

Rinnakkainen reitinhaku

Santeri Martikainen

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 18. tammikuuta 2017

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		18. tammikuuta 2017	19
Tiivistelmä — Referat — Abstract			
<p>Reitinhaku tuonee useimmille ensimmäisenä mieleen erilaiset karttasovellukset, jotka auttavat löytämään haluttuun osoitteeseen. Käyttökohteita on kuitenkin monella muullakin saralla, kuten esimerkiksi robottien liikeratojen ohjailussa, pelastustoimien suunnittelussa ja tietokonepeleissä. Tarkoitusta varten on kehitetty lukuisia algoritmeja, joiden soveltuvuus ongelman ratkaisuun riippuu pitkälti toimintaympäristöstä ja reitinhaulle asetetuista reunaehdoista.</p> <p><i>Rinnakkaisella</i> reitinhaulla puolestaan tarkoitetaan asetelmaa, missä reittiä ei haeta vain yhdelle toimijalle. Tällöin ongelmana on myös toimijoiden välisten konfliktitilanteiden tehokas ratkaiseminen, sekä usein myös laskennan suorittaminen aikarajan puitteissa. Tässä tutkielmassa tarkastellaan siis lähinnä rinnakkaista reitinhakua, sekä etenkin siihen kehitettyjä algoritmeja.</p> <p>ACM Computing Classification System (CCS): Computing methodologies → Artificial intelligence → Control methods → Motion path planning Computing methodologies → Artificial intelligence → Planning and scheduling → Multi-agent planning</p>			
Avainsanat — Nyckelord — Keywords			
reitinhaku, multi-agent pathfinding, MAPF			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

Sisältö

1 Johdanto	1
2 Reitinhaku	1
2.1 Dijkstran algoritmi	2
2.2 A*-algoritmi	3
2.3 Hierarchical Pathfinding A* (HPA*)	5
3 Rinnakkainen reitinhaku	5
4 A*-pohjaiset rinnakkaisen reitinhaun algoritmit	7
4.1 Local Repair A* (LRA*)	8
4.2 Cooperative A* (CA*)	8
4.3 Hierarchical Cooperative A* (HCA*)	9
4.4 Windowed Hierarchical Cooperative A* (WHCA*)	10
4.5 Conflict-Based Search (CBS)	11
5 Muut rinnakkaisen reitinhaun algoritmit	11
5.1 Increasing Cost Tree Search	12
5.2 Tree-based agent swapping strategy (TASS)	13
5.3 Push and Swap ja Push and Rotate	13
6 Yhteenveto	17
Lähteet	18

1 Johdanto

Reitinhaussa on pohjimmiltaan kyse mahdollisimman suoran ja nopean polun löytämisestä kahden pisteen välillä jossakin topologiassa. Topologia, eli kenttä tai alue, voi olla reaali maailmassa, 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 sovelluskohde mikä hyvänsä, varsinaista reitinhakua tarvitaan vain mikäli reitin löytäminen jokaisella suorituskerralla olla täysin suoraviivaista ja triviaalia. Mikäli näin olisi, asia voitaisiin ratkaista yksinkertaisesti laskeamalla lyhin etäisyys mitä pitkin liikkua suoraan kohteeseen ja toimia sen mukaan. Kysymys on siis reittien löytämisestä tietyin reunaehdoin. Esimerkiksi navigaattoria käyttävä 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 yhtäikäisen 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, 3].

Esteiden kiertämisen ja mahdollisesti 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 ennalta määrättyyn 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 ennakoon, 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. Koska reitinhaulta usein vaaditaan reaaliaikaisuutta, tällainen lähestymistapa soveltuu vain verrattaen pieneen toimintaympäristöön.

