen haun löytämät polut saattavat olla merkittävästi lyhintä polkua pidempiä, soveltuu se parhaiten ongelmiin, joissa optimaalista ratkaisua tärkeämpää on löytää jokin ratkaisu nopeasti.

Täysin heuristisen haun pseudokoodi

Algoritmin 2.3 pseudokoodi muistuttaa paljon Dijkstran algoritmia. Erona Dijkstran algoritmiin on etäisyysarviolistan d puute, sillä täysin heuristinen haku ei pidä kirjaa jo kuljetun polun pituudesta. Algoritmissa ei myöskään käytetä `length(p, q)`-funktiota; sen sijaan algoritmi käyttää heuristista funktiota `heur(p, q)`, joka palauttaa heuristisen arvion solmujen p ja q välisestä etäisyydestä.

```
1  function FHSearch(P, Q):
2      V = empty set //Vierailtujen solmujen joukko
3      A = min-priority queue //Avointen solmujen prioriteettijono
4      p = empty list //Vanhemmat
5      A.put(P, 0)
6      p[P] = null
7
8      while A is not empty:
9          s = A.extract-min()
10         if s == Q:
11             break
12         V.add(s)
13         for each vertex n adjacent to s:
14             if (n not in p):
15                 priority = heur(n, Q)
16                 d[n] = new_length
17                 p[n] = s
18                 A.put(n, new_length)
```

Algoritmi 2.3. *Täysin heuristisen haun pseudokoodi. [15]*

2.2.2 A*

A* (lausutaan *A tähti*, engl. *A star*) on Peter Hartin, Nils Nilssonin ja Bertram Raphaelin vuonna 1968 esittelemä polunetsintäalgoritmi [12], joka on laajasti käytetty täydellisyytensä, optimaalisuutensa ja aikatehokkuutensa vuoksi [19, kpl. 3.5.2]. Näiden ominaisuuksien ja laajan soveltuvuutensa vuoksi A* muodostaa perustan valtaosalle modernista polunetsinnästä, etenkin tarkastelemallamme alalla.

A* on paras ensin -haku, joka arvioi solmuja valitessaan sekä solmujen etäisyyttä aloitussolmusta että heuristisen arviointifunktion tulosta. Siinä missä Dijkstran algoritmi arvioi vain etäisyyttä aloitussolmusta $g(n)$ ja täysin heuristinen haku arvioi vain heuristisen funktion arvoa $h(n)$, A* arvioi näiden summaa $g(n) + h(n)$. Sitä voidaan täten ajatella Dijkstran algoritmin ja täysin heuristisen haun yhdistelmänä, ja se onkin toteutukseltaan identtinen Dijkstran algoritmin kanssa muutoin, kuin solmujen arvioinnin osalta.

A*:n pseudokoodi

```
1  function AStar(P, Q):
2      V = empty set //Vierailtujen solmujen joukko
3      A = min-priority queue //Avointen solmujen prioriteettijono
4      d = empty list //Etäisyysarviot
5      p = empty list //Vanhemmat
6      A.put(P, 0)
7      p[P] = null
8      d[P] = 0
9
10     while A is not empty:
11         s = A.extract-min()
12         if s == Q:
13             break
14         V.add(s)
15         for each vertex n adjacent to s:
16             new_length = d[s] + length(s, n)
17             if (n not in d) or (d[n] > new_length):
18                 d[n] = new_length
19                 priority = new_length + heur(n, Q)
20                 p[n] = s
21                 A.put(n, priority)
```

Algoritmi 2.4. *A*:n pseudokoodi. Huomaa ainoat erot Dijkstran algoritmiin riveillä 19 ja 21. [15]*

A*:n ominaisuuksista

A*:n ominaisuuksia on tutkittu paljon. Alkuperäisessä artikkelissaan Hart ja muut todistivat A*:n **täydellisyyden** (algoritmi löytää aina reitin, mikäli sellainen on olemassa), sekä tietyin olettamuksin sen **optimaalisuuden** (algoritmi vierailee pienimmässä joukossa solmuja, joissa annetulla informaatiolla on mahdollista vierailla) ja **luvallisuuden** (algoritmi löytää aina lyhimmän polun).

Hart ja muut todistivat, että A* on luvallinen, mikäli sen käyttämä heuristinen arviointifunktio on luvallinen. He todistivat myös A*:n optimaalisuuden, mikäli heuristinen arviointifunktio on sekä luvallinen että **monotoninen** (*consistent*). Funktion monotonisuudella he tarkoittavat, että funktio täyttää kolmioepäyhtälön; eli toisin sanottuna, että funktion arvo solmulle s on aina pienempi tai yhtä suuri kuin funktion arvo jollekin s:n naapurille n, plus s:n ja n:n välisen kaaren paino.

Myöhemmin Hart ja muut julkaisivat artikkeliinsa korjauksen, jossa he väittivät monotonisuusvaatimuksen olevan tarpeeton [13], mutta Dechter ja Pearl osoittivat myöhemmin tämän vääräksi ja todistivat, että monotonisuus on vaadittu optimaalisuuden takaamiseksi [4].

A*:n luvallisuudesta

Kuten yllä mainittiin, A* on luvallinen vain, mikäli sen käyttämä heuristinen arviointifunktio on luvallinen. Toisaalta Hart ja muut totesivat jo alkuperäisessä artikkelissaan, että ei-luvalliset heuristiikat saattavat tuottaa tilanteesta riippuen haluttavampia tuloksia, kuin tiukasti luvalliset heuristiikat [12, s. 107].

Monet A*:n aikaisimmista varianteista keskittyivätkin ei-luvallisten heuristiikkojen hyödyntämiseen. Esimerkiksi Pohl esitteli vuonna 1973 dynaamisesti painotettuna A*:nä (*Dynamic Weighting A**) tunnetun variantin [18]. Tässä variantissa heuristisen funktion arvoa painotetaan algoritmin alussa enemmän, ja algoritmin edetessä painotusta alennetaan. Pohl esitteli aiemmin myös variantin, jossa heuristiikkaa painotetaan staattisella kertoimella. Pearl analysoi painotettujen heuristiikkojen vaikutusta kirjassaan *Heuristics: Intelligent Search Strategies for Computer Problem Solving* [16], kuin myös Dechter ja Pearl (1985) [4].

A*:n suoritusajasta

A*:n huonoimman tilanteen suoritusaika vastaa Dijkstran algoritmia, ja kuin Dijkstran algoritminkin, niin myös A*:en suoritusajan pullonkaulaksi muodostuu algoritmin käyttämä prioriteettijono.

A*:n keskimääräinen aikakompleksisuus on monitahoinen ongelma, joka riippuu algoritmin käyttämästä heuristiikasta sekä graafin rakenteesta. A*:n aikakompleksisuutta on analysoinut tarkemmin mm. Pearl [16], mutta aiheen monimutkaisuuden vuoksi emme perehdy tässä tutkielmassa siihen tarkemmin.

