Rinnakkainen reitinhaku

Santeri Martikainen

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 13. tammikuuta 2017

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author

Santeri Martikainen

Työn nimi — Arbetets titel — Title

Rinnakkainen reitinhaku

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	13. tammikuuta 2017	19

Tiivistelmä — Referat — Abstract

Avainsanat — Nyckelord — Keywords

reitinhaku, multi-agent pathfinding, MAPF

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Reitinhaussa on pohjimmiltaan kyse mahdollisimman suoran ja nopean polun löytämisestä kahden pisteen välillä jossakin topologiassa. Topologia, eli kenttä tai alue, voi olla reaalimaailmassa, kuten tieverkosto autonavigaattorin reitinhaussa tai vaikkapa kokoonpanolinjan robotin käytettävissä oleva liikkumatila. Topologia voi myös olla täysin virtuaalinen, kuten tietokonepelien pelimaailmat. Muita käyttökohteita ovat muun muassa liikenteen ja väkijoukkojen mallintaminen, poliisin ja pelastustoimen tehtävät, sekä tietoliikenneverkot. Käyttökohteesta riippumatta topologia, jossa reittejä etsitään, on mallinnettava tietorakenteiksi sen läpikäyntiä varten.

Oli reitinhaun sovelluskohde mikä hyvänsä, mikäli varsinaista reitinhakua tarvitaan ei reitin löytäminen jokaisella suorituskerralla voi olla täysin suoraviivaista ja triviaalia. Mikäli näin olisi, asia voitaisiin ratkaista yksinkertaisesti laskemalla lyhin etäisyys mitä pitkin liikkua suoraan kohteeseen ja toimia sen mukaan. Kysymys on siis reittien löytämisestä tietyin reunaehdoin. Esimerkiksi navigaattoria käyttävää autoilija haluaa kaikella todennäköisyydellä ajaa määränpäähänsä lyhintä reittiä teitä pitkin oikaisematta yhdenkään metsän tai järven lävitse.

Tässä tutkielmassa tarkastellaan ensin reitinhakua itsessään ja sen ratkaisemista yhden toimijan (agent) reitinhakualgoritmeilla. Jäljempänä käsittelemme monen yhtaikaisen toimijan reitinhakua (multi-agent pathfinding, MAPF) ja siihen kehitettyjä algoritmeja.

2 Reitinhaku

Reitinhaun toteuttaminen jakaantuu kahteen vaiheeseen: Toimintaympäristöstä tai topologiasta muodostetaan ensin yksinkertaistettu malli, minkä jälkeen sitä käydään läpi jollakin algoritmilla halutun reitin löytämiseksi. Algoritmia ohjaa heuristiikkafunktio jolla arvioidaan etäisyyttä kohteeseen ja voidaan siten vertailla eri reittivaihtoehtojen keskinäistä paremmuutta. Jonkin reitin paremmuuteen muihin nähden voi vaikuttaa pituuden lisäksi myös sen nopeus.

Topologiasta riippumatta mallinnuksen tuloksena on useimmiten verkko `G=(V,E)`, missä `V` (vertex) on joukko solmuja ja `E` (edge) joukko solmuja yhdistäviä kaaria. Liikkuminen voi esimerkiksi tapahtua solmusta toiseen, tai sitten solmut voivat edustaa kiintopisteitä joiden läheisyydessä voidaan liikkua ilman erillistä reitinhakua. Solmut voivat myös toimia esteinä, jolloin niihin liikkuminen on estetty. Tämä voi myös olla vain solmun väliaikainen tila esimerkiksi jonkun toisen toimijan ollessa liikkumisen tiellä. Verkko voi siis olla dynaaminen, eli sen rakenne saattaa muuttua kesken reittien läpikäymisen, mikä asettaa omat haasteensa reitinhaulle [1, 2]. Topologia voidaan toisinaan mallintaa myös puurakenteena [3].

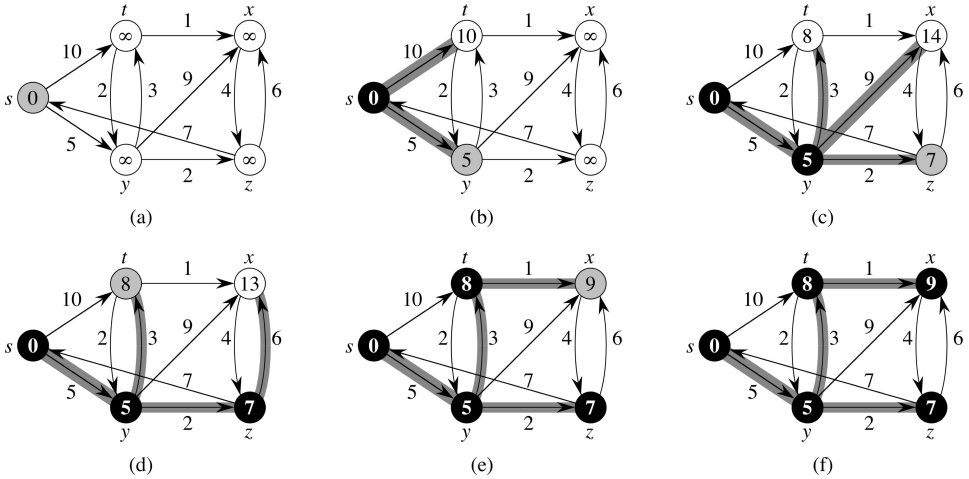
Edellämainittujen esteiden ja muuttuvan toimintaympäristön lisäksi reitinhaun keskeisiä ongelmia on resurssien käyttö. Mitä laajemmassa topologiassa reitinhaku suoritetaan, sitä enemmän laskentaresursseja (muistia, prosessoriaikaa) siihen menee. Joissakin tilanteissa voidaan tyytyä osittaiseen reitinhakuun, eli reittiä ei lasketa loppuun asti, vaan ainoastaan johonkin tiettyyn ennaltamäärättyyn pisteeseen ja uusi reitinhaku tehdään kun jokin välietappi on saavutettu. Tässä on luonnollisesti varmistuttava siitä, että valittuun suuntaan lähteminen todella mahdollistaa perille pääsemisen. Toinen lähestymistapa on kaikkien mahdollisten reittien laskeminen ennakkoon, jolloin haluttu reitti kahden sijainnin välillä yksinkertaisesti haetaan taulukosta tarvittaessa. Tämän menetelmän heikkoutena on, että kyseisestä taulukosta saattaa tulla niin suuri, että varsinainen reitinhaku on nopeampi toteuttaa. Niinikään se vaatii käytännössä mainitun taulukon pitämistä jatkuvasti muistissa.

Reitinhaun standardialgoritmi on jo pitkään ollut Dijkstran algoritmiin perustuva A* (eli A-star tai A-tähti), josta on kehitetty lukuisia eri variantteja eri toimintaympäristöjä ja tarpeita silmälläpitäen. Sen etuja on, että se pystyy varmuudella löytämään reitin kohteeseen, jos sellainen on ylipäätään olemassa. Niinikään se antaa parhaan mahdollisen reitin useista vaihtoehdoista, mikäli sen heuristiikkafunktio ei yliarvioi etäisyyttä kohteeseen [4, 5, 6]. Tärkeisiin heuristisiin algoritmeihin kuuluvat myös iteratiivinen syvyys-A* (iterative-deepening-A*, IDA*) ja syvyyssuuntainen-haarautuva-rajattu (depth-first branch-and-bound, DFBnB) [7]. Resurssitarpeiden kurissapitämiseksi on kehitetty myös erilaisia hierarkiamalleja soveltavia algoritmeja, kuten hierarkinen reitinhaku-A* (hierarchical pathfinding A*, HPA*).