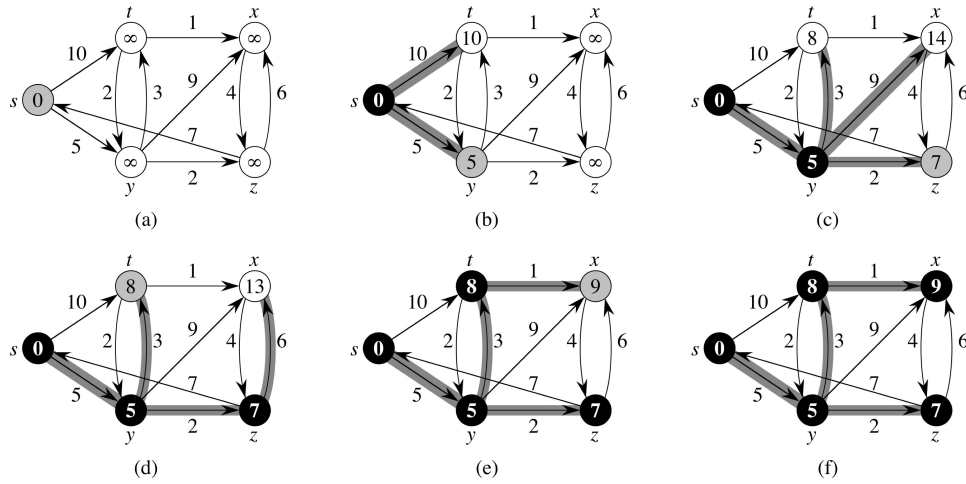
Reitinhaun standardialgoritmi on jo pitkään ollut Dijkstran algoritmiin perustuva A^* (eli A-star tai A-tähti), josta on kehitetty lukuisia eri variantteja eri toimintaympäristöjä ja tarpeita silmälläpitäen. Sen etuja on, että se pystyy varmuudella löytämään reitin kohteeseen, jos sellainen on ylipäättään 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 syvyysuuntainen-haaraus-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 lyhimmat 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 viisisolmuaisessa graafissa löydetään lyhyimmät rei-

tit 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 -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 [10, 11].



Kuva 1: Graafin läpikäynti Dijkstran algoritmilla [12].

2.2 A*-algoritmi

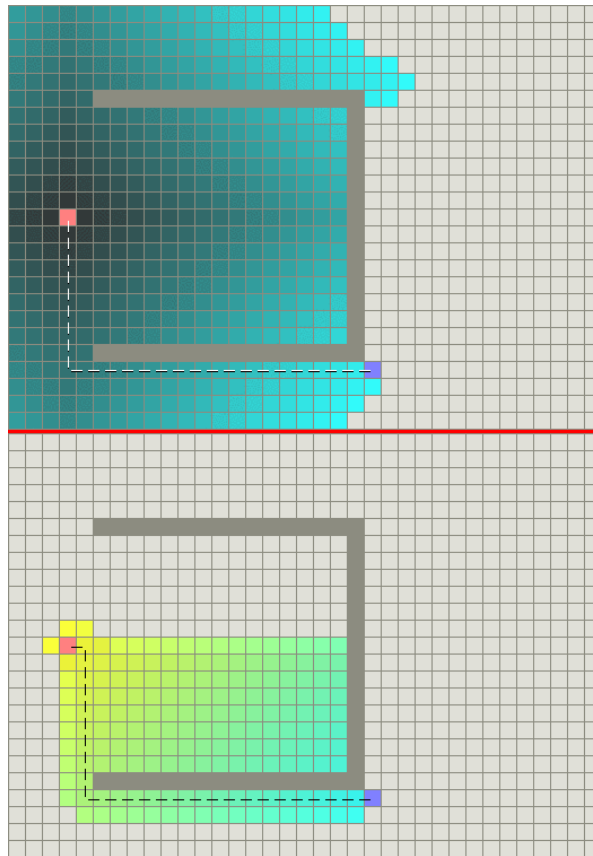
Dijkstran algoritmiin perustuva A* toteuttaa niin sanottua paras ensintyyliä, jossa jokaisen solmun tai solun kohdalla pyritään ensiksi etenemään suoraan kohti maalia. Jos tiellä on jokin este, algoritmi pyrkii kiertämään sen. Tämä tapahtuu valitsemalla tähänastisen reitin viereisistä soluista ne, joista 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.