Objektiivisten mittareiden määrittelyn hankaluuden vuoksi alalla on yleisesti tapana vertailla algoritmien suoritusaikoja keskenään kokeellisin metodein. Näissä kokeissa A*:ä käytetään usein lähtökohtana, johon muita algoritmeja vertaillaan. Tässäkin tutkielmassa tarkasteltavien erikoistuneiden algoritmien suoritusaikoja mitataan enimmäkseen A*:een verrattuna.

3 POLUNETSINTÄ RUUDUKKOKARTOILLA

Ruudukkokartat ovat yleisiä useilla polunetsinnän sovellusalueilla, kuten robotiikassa ja videopeleissä [9]. Niille tyypillisiä piirteitä ovat tiheys (solmuja on suuri määrä, ja neliöruudukkokartassa jokaisella niistä on 4 tai 8 naapuria), kaarten painojen korkea yhdenmukaisuus (ruudut ovat aina yhtä kaukana naapureistaan, mukaanlukematta ruudukkoja jotka sisältävät ei-tasapainoisia kaaria) ja näistä seuraava kartan sisältämien polkujen korkea symmetrisyys (kahden pisteen välillä on usein monta yhtä pitkää polkua).

Nämä piirteet muodostavat haasteellisen ongelman perinteisille ratkaisuille kuten A*:lle, pidentäen algoritmin keskimääräistä suoritusaikaa. A*:n tapauksessa suurimmiksi ongelmiksi muodostuvat ruudukkokarttojen tiheys ja niiden korkea symmetrisyys. Tiheys johtaa yksinkertaisesti siihen, että algoritmin on tutkittava suuri määrä solmuja, joka johtaa useisiin operaatioihin algoritmin käyttämässä prioriteettijonossa. Symmetrisyys taasen johtaa siihen, että algoritmi arvioi toistuvasti useaa yhtä hyvää polkua sen sijaan, että keskittyisi niistä vain yhteen [8].

Tästä syystä onkin kehitetty useita ruudukkokartoille optimoituja polunetsintäalgoritmeja. Tutustumme tässä kappaleessa kolmeen ruudukkokarttoihin erikoistuvaan algoritmiin, joista kukin edustaa eri lähestymistapaa.

Huomattakoon, että esitellyssä muodossaan nämä algoritmit toimivat vain ruudukkokartoilla, joissa kaarten painoarvot ovat yhdenmukaisia; tällaisia ruudukkoja kutsutaan tasapainoisiksi (*uniform-cost*). Vaikka tämä on yleinen ruudukkokarttojen ominaisuus, joissakin sovelluskohteissa saatetaan käyttää myös ruudukkoja, joissa jotkin kaaret ovat toisia painavampia; näin voi olla esimerkiksi mikäli osa ruuduista esittää vaikeakulkuisempaa maastoa. Näihin tapauksiin tässä esitellyt algoritmit eivät sellaisenaan sovellu.

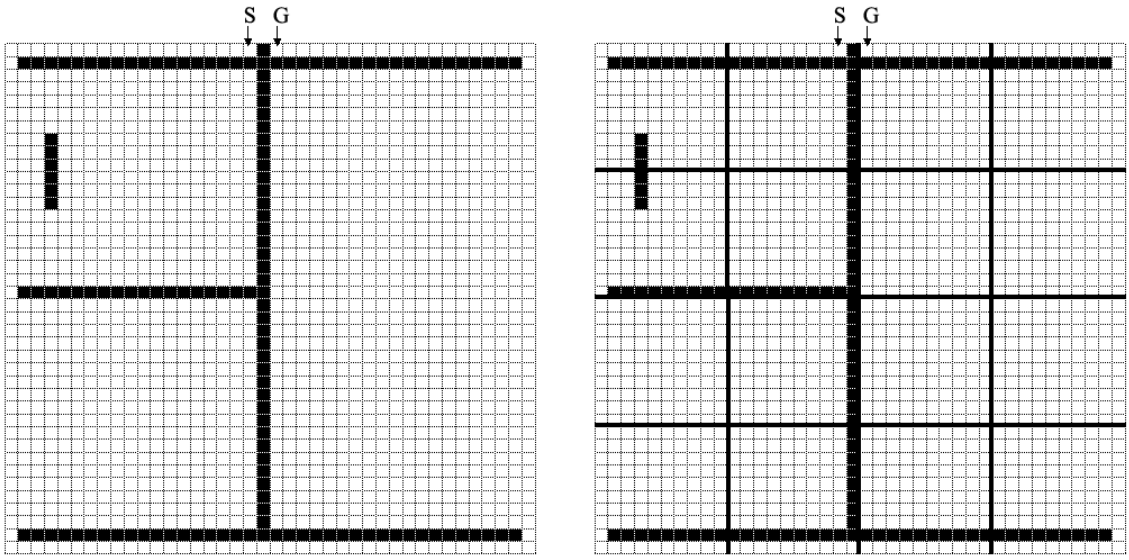
3.1 Hierarchical Path-Finding A*

Erilaisia tavalla tai toisella hierarkiaa hyödyntäviä polunetsintäratkaisuja on ollut olemassa jo pitkään. Esimerkiksi Holte ja muut esittelivät vuonna 1996 hierarkkiseksi A*:ksi (*Hierarchical A**) kutsumansa algoritmin, jota he käyttivät loogisten pulmapelien ratkaisuun [14].

Ruudukkokartoilla tapahtuvan polunetsinnän yhteydessä hierarkkisella A*:llä tarkoitetaan kuitenkin täysin erilaisia algoritmeja. Akatemiassa tällaisen algoritmin esittelivät yksityiskohtaisesti ensimmäiseksi Botea, Müller ja Schaeffer vuonna 2004 [2]. Heidän mukaansa hierarkkiset ratkaisut olivat jo aiemmin kaupallisella videopelialalla tunnettuja, mutta niistä tehty tutkimus oli vähäistä, sillä pelistudiot eivät yleensä julkaise ideoitaan tai lähdekoodiaan.

Botean et al. esittelemä **Hierarchical Path-Finding A*** (lyhemmin **HPA***) on ruudukkokartoilla toimiva A*:n variantti, joka hyödyntää etukäteen muodostettua abstraktia graafia. Algoritmi jakaa alkuperäisen ruudukon klustereihin — esimerkiksi 1000×1000 -kokoinen ruudukko voitaisiin jakaa 10×10 ruudun klustereihin — ja suorittaa näissä suurpiirteisen haun. Tämän jälkeen hakutulos tarkennetaan alkuperäisen ruudukon poluksi suorittamalla alkuperäisessä ruudukossa haku, joka on rajattu suurpiirteisen reitin klustereihin. Tällä tavoin algoritmin tarvitsee laajentaa merkittävästi vähemmän solmuja, joka nopeuttaa algoritmin suoritusaikaa perinteiseen A*:een verrattuna jopa monikymmenkertaisesti sillä hinnalla, ettei löydettyjen polkujen optimaalisuus ole taattua.