2.1 Dijkstran algoritmi

Edsger Dijkstra esitteli 1959 algoritmin, joka etsii lyhimmät reitit kaikkien graafissa esiintyvien solmujen välillä [8]. Kyseessä on leveyssuuntaista hakua toteuttava algoritmi. Tämä tarkoittaa, että ensin tutkitaan kaikki lähtösolmun naapurit, sitten naapureiden naapurit ja niin edelleen, kunnes joko kaikki solmut on käyty läpi, tai jokin algoritmille asetettu pysähtymisehto on täyttynyt. Tällainen ehto voi olla esimerkiksi reitin löytyminen kahden määritellyn solmun välillä.

Kuvassa 1 nähdään miten viisisolmuisessa graafissa löydetään lyhyimmät reitit solmujen (ympyrät) välillä. Solmuja yhdistää joukko suunnattuja kaaria, joiden vieressä oleva luku kuvaa kaaren painoa, eli sen kulkemisen hintaa. Ainoastaan lähtösolmu s on tässä vaiheessa saanut numeroarvon, joka kuvaa sen etäisyyttä lähtösolmuun. Kohdassa b on tutkittu ne solmut joihin edellisestä solmusta pääsee, sekä asetettu niille etäisyysarvot. C-kohdassa on jälleen jatkettu yksi askel eteenpäin, mutta on huomattava kuinka solmun t arvo on nyt muuttunut b-kohtaan verrattuna. Tämä johtuu siitä, että y-



Kuva 1: Graafin läpikäynti Dijkstran algoritmilla [10].

solmun kautta kulkeva reitti on havaittu lyhyemmäksi kuin suora yhteys s- ja t-solmujen välillä. Tätä jatketaan niin kauan kunnes koko graafi on käyty läpi [8, 9]. Dijkstran algoritmia käytetään edelleen muun muassa joissakin tietoliikenneverkkojen reititysprotokollissa.

2.2 A*-algoritmi

Dijkstran algoritmiin perustuva A* toteuttaa niin sanottua paras ensin -tyyliä, jossa jokaisen solmun tai solun kohdalla pyritään ensiksi etenemään suoraan kohti maalia. Jos tiellä on jokin este, algoritmi pyrkii kiertämään sen. Tämä tapahtuu etenemällä valitsemalla tähänastisen reitin viereisistä soluista ne joista arvioidaan olevan lyhyin etäisyys maaliin. Tätä jatketaan kunnes joko päästään kohteeseen, tai selviää että reittiä ei ole. A* poikkeaa siis Dijkstran algoritmista hakeutumalla koko ajan maalin suuntaan, mikä tapahtuu heuristiikkafunktion avulla (algoritmi 1:n rivi 13).

Kaikille algoritmin käsittelemille solmuille lasketaan arvo kaavalla

$$f(n) = g(n) + h(n) \tag{1}$$

missä $g(n)$ on lyhin tunnettu reitti lähtösolmusta solmuun n ja $h(n)$ on heuristinen arvio etäisyydestä maalisolmuun [5, 11]. Näin jokaisen läpikäydyn solmun kohdalla tiedetään sille lasketusta arvosta kuinka suoralla reitillä kohteeseen ollaan. Solmuille voidaan myös asettaa edellämainittuun kaavaan lisättävä arvo tekemään siihen siirtymisestä hintavampaa ja näin mallintaa hitaampaa kulkuyhteyttä kahden paikan välillä.

Kuvassa 2 on ylempänä havainnollistettu Dijkstran algoritmin löytämä reitti yksinkertaisessa ruudukossa. Liikkuminen on rajoitettu niin sanottuun Manhattan -tyyliin, missä sallitut kulkusuunnat ovat neljä pääilmansuuntaa.

Esteenä toimivat solmut on väritetty tummanharmaalla ja algoritmin läpikäymät solmut turkoosilla. Jälkimmäisen värisävyt kuvaavat laskennallista etäisyyttä vaaleanpunaiseen lähtösolmuun. Kuten kuvasta nähdään, Dijkstra käy läpi varsin suuren osan kentän solmuista, mutta löytää suorimman reitin violetilla merkittyyn kohteeseen.

Saman kuvan alemmalla puoliskolla nähdään miten A*-algoritmin toimintaperiaate poikkeaa Dijkstran algoritmista: Tutkittuja solmuja on paljon vähemmän heuristiikan ohjatessa läpikäynnin suuntaa. Vaikka reitit hieman poikkeavatkin toisistaan, ne ovat annettujen ennakkoehtojen valossa yhtä pitkiä.

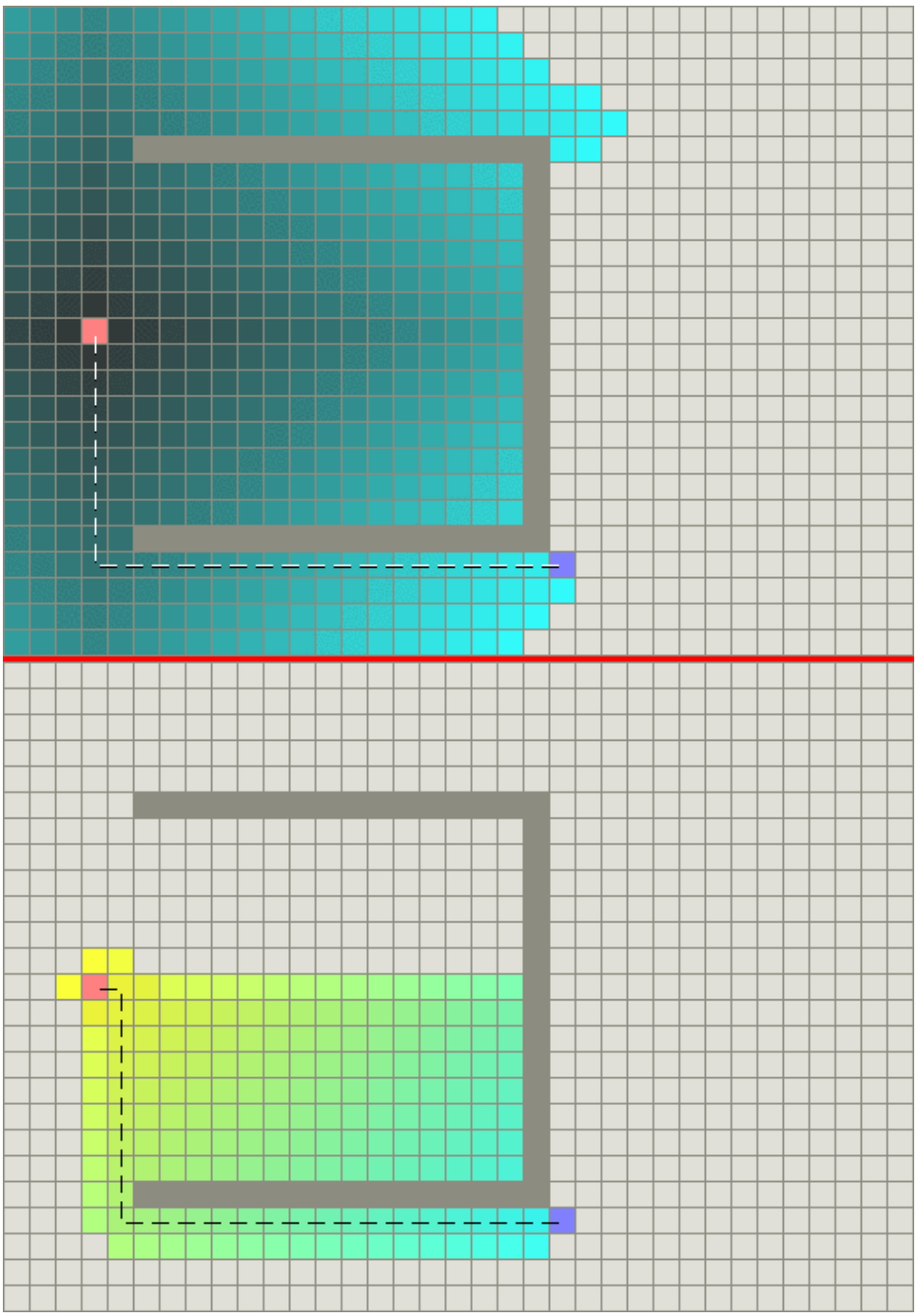
Algorithm 1 A*-algoritmin pseudokoodi

```
1:  procedure ASTAR(startnode, goalnode)
2:      initialize the open list
3:      initialize the closed list
4:      put the starting node on the open list (you can leave its f at zero)

5:      while the open list is not empty do
6:          find the node with the least f on the open list, call it 'q'
7:          pop q off the open list
8:          generate q's 8 successors and set their parents to q
9:          for each successor do
10:             if successor is the goal then
11:                 stop search
12:             successor.g = q.g + distance between successor and q
13:             successor.h = distance from goal to successor
14:             successor.f = successor.g + successor.h
15:             if a node with the same position as successor is in the OPEN
                list which has a lower f than successor then
16:                 skip this successor
17:             if a node with the same position as successor is in the CLOSED
                list which has a lower f than successor then
18:                 skip this successor
19:             otherwise, add the node to the open list
20:         end for
21:         push q on the closed list
22:     end while
23: end procedure
```