Kaikille algoritmin käsittelemille solmuille lasketaan arvo kaavalla

$$f(n) = g(n) + h(n), \quad (1)$$

missä $g(n)$ on lyhin tunnettu reitti lähtösolmusta solmuun n ja $h(n)$ on heuristinen arvio etäisyydestä maalisolmuun [5, 13]. 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 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. Punainen neliö edustaa lähtösolmua ja sininen maalisolmua. Värisävyt kuvaavat laskennallista etäisyyttä: mitä vaaleampi väri, sitä suurempi etäisyys. Kuten kuvasta nähdään, Dijkstra käy läpi varsin suuren osan kentän solmuista, mutta löytää suorimman reitin kohteeseen.



Kuva 2: Ylempänä reitinhaku Dijkstran algoritmilla ja alempana A-algoritmilla [14].*

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.

2.3 Hierarchical Pathfinding A* (HPA*)

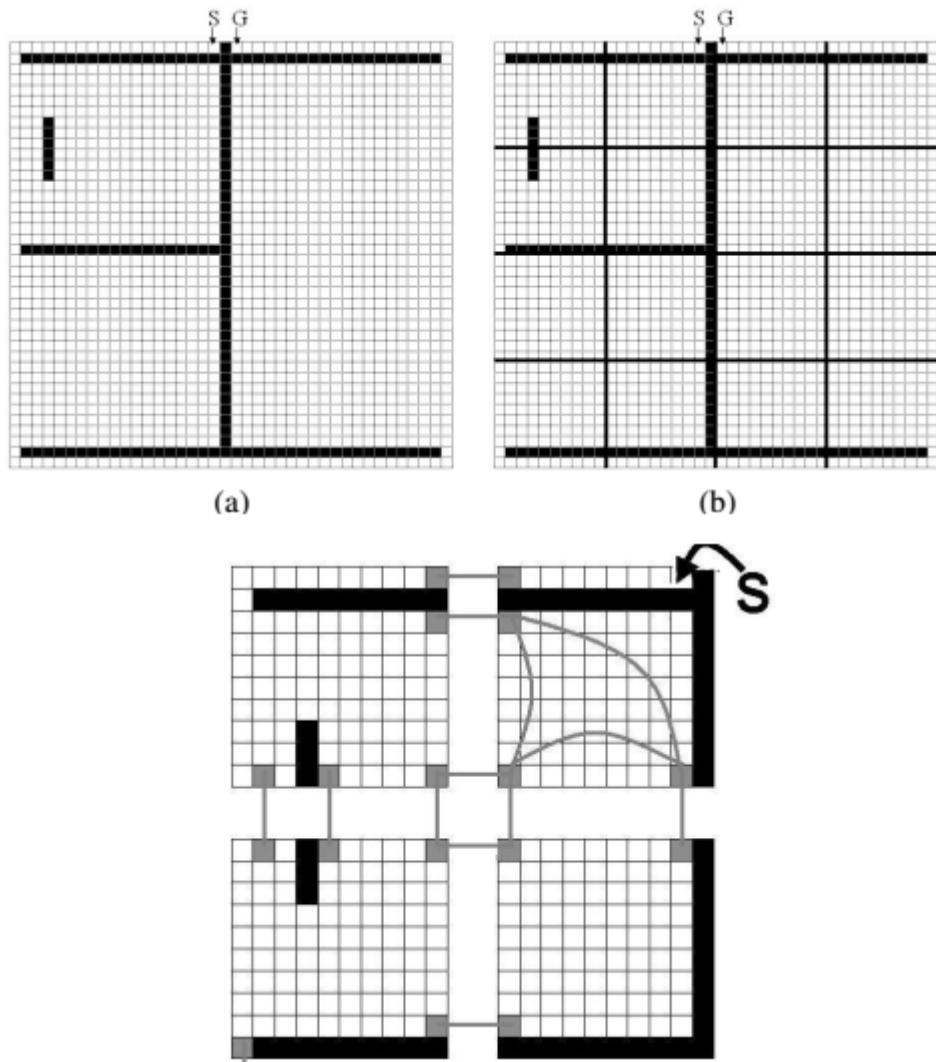
Tämä algoritmi poikkeaa A*:stä siinä, että reitinhaun apuna käytetään yhtä tai useampaa abstraktiotasoa. HPA*:n [6] kehittäjät Botea ja Müller käyttävät autovertausta havainnollistamaan algoritmin toimintaa: Kaupungista toiseen ajava autoilija ei ole kiinnostunut kaikista mahdollisista yksityiskohdista matkansa varrella, vaan häntä lähinnä kiinnostaa reitti lähtöpisteestä oikeaan suuntaan vievälle valtatielle, ja kohdekaupunkiin saavuttaessa reitti valtatieltä varsinaiseen määränpäähän. Itse valtatie 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.