3.1.1 Ruudukon esikäsittely

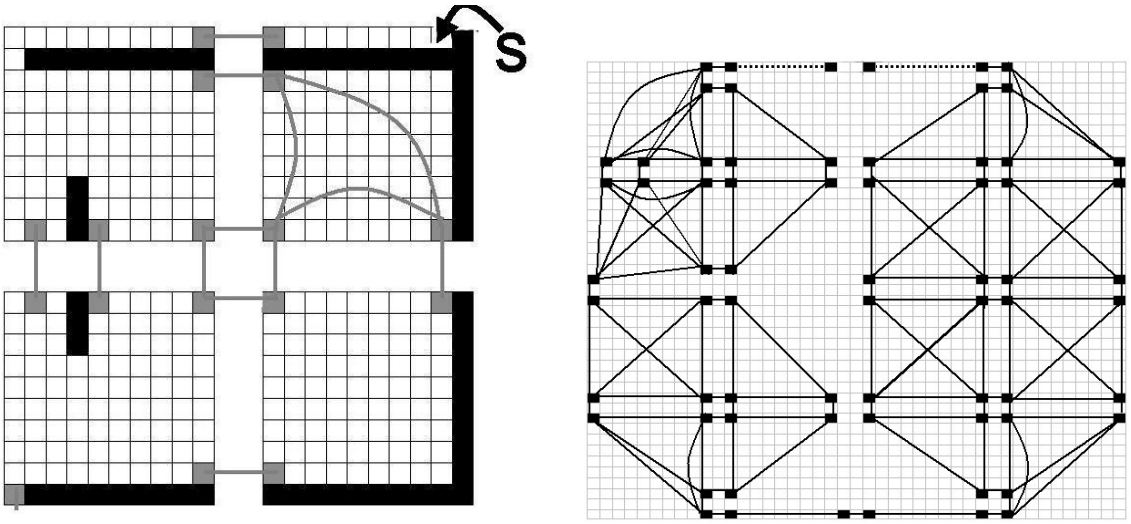


Kuva 3.1. (a) *40×40 -kokoinen esimerkkiruudukko. Esteet on merkitty mustalla. S ja G ovat alku- ja päätesolmut.* ***(b)*** *Paksunnetut viivat näyttävät 10×10 ruudun klustereiden rajat. [2]*

Algoritmin suorittamiseksi ruudukko on ensin esikäsiteltävä abstraktin hierarkkisen graafin muodostamiseksi. Ruudukko jaotellaan ensin suorakulmaisiin osiin, joita kutsutaan **klustereiksi** (*cluster*). Kuvan 3.1 esimerkissä 40×40 -kokoinen ruudukko on jaoteltu kuuteentoista 10×10 ruudun klusteriin.

Kultakin kahden klusterin väliseltä rajalta etsitään klusterit toisiinsa yhdistävien sisäänkäyntien (mahdollisesti tyhjä) joukko. Sisäänkäynti on maksimaalinen kuljettavien ruutujen jana kahden klusterin rajalla; Botea ja muut esittävät artikkelissaan sisäänkäynneille myös formaalin joukko-opillisen määritelmän [2, s. 8].

Kuva 3.2a esittää suurennettuna esimerkkiruudukon vasemman ylänurkan ja siitä muodostetun abstraktin graafin. Esimerkissä vasemman reunan kahta klusteria yhdistää kaksi sisäänkäyntiä, jotka ovat kolmen ja kuuden ruudun levyisiä. Artikkelissaan Botea et al. määrittelevät sisäänkäynnin leveydestä riippuen kullekkin sisäänkäynnille jo-



(a) *(b)*

Kuva 3.2. (a) *Esimerkkiruudukon vasemmasta ylänurkasta muodostettu abstrakti graafi. Abstraktin graafin solmut ja kaaret on merkitty harmaalla. Yksinkertaisuuden vuoksi klustereiden sisäiset kaaret on merkitty vain oikean yläreunan klusteriin.* ***(b)*** *Koko ruudukkoa kuvaava lopullinen abstrakti graafi. Graafiin on lisätty solmut S ja G, jotka on yhdistetty abstraktin graafin normaaleihin solmuihin katkoviivalla. [2]*

ko yhden tai kaksi **kulkureittiä** (*transition*). Esimerkissä alle kuuden ruudun levyisille sisäänkäynneille määritellään yksi kulkureitti sisäänkäynnin keskelle, ja tätä leveämmille sisäänkäynneille määritellään kaksi kulkureittiä sisäänkäynnin molemmille reunoille.

Näiden kulkureittien pohjalta muodostetaan abstrakti hierarkkinen graafi. Jokaista kulkureittiä kuvaamaan abstraktiin graafiin luodaan solmupari ja niitä yhdistävä kaari. Kaaria, jotka yhdistävät kaksi klusteria toisiinsa kutsutaan **ulkokaariksi** (*inter-edge*), ja niiden pituus on aina 1. Tällaisten ulkokaarien esittämä polku on alkuperäisessä ruudukossa triviaalisti löydettävissä.

Klusterien sisällä kukin klusterin alueella oleva solmu yhdistetään kaikkiin muihin klusterin sisältämiin solmuihin kaarella, jota kutsutaan **sisäkaareksi** (*intra-edge*). Sisäkaarille lasketaan painoarvo etsimällä klusterin alueelta lyhin polku kaaren yhdistämien solmujen välille käyttäen tavallista A*-hakua. Tämä polku voidaan joko unohtaa tai pitää, joskin polkujen säilyttäminen kasvattaa algoritmin muistivaatimuksia.

Kun abstrakti graafi on muodostettu, voidaan ruudukossa ajaa hierarkkista hakua. Mikäli ruudukko muuttuu, muuttuneiden klustereiden sisä- ja ulkokaaret on laskettava uudestaan.

Botea et al. tarkastelevat artikkelissaan myös monitasoisia hierarkioita, joissa abstraktin graafin pohjalta muodostetaan vielä korkeamman tason abstraktio [2, s. 13]. Tämä voi nopeuttaa polunetsintää erityisen suurissa ruudukoissa. Tässä tutkielmassa emme kuitenkaan käsittele monitasoisia abstraktioita tarkemmin.

3.1.2 Algoritmin toiminta

HPA*:ssä itse polunetsintä voidaan jaotella neljään askeleeseen, joista kaksi viimeistä voidaan tietyissä tilanteissa ohittaa tai suorittaa asteittain polkua kuljettaessa.

1. Yhdistetään alkupiste S ja päätepiste G abstraktiin graafiin. Tämä tehdään lisäämällä pisteet väliaikaisesti solmuiksi abstraktiin graafiin niihin klustereihin, joihin ne sisältyvät.
2. Suoritetaan tavallinen A*-haku, jolla etsitään lyhin polku S:n ja G:n välille abstraktissa graafissa. Tämä askel sisältää polun etsimisen S:stä S:n sisältämän klusterin rajalle, sekä vastaavasti G:n sisältämän klusterin rajalta G:hen.
3. Abstrakti polku voidaan **jalostaa** (*refine*) alkuperäisessä ruudukossa kuljettavaksi poluksi. Tätä ei ole pakko tehdä kerralla; reaaliaikaisissa käyttökohteissa reitti voidaan laskea myös klusteri kerrallaan agentin liikkuessa.
4. Polku voidaan **tasoittaa** (*path smoothing*) jalostetun polun ulkonäön ja pituuden parantamiseksi.