2.3 Hierarchical Pathfinding A* (HPA*)

Tämä algoritmi poikkeaa A*:stä siinä, että reitinhaun apuna käytetään yhtä tai useampaa abstraktiotasoa. HPA*:n kehittäjät A.Botea ja M.Müller käyt-



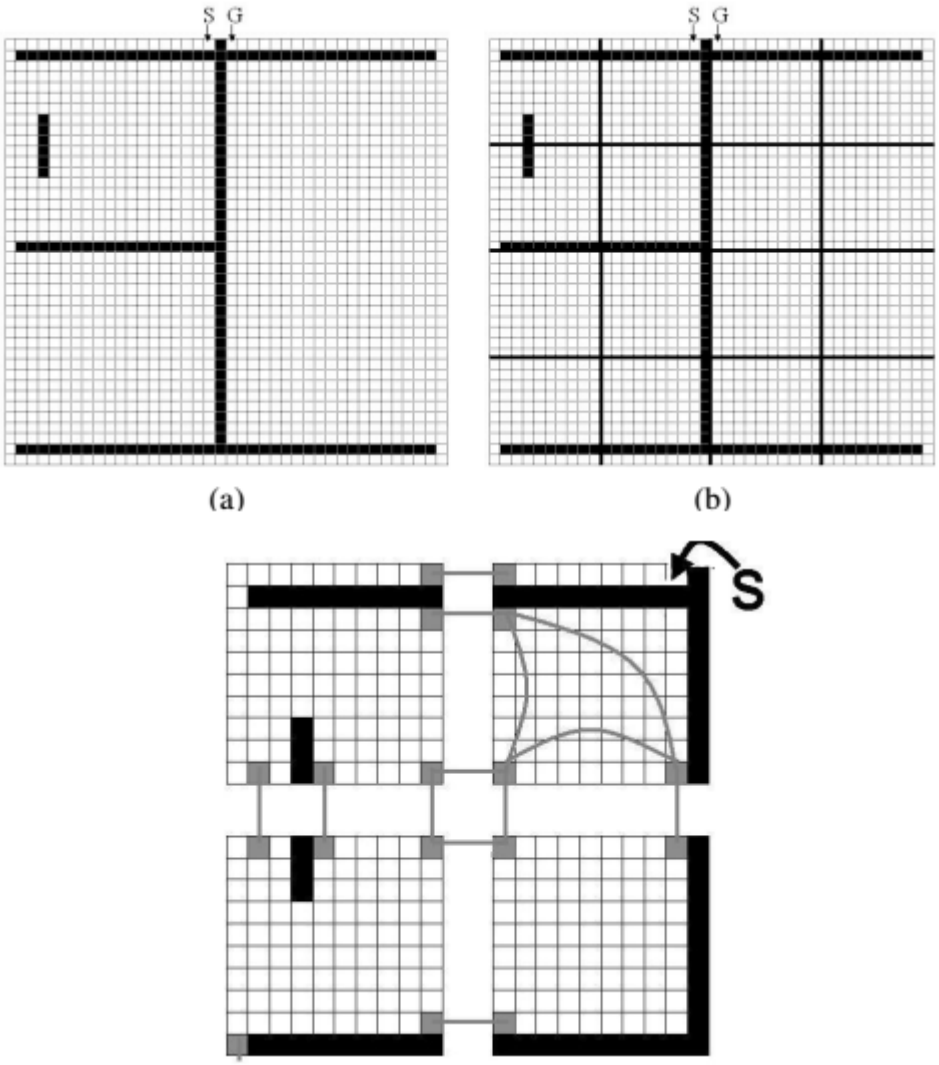
Kuva 2: Ylempänä reitinhaku Dijkstran algoritmilla ja alempana A-algoritmilla [12].*

tävät autovertausta havainnollistamaan algoritmin toimintaa: Kaupungista toiseen ajava autoilija ei ole kiinnostunut kaikista mahdollisista yksityiskohdista matkansa varrella, vaan häntä lähinnä kiinnostaa reitti lähtöpisteestä oikeaan suuntaan vievälle valtatielle, ja kohdekaupunkiin saavuttaessa reitti valtatieltä varsinaiseen määränpäähän. Itse valtatien ajamisen voidaan tässä esimerkissä siis nähdä olevan jokseenkin triviaalia toimintaa, joka ei vaadi sen kummempia toimenpiteitä. Tarkoitus on siis pyrkiä reitinhaun tehokkuuteen eliminoimalla sellaista laskentaa, joka liittyy esteettömien alueiden ylittämiseen [6]. Kuvassa refhpa nähdään abstraktion vaiheet testiruudukossa. Ruudut S ja G merkitsevät lähtö- ja maaliruutuja, ja mustaksi värjätyt ruudut esteitä. Kohdassa b on alkuperäiseen ruudukkoon merkitty kapeilla mustilla viivoilla alueen jako klustereihin, ja alimpana nähdään vasemman yläneljänneksen klusterit harmaalla merkittyine siirtymineen. Tässä esimerkissä on käytetty raja-arvona kuutta ruutua, jolloin alle raja-arvon levyiset avoimet ruutualueet kahden klusterin välillä saavat yhden siirtymän ja muualla siirtymiksi merkitään avoimien ruutualueiden reunat. Kun kentästä on näin muodostettu abstraktiomalli, sitä käytetään reitinhaussa ohjaamaan A*-algoritmia oikeaan suuntaan. Kyseessä on kolmivaiheinen menettely: Ensin absraktiomalliin liitetään aloitus- ja lähtösolmut S ja G. Tämän jälkeen abstraktiomallia käydään läpi A*:llä saamaan selville ne klustereita yhdistävät siirtymät, joiden kautta maaliin tulee kulkea. Tämä tuottaa joukon välietappeja, joita käytetään varsinaisen reitin selvittämiseen. Välietapit pitävät huolen siitä, että reitinhaku etenee koko ajan oikeaan suuntaan, jolloin voidaan välttää turhaa työtä. Algoritmin toimintaa voidaan tietyissä tilanteissa parantaa lisäämällä abstraktiotasoja, jolloin jokainen uusi abstraktiotaso on edellistä karkeampi esitys kentän topologiasta.