Abstraktin polun etsintä

Algoritmin ensimmäinen vaihe on etsiä polku abstraktissa graafissa S:stä G:hen. Jotta tämä olisi mahdollista, on S ja G sijoitettava osaksi abstraktia graafia. Tämä tehdään lisäämällä abstraktiin graafiin uusi solmu siihen klusteriin, johon piste kuuluu. Lisätystä pisteestä etsitään paikallisella haulla lyhin polku kuhunkin toiseen klusteriin kuuluvaan solmuun, ja mikäli polku löytyy, muodostetaan solmujen välille kaari, jonka painoarvoksi asetetaan polun pituus. Kuvassa 3.2b nämä kaaret on merkitty katkoviivoin.

Kun S ja G on lisätty abstraktiin graafiin, niiden välillä voidaan etsiä polku käyttäen mitä tahansa tällaisessa graafirakenteessa toimivaa polunetsintäalgoritmia, kuten A*:ä.

Polun jalostus

Jotta edellisessä osiossa muodostettu abstrakti polku voidaan kulkea, se tulee jalostaa alkuperäisen ruudukon poluksi. Tämä tehdään korvaamalla klustereiden sisäkaaria kulkevat reitit niitä vastaavilla alkuperäisen ruudukon reiteillä.

Mikäli esikäsittelyssä klusterin sisäiset reitit talletettiin, tässä askeleessa voidaan yksinkertaisesti käyttää näitä aiemmin laskettuja reittejä. Muussa tapauksessa algoritmi etsii kunkin klusterin sisäisen reitin uudelleen. Tämä voidaan tehdä online-tapaan, eli polku voidaan jalostaa klusteri kerrallaan agentin edetessä polulla — tällä tavoin polunetsinnän vaatima laskennallinen kompleksisuus voidaan levittää suuremmalle aikavälille, joka on hyödyllinen ominaisuus videopelien kaltaisissa reaaliaikaisissa käyttökohteissa.

Polun tasoitus

Koska abstrakti graafi määrittelee rajallisen määrän kulkureittejä klustereiden välisillä sisäänkäynneillä, ei jalostettujen reittien optimaalinen pituus ole taattu. Polkujen parantamiseksi voidaan suorittaa jälkiprosessointina tehtävä polun tasoitus, joka parantaa polkujen pituutta ja tekee niistä luonnollisemman näköisiä (esim. vähemmän mutkittelevia).

Botea et al. esittävät tasoittamiseen yksinkertaisen ratkaisun, mutta eivät tarkastele sitä kovinkaan yksityiskohtaisesti. Tasoitusalgoritmi aloittaa polun jommastakummasta päästä. Jokaisen pisteen kohdalla tarkistetaan, voidaanko pisteestä saavuttaa jokin polun myöhempi piste kulkemalla suorassa viivassa. Mikäli tällainen suora reitti löytyy, kahden pisteen välillä oleva reitti korvataan tällä suoralla viivalla. Lopputuloksena on polku, jossa on jäljellä vain solmut, jotka kiertävät esteiden kulmia [15].

3.1.3 Algoritmin suoritusaika ja muistivaatimukset

Botea et al. analysoivat artikkelissaan algoritminsa suoritusaikaa kokeellisesti vertailemalla sitä A*:een eri olosuhteissa. Taulukossa 3.1 vertaillaan kuvan 3.1 esimerkissä suoritettavan A*-haun suoritusaikaa hierarkkisen A*-haun suoritusaikaan algoritmin tutkimien solmujen määrän perusteella. Tässä esimerkissä, joka esittää pahinta mahdollista skenaariota, tavallinen A* tutki kymmenkertaisen määrän solmuja HPA*:een verrattuna.

Hakutekniikka	SG	Haku	**Yhteensä**	Jalostus
A*	0	1462	**1462**	0
HPA*	16	67	**83**	145

Taulukko 3.1. *Kuvan 3.1 ruudukossa suoritettujen hakujen vertailu. Taulukossa näytetään tutkittujen solmujen määrä.* SG *on solmuja* S *ja* G *abstraktiin graafiin lisätessä tutkittujen solmujen määrä.* Yhteensä *on kahden edellisen sarakkeen summa; tämä vastaa abstraktissa graafissa löydettyä polkua.* Jalostus *näyttää polun jalostuksessa tutkittujen solmujen määrän. [2]*

Botea et al. vertailivat algoritmeja myös *Baldur's Gate* -pelin kartoissa suoritetuilla hauilla. Heidän analyysinsä mukaan HPA* on yleensä merkittävästi A*:ä nopeampi, mutta hyvin lyhyillä reiteillä A* suoriutui nopeammin; lyhyillä reiteillä HPA*:n overhead oli korkeampi, kuin algoritmin säästämä aika. Heidän mukaansa A* suoriutui paremmin myös silloin, kun S:n ja G:n välinen reitti oli suora viiva, jolloin etäisyyttä heuristiikkana käyttävä A* tutkii optimaalisesti pienimmän mahdollisen määrän solmuja.

Botea et al. sanovat algoritmin muistivaatimusten olevan abstraktin graafin suhteellisesta yksinkertaisuudesta johtuen pieniä. *Baldur's Gate* -kartalle he arvioivat algoritmin muistioverheadin olevan 8.83%, mutta käytetyn muistin tarkka määrä riippuu graafin teknisestä toteutustavasta.

3.2 Rectangular Symmetry Reduction

Rectangular Symmetry Reduction eli **RSR** on Haraborin, Botean ja Kilbyn vuonna 2011 esittelemä esikäsittelyalgoritmi, joka vähentää ruudukkokarttojen symmetrisyyttä mutta säilyttää löydettyjen polkujen optimaalisuuden [8]. Se on Haraborin ja Botean aiemman, vain 4-naapurisilla ruudukoilla toimivan algoritmin [11] generalisaatio, joka toimii myös 8-naapurisilla ruudukoilla.

RSR:n perusideana on löytää esikäsittelyssä ruudukosta tyhjiä "huoneita"; suorakaiteen muotoisia esteistä tyhjiä alueita. Näiden huoneiden sisältämät (sekä jotkin niiden reunoilla olevat) solmut karsitaan pois ja korvataan *makrokaarilla* (*macro edges*), jotka yhdistävät huoneen reunoilla olevia solmuja. Makrokaarten painoarvoksi asetetaan solmujen välisen lyhimmän polun pituus, joka on 4-naapurisissa ruudukoissa niiden Manhattan-etäisyys, ja 8-naapurisissa ruudukoissa niiden oktiilinen etäisyys.