3 Rinnakkainen reitinhaku

Rinnakkaisessa reitinhaussa on yleensä useita, joskus jopa satoja tai tuhansia, eri toimijoita. Tällainen monen toimijan rinnakkainen reitinhaku (multi-agent pathfinding, MAPF) tuo yksittäisen toimijan reitinhakuun verrattuna uusia ongelmia. On muun muassa ratkaistava sallitaanko reittien risteäminen, voiko kaksi tai useampi toimijaa olla samaan aikaan samassa paikassa, tuleeko liikkumista porrastaa odottamalla että reitti edessä vapautuu ja tuleeko toimijan väistää ollessaan toisen toimijan tiellä [13]. Näiden ongelmien ratkaisemiseen kehitetyistä algoritmeja useimmat hyödyntävät A*-algoritmia jollakin tapaa [5].

Mitä enemmän toimijoita tutkittavalla alueella on, sitä oleellisemmaksi muodostuu yhteentörmäyksien ja tahattoman reittien sulkemisen välttäminen. Yksi oleellinen ero lähestymistavoissa on käsitelläänkö toimijoita yhdistettynä (coupled) toimijana, vai toisistaan erotettuina (decoupled) toimijoina. Erotetussa menetelmässä reitit lasketaan jokaiselle toimijalle erikseen ja mah-



Kuva 3: Ylärivillä 40x40 -ruudukon jakaminen 16:een klusteriin HPA-algoritmin esiprosessoinnissa, alla vasemman yläneljänneksen klustereiden siirtymät [6].*

dolliset yhteentörmäykset ratkaistaan valitulla menettelyllä sikäli kuin niitä tapahtuu. Tämä menetelmä on tyypillinen tilanteille, joissa toimijoita on verrattaen paljon [14]. Yhdistetyssä menetelmässä kaikkia toimijoita käsitellään sen sijaan kokonaisuutena. Tämä voidaan toteuttaa esimerkiksi varaamalla graafin solmuja toimijoiden käyttöön siksi aikaa kun näiden odotetaan oman reitinhakunsa perusteella niissä olevan. Lukuisten toimijoiden yhtäaikainen huomioonottaminen luonnollisesti lisää ongelman monimutkaisuutta.

Yksi graafien ominaisuuksia on *haarautuvuus* (branching). Tällä tarkoitetaan keskimääräistä lukumäärään naapurisolmuja, joihin jostakin graafin solmusta voidaan liikkua. Jos meillä on graafi, jonka haarautuvuus on b, niin erotetussa menetelmässä jokaisella toimijalla on $O(b+1)$ mahdollista liikettä, eli siirtymät toisiin solmuihin ja paikallaan odottaminen. Yhdistetyssä menetelmässä on sen sijaan joka askeleella otettava myös huomioon k toimijaa, jolloin mahdollisia liikkeitä on $O(b+1)^k$. Tämä hidastaa reittien laskemista [14].

Kuten edellä todettiin, useimmat rinnakkaisen reitinhaun ratkaisemiseen kehitetyt menetelmät ovat A*-pohjaisia. A*-agoritmi vaatii kuitenkin verrattaen paljon resursseja käyttöönsä, mikäli kyse on laajemmasta reitinhakuongelmasta. Siksi onkin kehitetty useita erilaisia lähestymistapoja, joilla pyritään pitämään läpikäytävien solmujen määrä, ja siten laskentaresurssien tarve, mahdollisimman pienenä.

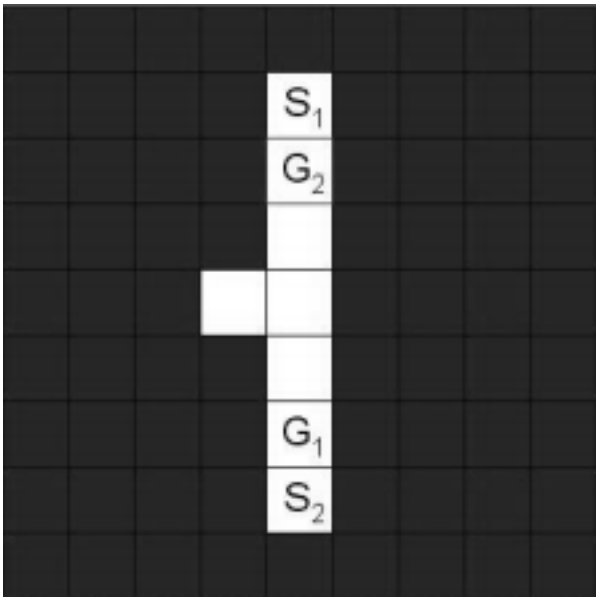
4 A*-pohjaiset rinnakkaisen reitinhaun algoritmit

4.1 Local Repair A* (LRA*)

LRA* on yleistermi joukolle A*-pohjaisia algoritmeja, jotka jakavat saman toimintaperiaatteen: Jokainen toimija etsii reitin A*:lla ja seuraa sitä siihen asti kunnes siirtyminen seuraavaan solmuun saisi aikaan yhteentörmäyksen jonkun toisen toimijan kanssa, eli solmu johon pitäisi siirtyä on varattu. Tällöin tehdään uusi A*-haku ja jatketaan niin kauan kunnes on tultu maaliin.

Syklit ovat tässä menetelmässä sekä mahdollisia että yleisiä, joten ongelmaa on pyritty ratkaisemaan lisäämällä niin kutsuttua kohinaa etäisyysheuristiikkaan joka kerta kun yhteentörmäys havaitaan [15]. Jos siis jatkuvasti kohdataan esteitä jossakin solmussa tai jollakin alueella, niin algoritmin laskeman kustannuksen sinne etenemisestä pitäisi ennenpitkää nousta niin suureksi, että toimija hakeutuu ongelma-alueen ympäri tai etsii kokonaan uuden reitin.

Tällainen lähestymistapa johtaa ruuhkatilanteissa helposti oudolta näyttävään poukkoiluun, ja sen myötä prosessoriajan hävikkiin laskettaessa reitti jokaisen yhteentörmäyksen yhteydessä uudestaan.



Kuva 4: Esimerkki kohtaamis-ongelmasta, jossa toimijat S1 ja S2 voivat päästä maalisolmuihin G1 ja G2 vain jos jompikumpi väistää sivulle.

4.2 Cooperative A* (CA*)

CA* pyrkii estämään yhteentörmäykset ennalta ottamalla aikaulottuvuuden huomioon reittejä suunniteltaessa. Jokaiselle toimijalle lasketaan reitti A*-algoritmilla ja suunnitellun reitin solut merkitään taulukkoon, esimerkiksi kolmiulotteiseen hajautustauluun, jossa kaksi ensimmäistä alkiota merkitsevät x- ja y-koordinaatteja, ja kolmas alkio aikaa milloin kyseinen solmu on tämän toimijan käytössä. Nyt seuraavien toimijoiden reittejä laskettaessa voidaan verrata suunniteltuja askeleita edellämainittuun taulukkoon ja odottamalla havaituissa törmäystilanteissa halutun reittisolmun vapautumista.

Tämän algoritmin heikkoutena on kyvyttömyys ratkaista joitakin verrattaen yksinkertaisia tilanteita. Kuvassa 4 nähdään tilanne, missä toimijat S1 ja S2 pyrkivät vastaavasti maaleihin G1 ja G2. Intuitiivisesti nähdään, että jommankumman toimijan olisi mahdollista joko väistää sivulle tai nämä voisivat vaihtaa paikkoja kohdatessaan ja jatkaa sitten määränpäihinsä. Perusmuotoinen CA* ei kuitenkaan pysty tällaiseen ratkaisuun, sillä algoritmissa ei ole tällaista toiminnallisuutta.

4.3 Hierarchical Cooperative A* (HCA*)

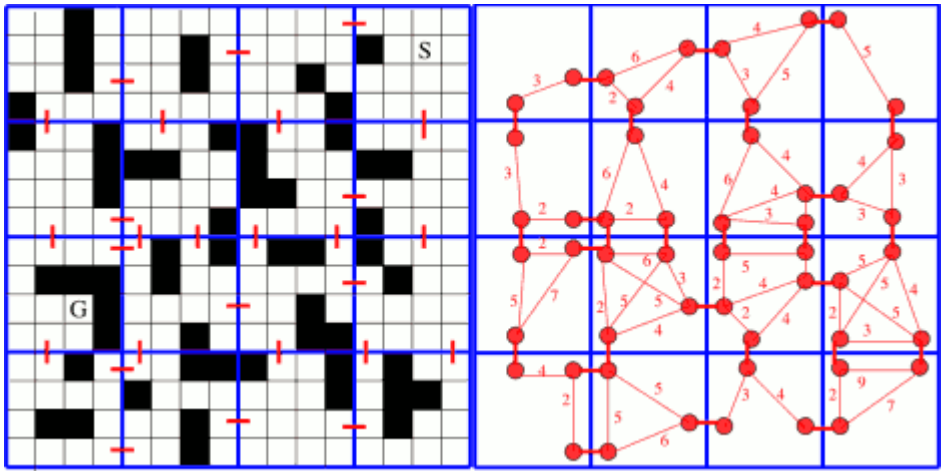
R.C.Holte ja kumppanit esittelivät vuonna 1996 hierarkkisen A*-reitinetsintäalgoritmin HA* [16] ratkaisemaan CA*-algoritmin ongelmia. Siinä varsinai-

sen topologian rinnalla käytetään toista, abstraktia, topologiaa auttamaan A*-agoritmia löytämään kohteeseen. Alkuperäisen topologian solmut ryhmitetään halutun abstraktioetäisyyden perusteella isommiksi abstraktiosolmuiksi, ja tätä jatketaan rekursiivisesti niin kauan, kunnes koko topologiasta on muodostettu yksittäinen abstraktiosolmu. Varsinaisen topologian rinnalla on nyt siis monitasoinen abstraktiohierarkia, jonka alimmalla tasolla on alkuperäinen topologia. Tätä monitasoista hierarkiaa käytetään heuristiikan apuna ja liikkumalla siinä tasolta toiselle sen mukaan miten tarkkaa jakoa esimerkiksi esteiden ympärillä tarvitaan, ja käyttämällä sen antamia etäisyyksiä reitinhakualgoritmin apuna. Etäisyydet kohteeseen lasketaan vain tarvittaessa [15]. [aukaisu]

HCA* eli hierarkkinen yhteistoiminnallinen A*-reitinetsintäalgoritmi puolestaan käyttää yksinkertaistettua hierarkiaa, joka koostuu vain yhdestä topologian kaksiulotteisesta abstraktiosta jättäen huomiotta niin aikaulottuvuuden, solmujen varaustaulun, kuin muut toimijat. Kuvassa 2 nähdään esimerkki topologian abstraktiosta, missä vasemmalla oleva ruudukko on mallinnettu graafiksi jakamalla se isommiksi abstraktiosolmuiksi (siniset kehykset). Kirjaimet S ja G viittaavat lähtö- ja maalisolmuihin. Oikeanpuoleisessa kuvaan on puolestaan merkitty punaisella jokaisen abstraktiosolmun sisällä olevat solmut ja niitä yhdistävät kaaret. Kaaren vierellä oleva luku tarkoittaa kaaren painoa, eli kuinka suuri hinta kaaren kulkemisella on.

Kun reitti on laskettava uudelleen esimerkiksi jonkun toisen toimijan tullessa eteen, pyritään säästämään resursseja käyttämällä käänteistä jatkettavaa A*-hakua RRA* (Reverse Resumable A*) abstraktiotasolla. Siinä missä alkuperäinen reitti haettiin lähtöpisteestä maaliin, RRA* etsii reitin maalista haluttuun solmuun, kuten toimijaa lähimpään abstraktiotason solmuun [15]. Mikäli tässä vaiheessa optimaalisen reitin varrella on muita toimijoita, tulee lasketusta reitistä näiden väistämisen vuoksi luonnollisesti pidempi, aivan kuten alkuperäisen reitinhaunkin suhteen.

Ongelmana tämän algoritmin käytössä on reitinhaun lopettamisen määritteleminen, sillä vaikka jokin toimija olisi jo tullut päämääräänsä, sen täytyy mahdollisesti vielä väistää jotain toista toimijaa ja hakeutua tämän jälkeen uudestaan maaliin [1]. Niinikään toimijoiden vuorojärjestyksellä on väliä: Staattinen vuorottelu voi johtaa siihen, ettei reittiä maaliin koskaan löydetä joidenkin toimijoiden sulkiessa toisiltaan tien. Tämä voidaan välttää antamalla toimijoille erilaiset prioriteetit jo alun alkaen, tai sitten sitten korkeampi prioriteetti voidaan antaa halutuille toimijoille vuorotellen lyhyeksi aikaa [15]. Tässä, kuten aiemmissakin yhteistoiminnallisissa reitinhakualgoritmeissa, on resurssien käytön suhteen ongelmana potentiaalisesti turhaan tehty työ.



Kuva 5: Vasemmalla olevasta ruudukosta on muodostettu hierarkiamallin mukainen abstraktio.

4.4 Windowed Hierarchical Cooperative A* (WHCA*)

WHCA* eroaa HCA*:sta siinä, että abstraktiotasolla reitti lasketaan maaliin ast, mutta konkreettisen topologian tasolla reitinhaku on rajoitettu johonkin ennaltamääriteltyyn syvyyteen [15, 1]. Kun reittiä on kuljettu johonkin ennaltamääriteltyyn raja-arvoon asti, esimerkiksi puolet aiemmin lasketusta reitistä, lasketaan uusi osittainen reitti ja niin edelleen. ”Windowed” tarkoittaa tässä siis aikaikkunaa tai kehystä, mihin asti konkreettinen reitti on nähtävillä ja mitä siirretään aina tarpeen mukaan. Resurssien käyttöä voidaan myös tasata antamalla toimijoille erikokoiset ikkunat, jolloin taakka reittien laskemisesta saadaan jakautumaan tasaisemmin. Kuten HCA*, myös WHCA* hyödyntää käänteistä jatkettavaa A*-hakua eli RRA*:ta. Näin voidaan hyödyntää edellisen ikkunan aikana tehtyä hakua, mikä toisaalta tarkoittaa toimijakohtaista kirjanpitoa läpikäydyistä solmuista. Koska yksittäisiä toimijoita WHCA*-algoritmissa ei johdeta keskitetysti, vaan ne pyrkivät itsenäisesti etsimään lyhintä reittiä maaliin, tuloksena voi olla tarpeetonta törmäilyä tilanteessa missä muuten olisi runsaasti tilaa lähettyvillä [5].