Esikäsittelyn jälkeen ruudukossa voidaan käyttää mitä tahansa ruudukolla toimivaa polunetsintäalgoritmia; alkuperäisessä artikkelissaan Harabor, Botea ja Kilby käyttivät A*:ä sopivalla heuristiikalla (oktiilinen tai Manhattan-etäisyys). Polkua haettaessa algoritmi voi oikaista huoneen poikki yhdellä askeleella, vähentäen laajennettavien solmujen määrää ja nopeuttaen hakua. Harabor, Botea ja Kilby totesivat kokeellisesti algoritmin olevan ruudukon rakenteesta riippuen noin 2-8 kertaa tavallista A*:ä nopeampi 8-naapurisissa ruudukoissa, ja 3-18 kertaa nopeampi 4-naapurisissa ruudukoissa.

3.2.1 Algoritmin toiminta

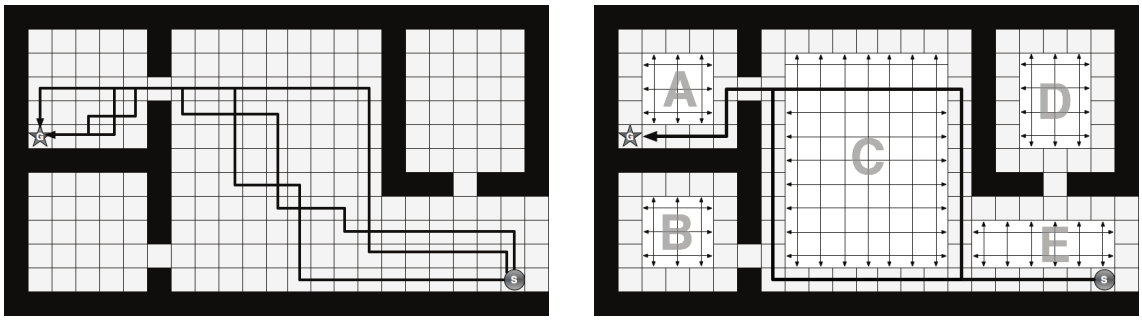
Harabor et al. kuvailevat RSR:n toimintaa neljällä askeleella [8]:

1. Ruudukko jaotellaan suorakaiteen muotoisiksi esteettömiksi huoneiksi. Huoneiden pinta-alan maksimointi on polunetsinnän nopeuttamisen kannalta tärkeää [11].
2. Huoneiden sisäsolmut sekä sellaiset reunasolmut, jotka eivät ole minkään toisen huoneen naapureita karsitaan pois.
3. Karsittujen solmujen tilalle huoneen jäljelle jäävien reunasolmujen välille määritellään *makrokaaria*, joiden painoarvoksi asetetaan solmujen välinen oktiilinen (8-naapurisissa ruudukoissa) tai Manhattan-etäisyys (4-naapurisissa ruudukoissa).
4. Polkua etsittäessä aloitus- ja päätesolmut lisätään väliaikaisesti takaisin karttaan, mikäli ne on huoneita muodostettaessa karsittu pois.

Huoneiden ja makrokaarien muodostaminen

Harabor, Botea ja Kilby esittelevät tulvatäyttöön pohjautuvan yksinkertaisen algoritmin huoneiden muodostamiseen [11]. He toteavat kuitenkin, ettei heidän esittämänsä metodi maksimoi huoneiden kokoa optimaalisesti. Emme käsittele huoneiden muodostamisen teknisiä yksityiskohtia tilan säästämiseksi.

Huoneiden muodostamisen jälkeen huoneen reunoille jäävät solmut yhdistetään toisiinsa makrokaarilla. Makrokaaria luodaan korvaamaan vain sellaisia polkuja, jotka ovat



(a) *(b)*

Kuva 3.3. (a) *Esimerkki polkusymmetriasta 4-naapurisessa ruudukossa. Pisteiden välillä on useita symmetrisiä polkuja, joista tässä esitetään kolme.* ***(b)*** *Sama kartta huoneet muodostettuna. Huoneiden muodostaminen vähentää symmetristen polkujen määrää, mutta ei eliminoi niitä täysin; tähän esimerkkiin on lisätty yksi symmetrinen polku. [11]*

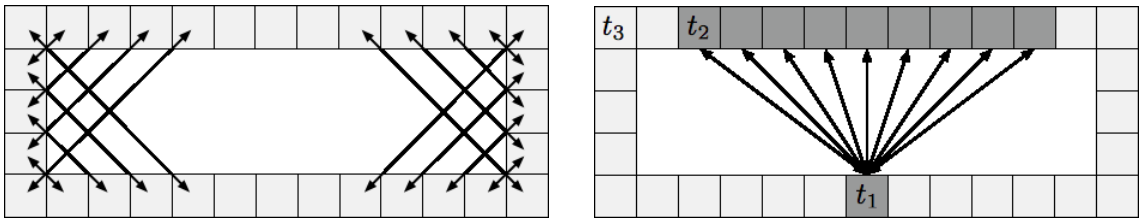
yksioikoisesti lyhin polku solmujen välillä; ei siis saa olla olemassa jotakin toista yhtä pitkää polkua, joka yhdistää solmut ilman makrokaarta. Harabor et al. esittävät 8-naapurisille ruudukoille kolme mahdollista tapausta:

1. Solmut, jotka ovat suorakaiteen samalla sivulla yhdistetään suoraan, kuten alkuperäisessäkin ruudukossa.
2. Solmut, jotka ovat suorakaiteen vierekkäisillä sivuilla yhdistetään jos ja vain jos lyhin polku niiden välillä on diagonaalinen suora viiva. Kuva 3.4a esittää tällaista tilannetta.
3. Solmut, jotka ovat suorakaiteen vastakkaisilla sivuilla yhdistetään seuraavasti. Kullekin tällaiselle solmulle muodostetaan “viuhka” naapureita vastakkaisella puolella suoraan vastakkaisella puolella olevasta naapurista aloittaen, kuten kuvassa 3.4b. Viuhkaa jatketaan reunoja kohti kunnes solmujen välinen kulma on 45° (eli niiden välille voitaisiin alkuperäisessä ruudukossa muodostaa suora diagonaalinen polku), tai kunnes saavutetaan nurkka. Viuhkan ulkopuolella vastakkaisella sivulla olevat solmut, kuten t_3, voidaan saavuttaa optimaalisesti kulkemalla viuhkaan kuuluvan solmun, kuten t_2:n, kautta.

4-naapurisille ruudukoille makrokaaret muodostetaan yksinkertaisesti suoraan vastakkaisiin solmuihin, sekä samalla sivulla oleviin solmuihin mikäli reunoilla olevia solmuja karsittiin huoneita muodostaessa.

Solmujen lisääminen ruudukkoon

Mikäli polunetsinnän alku- tai päätesolmu karsittiin huoneita muodostaessa pois, on solmu lisättävä väliaikaisesti takaisin ruudukkoon hakua tehtäessä. Mikäli alku- ja päätesolmu ovat samassa huoneessa, niiden välinen polku on triviaalisti löydettävissä eikä lisäystä tarvitse tehdä. Muussa tapauksessa solmu sijoitetaan huoneen keskelle ja siitä muodostetaan 8-naapurisessa ruudukossa neljä “viuhkaa”, yksi jokaista huoneen sivua kohden, samalla tavalla, kuin ylempänä tapauksessa 3. 4-naapurisessa ruudukossa solmu yhdistetään yksinkertaisesti ortogonaalisesti lähimpiin naapureihinsa kullakin sivulla.



Kuva 3.4. *Makrokaarten muodostaminen 8-naapurisessa ruudukossa.* ***(a)*** *Makrokaarten muodostaminen vierekkäisten sivujen välillä.* ***(b)*** *Makrokaarten muodostaminen vastakkaisten sivujen välillä. [8]*

Kun alku- ja päätesolmu on väliaikaisesti lisätty ruudukkoon, voidaan niiden välillä oleva lyhin polku hakea käyttämällä mitä tahansa hakualgoritmia. Harabor, Botea ja Kilby käyttävät artikkelissaan A*:ä, jonka heuristiikkana on joko oktiilinen- tai Manhattanetäisyys riippuen siitä, onko ruudukko 4- vai 8-naapurinen.

3.2.2 Algoritmin suoritusaika ja muistivaatimukset

Harabor, Botea ja Kilby vertailevat algoritminsa suoritusaikaa suhteessa A*:een kahden muun samankaltaisen algoritmin, Swamps-algoritmin [17] ja Portal Heuristic -algoritmin [7], kanssa. He käyttivät kokeissaan kahta synteettistä testikarttaa sekä *Baldur's Gate* -pelistä otettua karttaa.

Harabor et al. toteavat RSR:n suoriutuvan Swampsia jopa noin 150% nopeammin, kun ruudukko jakautuu luonnollisesti suorakaiteen muotoisiin huoneisiin. *Baldur's Gate* -pelin kartassa, joka ei jakaudu erityisen hyvin suorakaiteiksi, RSR toimi noin 25% Swampsia hitaammin. Portal Heuristic -algoritmiin verrattuna RSR suoriutui suurin piirtein yhtä nopeasti, mutta käytti jopa 7 kertaa vähemmän muistia.

RSR:n muistivaatimukset ovat Haraborin ja muiden mukaan lineaariset suhteessa graafin solmujen määrään.

3.3 Jump Point Search

Joissakin tilanteissa ruudukon esiprosessointi ei ole mahdollista tai järkevää, jolloin luonnollisesti ei voida myöskään hyödyntää HPA*:ä tai RSR:ää. Haraborin ja Grastienin vuonna 2011 esittelemä **Jump Point Search** (lyhemmin **JPS**) on algoritmi, joka nopeuttaa polunetsintää vähentämällä ruudukon symmetriaa ajonaikaisesti [9].

Jump Point Search hyödyntää solmuja laajentaessaan niin kutsuttuja karsintasääntöjä (*pruning rules*). Karsintasääntöjen avulla algoritmi voi sivuuttaa sellaiset tarkasteltavan solmun naapurit, joihin on olemassa lyhempi tai yhtä lyhyt polku, joka ei kulje tarkasteltavan solmun läpi. Käyttämällä näitä sääntöjä rekursiivisesti jokaiseen solmun naapuriin jota ei poissuljettu karsinnassa, algoritmi löytää niin kutsuttuja **hyppypisteitä** (*jump point*); solmuja, jotka ovat välttämätön osa lyhintä reittiä yhteen tai useampaan naapuriinsa. Keskittymällä näihin solmuihin algoritmi karsii pois symmetrisiä polkuja.

Harabor ja Grastien todistavat, että optimaalinen lyhin polku on mahdollista löytää laajentamalla vain hyppypistesolmuja. Tämä vähentää merkittävästi laajennettujen solmujen määrää, joka poistaa A*:lle tyypillisen prioriteettijonosta muodostuvan pullonkaulan. Toisaalta karsintasääntöjen rekursiivinen käyttö tekee yksittäisistä laajennusoperaatioista hitaampia, kuin A*:n laajennusoperaatiot, jolloin laajennusoperaatiot muodostuvat algoritmin uudeksi pullonkaulaksi [10].

Hitaammasta laajentamisesta huolimatta Jump Point Search tuottaa A*:een verrattuna mittavia nopeusetuja; Haraborin ja Grastienin vertailussa JPS oli yhtä nopea tai jopa nopeampi, kuin aiemmin esitelty ei-luvallinen HPA*-algoritmi, ja toimi parhaimmillaan monikymmenkertaisesti A*:ä nopeammin [9]. Harabor ja Grastien esittivät JPS:ään myöhemmin optimisaatioita, jotka voivat nopeuttaa algoritmia entisestään moninkertaisesti [10].

3.3.1 Algoritmin toiminta

Harabor ja Grastien kuvailevat Jump Point Searchia makro-operaattoriksi, joka vaikuttaa siihen, mitä solmuja A* generoi. He eivät kuitenkaan tarjoa esimerkkitoteutusta algoritmistaan.

Käytännössä Jump Point Search eroaa A*:stä siten, että solmua laajennettaessa sen naapureiden sijaan tarkastellaan sen hyppypisteitä. Harabor ja Grastien esittelevät algoritmin *Identify Successors*, joka saa parametreinään nykyisen solmun sekä polunetsinnän alku- ja loppupisteen, ja palauttaa solmun hyppypisteet. Algoritmissa 2.4 esitellyn A*-algoritmin voisi siis muuntaa JPS:ksi muuntamalla rivin 15 iteroimaan solmun naapureiden sijaan *Identify Successors* -algoritmin palauttamia hyppypisteitä.

Alla käymme tarkemmin läpi, miten JPS määrittelee ja löytää hyppypisteet.

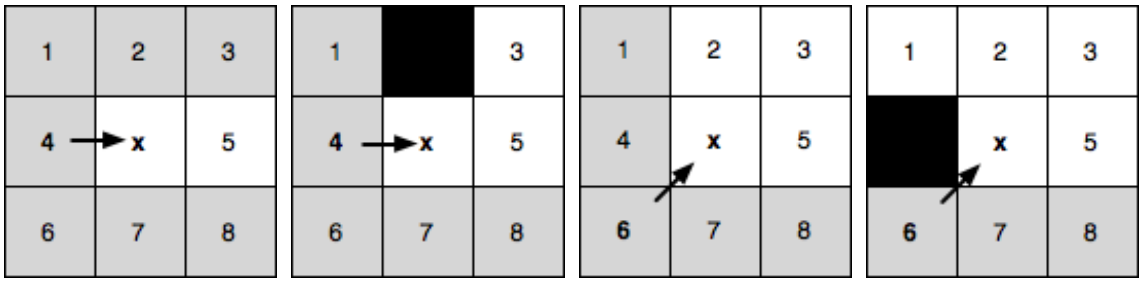
Karsintasäännöt

JPS:n avaimena voidaan pitää kahta yksinkertaista karsintasääntöä, joita käyttämällä algoritmi voi solmua laajentaessaan sivuuttaa naapurisolmut, joita ei ole tarpeen generoida. Näistä kahdesta säännöstä ensimmäistä käytetään ortogonaalisten askeleiden kohdalla, ja toista diagonaalisten askeleiden kohdalla.