4.5 Conflict-Based Search (CBS)

Rinnakkaisen reitinhaun optimoimiseen pyrkivä CBS-algoritmi [5] käyttää kaksitasoista lähestymistapaa, missä ensin käydään läpi ylemmän tason binääristä rajoituspuuta (constraint tree). Rajoituspuun jokaisessa solmussa on joukko rajoitteita (tieto siitä milloin joku topologian solmu on jonkun toimijan varaama), jotka kuuluvat jollekin yksittäiselle toimijalle. Toiseksi rajoituspuun solmuihin on myös talletettu joukko alemmalta tasolta saatuja reittejä, yksi jokaista toimijaa kohti, joiden tulee olla vapaita tiedetyistä yhteentörmäyksistä. Kolmanneksi niissä on myös yhteenlaskettu solmun

reittikustannus, joka koostuu yksittäisen toimijan koko siihenastisen polun kustannuksesta. Rajoituspuu on järjestetty reittikustannuksen mukaan.

Alemmalla tasolla puolestaan voidaan käyttää tavanomaista yhden toimijan reitinhakualgoritmia kuten A* hyödyntäen samalla ylemmältä tasolta saatua tietoa siitä, missä ja milloin on odotettavissa yhteentörmäys jonkun toisen toimijan kanssa. Mikäli kaikesta huolimatta alemmalla tasolla havaitaan yhteentörmäyksiä, päivitetään ylemmän tason rajoituspuuta vastaavasti laajentamalla rajoituspuuta lisäämällä siihen solmuja. Tämän jälkeen tehdään uusi alemman tason haku ja toistetaan edellinen vaihe tarvittaessa.

Algorithm 2 ICBS-algoritmin ylempi taso

```
1:  Main(MAPF problem instance)
2:      Init R with low-level paths for the individual agents
3:      insert R into OPEN
4:      while OPEN not empty do
5:          N ← best node from OPEN // lowest solution cost
6:          Simulate the paths in N and find all conflicts
7:          if N has no conflict then
8:              return N.solution // N is goal
9:          C ← find-cardinal/semi-cardinal-conflict(N) // (PC)
10:         if C is not cardinal then
11:             if Find-bypass(N, C) then // (BP)
12:                 Continue
13:         if should-merge(a_i, a_j) then // Optional, MA-CBS:
14:             a_ij = merge(a_i, a_j)
15:             if MR active then // (MR)
16:                 Restart search
17:             Update N.constraints()
18:             Update N.solution by invoking low-level(a_ij)
19:             Insert N back into OPEN
20:             continue // go back to the while statement
21:         foreach agent a_i in C do
22:             A ← Generate Child(N, (a_i, s, t))
23:             Insert A into OPEN

24: Generate Child(Node N, Constraint C = (a_i, s, t))
25:     A.constraints ← N.constraints + (a_i, s, t)
26:     A.solution ← N.solution
27:     Update A.solution by invoking low level(a_i)
28:     A.cost ← SIC(A.solution)
29:     return A
```

5 Muut rinnakkaisen reitinhaun algoritmit

5.1 Increasing Cost Tree Search

Vuonna 2012 julkaistu ICTS-algoritmi on kaksitasoista hierarkiaa hyödyntävä algoritmi [14]. Hierarkian ylemmällä tasolla käydään läpi puuta nimeltä *increasing cost tree*. Se muodostuu solmuista, joista jokaisessa oleva taulukko $[C_1, ..., C_k]$ edustaa kaikkia niitä mahdollisia reittiratkaisuja, missä toimijoiden reittien hinta on täsmälleen taulukkoon niitä vastaaville paikoille generoitu luku. Toisin sanoen toimijan a_i reittikustannus löytyy taulukon kohdasta C_i.

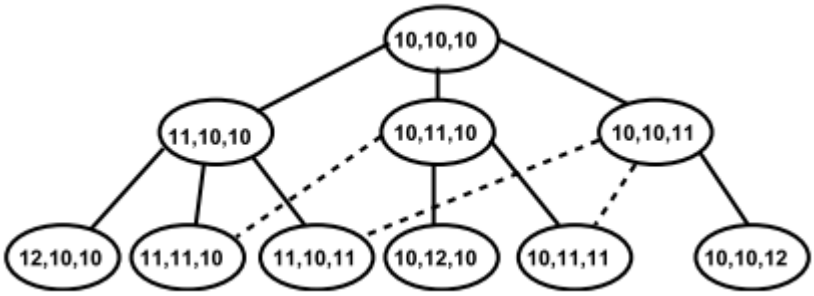
Kuvassa 6 on havainnollistettu tällaisen puun rakennetta kolmen toimijan tapauksessa. Jokaisessa puun solmussa on kolme lukua, jotka siis edustavat näiden kolmen toimijan etäisyyttä maaliin solmussa pidettyjen reittiratkaisujen osalta. Juurisolmun lukemat ovat toimijoiden suorimmat mahdolliset etäisyydet maaliin, eli tilanne missä maaleihin voitaisiin kulkea välittämättä lainkaan muista toimijoista. Tässä esimerkissä jokaisen kolmesta toimijasta laskennallinen etäisyys maaliin on 10. Juurisolmulle ja jokaiselle solmulle sittemmin lisätään toimijoiden määrää vastaava määrä lapsisolmuja. Näihin lapsisolmuihin puolestaan lisätään siinä oleville toimijoille jokaiselle vuorollaan jokin lukuarvo (tässä 1) edustamaan pidempää reittiä. Toisen rivin ensimmäisessä solmussa on siis ensimmäinen toimija saanut arvon 11, toisessa solmussa keskimmäinen ja kolmannessa viimeinen.

ICT-puuta läpikäytäessä jokaisen solmun kohdalla kutsutaan alemman tason rutiinia tarkistamaan josko solmun määrittämä reittiyhdistelmä on löytynyt. Eli aluksi kokeillaan onko juurisolmun etäisyyksillä löydettävissä reitit kaikille toimijoille ja jos ei, siirrytään puussa alaspäin. Mitä alemmas puussa siis mennään, sitä suurempi on solmuissa olevien reittien hintojen summa. Puuta käydään läpi leveyssuuntaisella haulla juuresta lähtien, joten ensimmäinen löydetty toteutettavissa oleva reittiratkaisu on myös väistämättä halvin.

Jokaista ICT-solmua tutkittaessa suoritetaan siis reittivertailu alemmalla hierarkiatasolla. Tämä tapahtuu käymällä läpi kunkin toimijan kaikki sellaiset reitit, joiden enimmäishinta on kyseiselle toimijalle ICT-solmussa asetettu arvo, ja vertailemalla näitä keskenään kunnes tuloksena on joukko ristiriidattomia reittejä. Koska eri reittejä voi olla hyvinkin suuri määrä, niiden käsittelemiseen on kehitetty oma algoritminsa, *multi-value decision diagram* (MDD). Varsinaisten reittien muodostamiseen konkreettisessa topologiassa voidaan käyttää sopivaksi katsottua yhden toimijan reitinhakualgoritmia.