Ortogonaaliset askeleet: Mikäli naapurisolmuun voidaan muodostaa nykyisen solmun vanhemmasta alkaen polku, joka ei kulje nykyisen solmun läpi, ja mikäli tuo polku on *lyhempi tai yhtä lyhyt kuin* nykyisen solmun läpi kulkeva polku, voidaan naapuri sivuuttaa.

Diagonaaliset askeleet: Mikäli naapurisolmuun voidaan muodostaa nykyisen solmun vanhemmasta alkaen polku, joka ei kulje nykyisen solmun läpi, ja mikäli tuo polku on *lyhempi kuin* nykyisen solmun läpi kulkeva polku, voidaan naapuri sivuuttaa.

Harabor ja Grastien jaottelevat karsinnan jälkeen jäljellä olevat naapurit kahteen ryhmään. **Luonnolliset naapurit** (*natural neighbour*) ovat sellaisia solmuja, jotka olisivat karsinnan jälkeen kelpaavia naapureita tilanteessa, jossa nykyisen solmun naapurina ei ole yhtään estettä; kuvien 3.5a ja 3.5c valkoiset solmut ovat tällaisia naapureita.



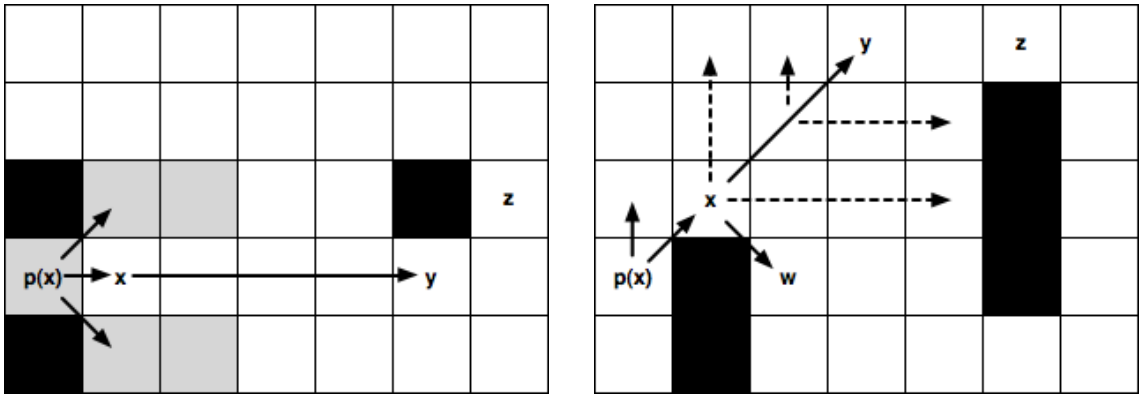
Kuva 3.5. *Esimerkkejä karsintasääntöjen käytöstä. Musta nuoli esittää kulkusuunnan solmun vanhemmasta solmuun. Harmaat ruudut esittävät karsintasäännön mukaan sivuutettavia solmuja. Mustat ruudut ovat esteitä. [9]*

Jos solmun vieressä on este, ei kaikkia sen naapureista voida karsintasäännöillä sivuuttaa; kuvat 3.5b ja 3.5d ovat tästä esimerkki. Harabor ja Grastien kutsuvat tällaisia naapureita **pakollisiksi naapureiksi** (*forced neighbour*).

Hyppypisteet

Haraborin ja Grastienin määritelmän mukaan solmu y on solmun x hyppypiste, mikäli jokin seuraavista ehdoista on tosi:

1. Solmu on polunetsinnän päätesolmu.
2. Solmulla on vähintään yksi pakollinen naapuri.
3. Solmuun astuttiin diagonaalisesti, ja on olemassa ehdon 1 tai 2 täyttävä solmu, joka on saavutettavissa diagonaalisen askeleen suuntaan verrattuna 45° kulmassa suoraan ortogonaalisesti kulkemalla; kuvan 3.6b y-solmu on esimerkki tällä ehdolla löydestystä hyppypisteestä.



Kuva 3.6. *Esimerkki suorasta* ***(a)*** *ja diagonaalisesta* ***(b)*** *hyppypisteestä. Paksunnetut nuolet merkitsevät hyppypistejälkeläisiä. Katkonuolet merkitsevät solmua laajennettaessa tehtyä rekursiota, joka päättyy umpikujaan. [9]*

Hyppypisteiden etsintä suoritetaan solmua laajennettaessa. Laajennettavan solmun naapurit karsitaan. Jäljelle jäävien luonnollisten ja pakollisten naapureiden generoinnin

sijaan algoritmi “hyppää” solmun yli ja tutkii sen jälkeen rekursiivisesti samassa relatiivisessa suunnassa olevia solmuja, kunnes haku joko päättyy esteeseen tai ruudukon reunaan, tai löydetään hyppypisteen ehdot täyttävä solmu. Mikäli tällainen solmu löydetään, alkuperäisen naapurin sijasta generoidaan löydetty hyppypiste.

Huomattavaa on, että hyppypisteen määritelmän 3. kohdan vuoksi diagonaalisesti hypätessä jokaisen arvioitavan solmun kohdalla algoritmi hyppää edelleen ortogonaalisesti kahteen suuntaan tarkistaakseen, löytyykö näistä suunnista hyppypistettä. Algoritmi saattaa siis etenkin diagonaalisia hyppyjä tehdessään tarkastaa rekursiivisesti suurenkin määrän solmuja.

3.3.2 Algoritmin suoritusaika ja muistivaatimukset

Alkuperäisessä artikkelissaan Harabor ja Grastien vertaavat Jump Point Searchia kahteen muuhun algoritmiin, HPA*:een [2] ja Swampsiin [17], vertaillen kokeellisesti algoritmien nopeutta suhteessa A*:een. He käyttivät kokeissaan sekä erilaisia synteettisiä testikarttoja että *Baldur's Gate* ja *Dragon Age* -peleistä saatuja karttoja [9].

Harabor ja Grastien toteavat Jump Point Searchin saavuttavan polun pituudesta ja ruudukon rakenteesta riippuen 2-30 -kertaisesti A*:ä paremman suoritusajan; algoritmin nopeushyödyt ovat suuria, kun kartassa on paljon avointa tilaa ja pieniä, mikäli kartta koostuu kapeista käytävistä. He sanovat JPS:n olevan selkeästi verrokkialgoritmeja, HPA*:ä ja Swampsia, nopeampi kaikissa paitsi yhdessä testissä. *Dragon Age* -pelin kartoissa suoritetussa testissä he totesivat JPS:n ja HPA*:n suoriutuvan suurin piirtein yhtä hyvin.

Vuonna 2014 julkaisemassaan artikkelissa *Improving Jump Point Search* Harabor ja Grastien esittelevät erilaisia optimisaatiotekniikoita, jotka nopeuttavat algoritmin toimintaa entisestään. Osa heidän tekniikoistaan vaatii graafin esikäsittelyä ja osa toimii alkuperäisen algoritmin lailla online-tapaan. He mittaavat optimisaatioiden tehokkuutta alkuperäisen artikkelin tapaan kokeellisesti ja toteavat niiden vähentävän laajennettujen solmujen määrää parhaimmillaan noin puoleen ja nopeuttavan algoritmin suoritusaikaa 2-10 -kertaisesti [10].

Jump Point Search ei käytä hakua tehdessään tavallista A*:ä enempää muistia [9]. Koska JPS generoi A*:ä pienemmän määrän solmuja, pitäen avointen solmujen prioriteettijonon lyhyenä, sen muistivaatimukset saattavat olla jopa A*:ä vähäisemmät.

4 YHTEENVETO JA JOHTOPÄÄTÖKSET

Tutustuimme aluksi polunetsinnän perusteisiin, Dijkstran algoritmiin ja A*:een, joihin valtaosa modernista polunetsinnästä suoraan perustuu. Totesimme näiden algoritmien suotuisat ominaisuudet, kuten A*:n täydellisyyden, optimaalisuuden sekä luvallisuuden, joiden vuoksi ne ovat laajasti käytössä.

Tämän jälkeen tarkastelimme polunetsinnän erikoistapausta: ruudukkokartoilla tapahtuvaa polunetsintää. Esittelimme kolme algoritmia, jotka edustavat erilaisia lähestymistapoja ongelmaan — hierarkiaa hyödyntävän ei-luvallisen HPA*:n [2]; RSR-esiprosessointitekniikan, joka säilyttää löydettyjen polkujen optimaalisuuden [8]; sekä JPS-algoritmin, joka on ilman esiprosessointia toimiva A*:n optimisaatio [9] — ja vertailimme niiden suorituskykyä A*:een sekä muihin ruudukkokarttoihin erikoistuviin algoritmeihin.

Tarkastelemistamme algoritmeista Jump Point Search vaikuttaa yleisesti ottaen parhaalta ratkaisulta polunetsintään tasapainoisissa ruudukkokartoissa. Se ei vaadi kartan esiprosessointia, se säilyttää A*:n täydellisyyden, optimaalisuuden ja luvallisuuden, sen muistivaatimukset ovat yhtä suuret tai pienemmät, kuin A*:llä, ja se suoriutui yhtä nopeasti tai nopeammin, kuin yksikään toinen tarkastelemamme (ja JPS:n tekijöiden tarkastelema) algoritmi.

Yksi mahdollinen suunta jatkotutkimukselle olisi tarkastelemiemme algoritmien yhdistäminen toistensa sekä muiden algoritmien kanssa. Koska algoritmit edustavat eri lähestymistapoja, voi olla mahdollista nopeuttaa hakua entisestään käyttämällä niitä yhdessä jonkin toisen algoritmin kanssa. Tätä mahdollisuutta korostavat myös algoritmien tekijät; esimerkiksi JPS:n esittelevässä artikkelissaan Harabor ja Grastien ehdottavat, että algoritmia voisi käyttää nopeuttamaan HPA*:n toimintaa [9].

Toinen mielenkiintoinen suunta voisi olla algoritmien soveltaminen ruudukkoihin, jotka eivät ole tasapainoisia. Tällaisia ruudukoita käytetään usein esimerkiksi videopeleissä, joissa jotkin ruudut saattavat esittää vaikeakulkuista maastoa. Tarkastelemamme algoritmit eivät sellaisenaan sovellu tällaisiin ruudukkoihin, ja koska ne nojaavat vahvasti olettamukseen ruudukon tasapainoisuudesta, on todennäköistä että niiden sovittaminen vaatisi merkittäviä kompromisseja algoritmin nopeuden suhteen. Tarkastelemistamme algoritmeista Rectangular Symmetry Reduction voisi soveltua tällaisiin ruudukkoihin muita algoritmeja paremmin; huoneita muodostettaessa voitaisiin esimerkiksi käyttää sääntöä, ettei huone saa sisältää kuin tasapainoisia kaaria, tai kaarten vaihtelevat painoarvot voitaisiin ottaa makrokaaria muodostettaessa huomioon.

LÄHTEET

[1] M. J. Atallah ja M. Blanton, toim. *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*. 2. painos. CRC press, 2009.

[2] Botea, A., Müller, M. ja Schaeffer, J. Near Optimal Hierarchical Path-Finding. *Journal of Game Development* 1.1 (2004), 7–28.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L. ja Stein, C. *Introduction to Algorithms*. 2. painos. MIT Press, 2001. ISBN: 0-262-03293-7.

[4] Dechter, R. ja Pearl, J. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM* 32.3 (1985), 505–536.

[5] Dijkstra, E. W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* 1.1 (joulukuu 1959), 269–271. ISSN: 0945-3245.

[6] Frana, P. L. ja Misa, T. J. An Interview With Edsger W. Dijkstra. *Communications of the ACM* 53.8 (2010), 41–47.

[7] Goldenberg, M., Felner, A., Sturtevant, N. ja Schaeffer, J. Portal-Based True-Distance Heuristics for Path Finding. *Third Annual Symposium on Combinatorial Search*. 2010, 39–45.

[8] Harabor, D. D., Botea, A. ja Kilby, P. Path Symmetries in Undirected Uniform-Cost Grids. *Ninth Symposium of Abstraction, Reformulation, and Approximation*. 2011, 58–61.

[9] Harabor, D. D. ja Grastien, A. Online Graph Pruning for Pathfinding on Grid Maps. *Twenty-Fifth AAAI Conference on Artificial Intelligence*. 2011, 1114–1119.

[10] Harabor, D. D. ja Grastien, A. Improving Jump Point Search. *Twenty-Fourth International Conference on Automated Planning and Scheduling*. 2014, 128–135.

[11] Harabor, D. ja Botea, A. Breaking Path Symmetries on 4-Connected Grid Maps. *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2010, 33–38.

[12] Hart, P. E., Nilsson, N. J. ja Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), 100–107.

[13] Hart, P. E., Nilsson, N. J. ja Raphael, B. Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'. *ACM SIGART Bulletin* 37 (1972), 28–29.

[14] Holte, R. C., Perez, M. B., Zimmer, R. M. ja MacDonald, A. J. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. *AAAI/IAAI, Vol. 1*. Citeseer. 1996, 530–535.

[15] Patel, A. *Introduction to the A* Algorithm*. 2014. URL: https://www.redblobgames.com/pathfinding/a-star/introduction.html (viitattu 29. 04. 2020).

[16] Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984. ISBN: 0-201-05594-5.

[17] Pochter, N., Zohar, A., Rosenschein, J. S. ja Felner, A. Search Space Reduction Using Swamp Hierarchies. *Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010, 155–160.

[18] Pohl, I. The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. 1973, 12–17.

[19] S. Russell ja P. Norvig, toim. *Artificial Intelligence: A Modern Approach*. 3. painos. Prentice Hall, 2010.