5.2 Tree-based agent swapping strategy (TASS)

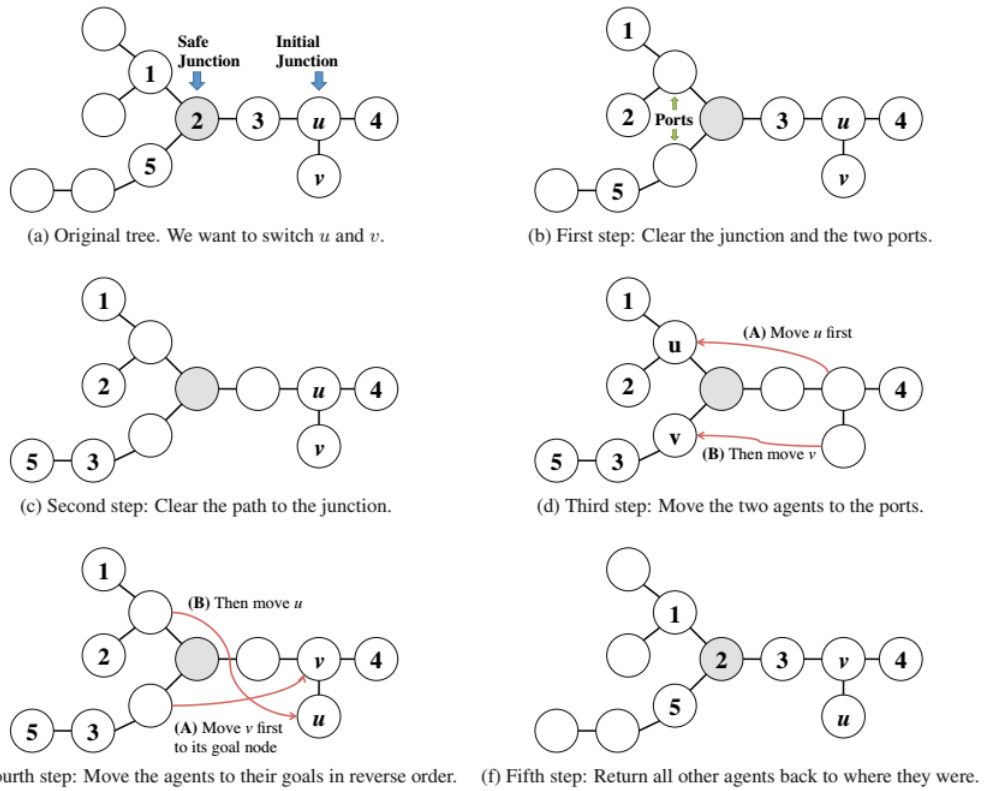
Tämä vuoden 2011 algoritmi hyödyntää graafien sijaan puurakenteita ja olemassaolevien graafien konvertoimiseen onkin kehittäjien toimesta luotu



Kuva 6: Increasing Cost Tree kolmella toimijalla [5].

Algorithm 3 Increasing Cost Tree

Input: *(k, n)* MAPF
Build the root of the ICT
foreach *ICT node in a breadth-first manner* **do**
    foreach *agent* a_i **do**
        Build the corresponding MDD_i
    [**//optional**
    foreach *pair (triple) of agents* **do**
        Perform node-pruning
        if *node-pruning failed* **then**
            Break *//Conflict found. Next ICT node*
    ]
    Search the k-agent MDD search space *// low-level search*
    if *goal node was found* **then**
        return Solution



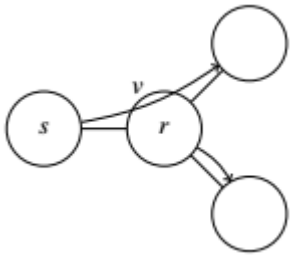
(a) Original tree. We want to switch u and v.
(b) First step: Clear the junction and the two ports.
(c) Second step: Clear the path to the junction.
(d) Third step: Move the two agents to the ports.
(e) Fourth step: Move the agents to their goals in reverse order.
(f) Fifth step: Return all other agents back to where they were.

Kuva 7: Haarautuman hyväksikäyttö uudelleenjärjestämisessä TASS-algoritmilla [3].

Graph-to-Tree Decomposition (GTD)-algoritmeja. TASS on kehitetty erityisesti tilanteisiin, joissa vapaata liikkumatilaa on hyvin vähän. TASS:n toimintaperiaatteena on siis topologian käsittely puuna, jossa siirtymät solmujen välillä on sallittu molempiin suuntiin. Puiden rakenteesta johtuen risteyksiä (tai haaroja) käytetään hyväksi toimijoiden uudelleenjärjestämisessä silloin, kun toimijat ovat toistensa tiellä. Tämä on havainnollistettu kuvassa 7. Alkiot u ja v eivät voi tässä algoritmissa vaihtaa paikkaa päikseen, vaan ne on uudelleenjärjestettävä tekemällä tilaa lähimpään risteyssolmuun ja sen naapureihin, siirrettävä alkiot risteyksen yli uudelleenjärjestämistä varten, ja sitten palautettava lähtötilanne muilta osin ennalleen. Suurempi haarautuvuus nopeuttaa algoritmin toimintaa [3].

5.3 Push and Swap

Push and Swap edustaa rinnakkaisen reitinhaun uutta sukupolvea (2011) ja on verrattaen yksinkertainen toimintaperiaatteeltaan. Nimensä mukaisesti algoritmissa on kaksi keskeistä operaatiota, *Push* ja *Swap*, sekä kaksi apuoperaatiota, *Clear* ja *Resolve*. Toimijoiden suoritusjärjestys määräytyy jonkin ennaltamääritellyn prioriteettijonon perusteella [17].



Kuva 8: Push and Swap-algoritmi: Alkioiden r ja s paikat vaihdetaan Swap-operaatiolla [17].

Push-operaatio liikuttaa toimijaa r kohti maalia. Mikäli maaliin ei päästä jonkun toisen toimijan s tukkiessa tien ja ollessa alempi prioriteetiltaan, Push yrittää työntää tiellä olevat toimijat pois edestä vapaisiin solmuihin ja jatkaa sitten maaliin. Mikäli vapaata tilaa ei ole, työntäminen vaatisi r:n itsensä siirtymistä, työnnettävällä toimijalla on korkeampi prioriteetti, tai pitäisi työntää jo omiin maaleihinsa löytäneitä toimijoita sivuun, otetaan Swap-operaatio käyttöön.

Swap siirtää sekä r:n että s:n lähimpään sellaiseen paikkaan graafissa, jossa on riittävästi tilaa paikkojen vaihtamiseen, kuten nähdään kuvassa 8. Seuraavaksi suoritetaan Clear-operaatio, jolla pyritään tyhjentämään kohdesolmun viereiset solmut työntämällä niissä olevat toimijat muihin vapaisiin solmuihin. Esimerkkikuvassamme kohdesolmu on keskellä sijaitseva v, jonka vierellä on kaksi tyhjää solmua joihin sekä r että s voivat väistää. Jos Clear onnistui, palautetaan mahdollisesti edestä työnnetyt toimijat takaisin alkuperäisille paikoilleen. Toisin kuin Push, Swap voi siirtää korkeamman prioriteetin omaavia, sekä omiin maaleihinsa jo päässeitä toimijoita paikoiltaan. Tämä voi johtaa siihen, että r on nyt jonkun muun toimijan t maalisolmussa. Tilanne ratkaistaan Resolve-operaatiolla: Ensin r yritetään työntää edelleen kohti maaliaan. Mikäli tämä onnistuu, voidaan t palauttaa maalisolmuunsa. Mikäli ei, Swap:ia kutsutaan r:lle uudestaan. Jos tämä ei onnistu, suoritetaan Swap t:lle [18].

Tässä algoritmissa vaivaa kuitenkin joukko ongelmia: Niin kutsuttujen *monikulmiograafien* (polygon graph) tapauksessa algoritmi voi jäädä silmukkaan. Kyseiset graafit ovat määritelmällisesti sellaisia, että jokaisella solmulla on korkeintaan kaksi naapuria. Swap voi myös johtaa laittomaan tilaan (illegal state), mikäli sen toteuttaminen saa aikaan muita Swap-operaatioita ahtaassa tilassa, jolloin algoritmin rakenteesta johtuen operaation ennakkoehdot eivät tietyissä olosuhteissa enää päde. Graafissa esiintyvä kannas (isthmus), eli osio missä on peräkkäisiä solmuja joilla on vain kaksi naapuria, voi johtaa lukkiutumiseen. Clear-operaatio ei osaa käsitellä tilannetta, missä siirtämällä tiellä olevaa toimijaa kahdesti saataisiin riittävästi tilaa paikanvaihtoa varten.

5.4 Push and Rotate

Push and Rotate vuodelta 2013 on Push and Swap-algoritmin jatkokehitetty versio, joka pyrkii paikkaamaan siinä havaittuja puutteita: Monikulmiograafien ongelma on ratkaistu priorisoimalla reitinhaussa sellaisia reittejä, joiden varrella ei ole lainkaan jo maaliin päässeitä toimijoita. Clear-operaation toimintaa on laajennettu ja niin kutsuttu *kannasongelma* on ratkaistu jakamalla kokonaisuus eri tavalla priorisoituihin aliongelmiin. Niinikään tekijöiden *recursive resolve*:ksi nimeämä ongelma ratkaistaan uudella Rotate-operaatiolla [18].

6 Yhteenveto

Reitinhaku ja sen erityistapaus rinnakkainen reitinhaku pysyvät avoimina ongelmina vuosikymmeniä kestäneestä tutkimuksesta huolimatta. Syynä tähän on ensisijaisesti algoritmien vaihteleva soveltuvuus eri tarpeisiin, ja toisaalta kasvaneet tehokkuusvaatimukset etenkin tietokonepelien saralla. Kehitys on toisaalta kulkenut kohti algoritmeja, jotka hyödyntävät useampitasoista hierarkiaa vähentämään laskentaresurssien tarvetta abstraktion kautta, sekä toisaalta kohti perinteistä poikkeavia lähestymistapoja, kuten *Push and Swap* tai *Push and Rotate*. Mikäli liikkumatila ei ole minimissään ja toimijoita on paljon, ovat *erotetut* algoritmit, joissa toimijoita käsitellään itsenäisinä tekijöinä, yleensä tehokkaampia ratkaisuja. Ne eivät tavallisesti kuitenkaan johda yhtä optimaalisiin reittivalintoihin kuin *yhdistetyt* algoritmit, joissa toimijoita käsitellään kokonaisuuden osina ja joiden reittejä suunnitellaan alusta asti muita toimijoita silmälläpitäen [17]. Ohjelmoijan vastuulle jääkin pohtia topologiaa, toimijoiden lukumäärää ja laskentanopeuden tarvetta sopivaa reitinhakualgoritmia valitessaan. Kaikkiin tilanteisiin sopivaa yleisalgoritmia ei ole olemassa, eikä sellaista kenties ole mahdollista toteuttaakaan.

Lähteet

[1] Botea, Adi, Bouzy, Bruno, Buro, Michael, Bauckhage, Christian ja Nau, Dana: *Pathfinding in games.* Dagstuhl Follow-Ups, 6, 2013.

[2] Algfoor, Zeyad Abd, Sunar, Mohd Shahrizal ja Kolivand, Hoshang: *A comprehensive study on pathfinding techniques for robotics and video games.* International Journal of Computer Games Technology, 2015:7, 2015.

[3] Khorshid, Mokhtar M, Holte, Robert C ja Sturtevant, Nathan R: *A polynomial-time algorithm for non-optimal multi-agent pathfinding.* Teoksessa *Fourth Annual Symposium on Combinatorial Search*, 2011.

[4] Cui, Xiao ja Shi, Hao: *A*-based pathfinding in modern computer games.* International Journal of Computer Science and Network Security, 11(1):125–130, 2011.

[5] Sharon, Guni, Stern, Roni, Felner, Ariel ja Sturtevant, Nathan R: *Conflict-based search for optimal multi-agent pathfinding.* Artificial Intelligence, 219:40–66, 2015.

[6] Botea, Adi, Müller, Martin ja Schaeffer, Jonathan: *Near optimal hierarchical path-finding.* Journal of game development, 1(1):7–28, 2004.

[7] Felner, Ariel, Korf, Richard E ja Hanan, Sarit: *Additive pattern database heuristics.* J. Artif. Intell. Res.(JAIR), 22:279–318, 2004.

[8] Dijkstra, Edsger: *Numerische Mathematik.* Numerische Mathematik, 1(1):269–271, 1959.

[9] Skiena, S: *Dijkstra's algorithm.* Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley, sivut 225–227, 1990.

[10] Bloomington, Indiana University: *Representing graphs: Undirected*, 12.1.2017. `https://www.cs.indiana.edu/~achauhan/Teaching/B403/LectureNotes/10-graphalgo.html`, vierailtu 2017-01-12 .

[11] Stout, Bryan: *Smart moves: Intelligent pathfinding.* Game developer magazine Vol. 10, sivut 28–35, 1996.

[12] University, Stanford: *Introduction to A**, 12.1.2017. `http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html`, vierailtu 2017-01-12 .

[13] Erdem, Esra, Kisa, Doga Gizem, Öztok, Umut ja Schueller, Peter: *A General Formal Framework for Pathfinding Problems with Multiple Agents.* Teoksessa *AAAI*, 2013.

[14] Sharon, Guni, Stern, Roni, Goldenberg, Meir ja Felner, Ariel: *The increasing cost tree search for optimal multi-agent pathfinding.* Artificial Intelligence, 195:470–495, 2013.

[15] Silver, David: *Cooperative Pathfinding.* AIIDE, 1:117–122, 2005.

[16] Holte, Robert C, Perez, Maria B, Zimmer, Robert M ja MacDonald, Alan J: *Hierarchical A*: Searching abstraction hierarchies efficiently.* Teoksessa *AAAI/IAAI, Vol. 1*, sivut 530–535. Citeseer, 1996.

[17] Luna, Ryan ja Bekris, Kostas E: *Push and swap: Fast cooperative path-finding with completeness guarantees.* Teoksessa *IJCAI*, sivut 294–300, 2011.

[18] Wilde, Boris de, Mors, Adriaan W ter ja Witteveen, Cees: *Push and rotate: cooperative multi-agent path planning.* Teoksessa *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, sivut 87–94. International Foundation for Autonomous Agents and Multiagent Systems, 2013.