Kuvassa 3 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 abstraktiomalliin 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 reitinhaaku 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 reitinhaaku

Rinnakkaisessa reitinhaussa on yleensä useita, joskus jopa satoja tai tuhansia, eri toimijoita. Tällainen monen toimijan rinnakkainen reitinhaaku



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

(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 [15]. Näiden ongelmien ratkaisemiseen kehitetyistä algoritmeista 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 käsitelläkö toimijoita yhdistetynä (coupled) toimijana, vai toisistaan erotettuina (decoupled) toimijoina. Erotetussa menetelmässä reitit lasketaan jokaiselle toimijalle erikseen ja mahdolliset yhteentörmäykset ratkaistaan valitulla menettelyllä sikäli kuin niitä tapahtuu. Tämä menetelmä on tyypillinen tilanteille, joissa toimijoita on verrattaen paljon [16]. 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 lukumäärää naapurisolmuja, joihin jostakin graafin solmusta voidaan liikkua. Jos meillä on graafi, jonka haarautuvuus on b , niin erotetussa menetelmässä jokaisella toimijalla on $O(b)$ 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 $O((b)^k)$. Tämä hidastaa reittien laskemista [16].

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

4 A*-pohjaiset rinnakkaisen reitinhaun algoritmit

Tässä kappaleessa käydään läpi muutamia algoritmeja, joissa A* on kiinteänä osana rakennetta. Kyse ei siis ole A*-varianteista, joissa A*:n toimintaa olisi vain mukautettu joltain toimintaympäristöstä silmälläpitäen, vaan laajemmista algoritmeista jotka jossain toimintansa osana käyttävät A*:ä reittien muodostamisessa.

4.1 Local Repair A* (LRA*)

LRA* [17] 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. Suoritusjärjestys voi olla erikseen määriteltä, tai täysin satunnainen.

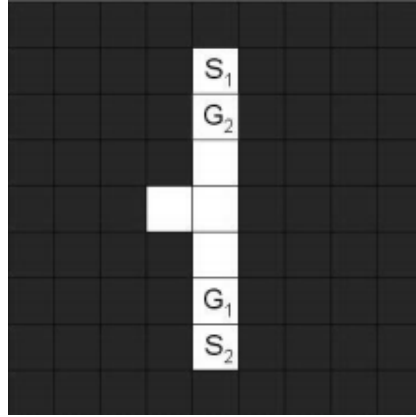
Syklit ovat tässä menetelmässä sekä mahdollisia että yleisiä, joten ongelmaa on pyritty ratkaisemaan lisäämällä niin kutsuttua kohinaa etäisyysheuristiikkaan joka kerta kun yhteentörmäys havaitaan [17]. Jos siis jatkuvasti kohdetaan esteitä jossakin solmussa tai jollakin alueella, niin algoritmin laskeman 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. LRA* toimii heikosti sellaisessa topologiassa, jossa on kapeita käytäviä ja lukuisia toimijoita suhteessa käytettävissä olevaan tilaan.

4.2 Cooperative A* (CA*)

CA* [17] pyrkii estämään yhteentörmäykset ennalta ottamalla aikaulottuvuuden huomioon reittejä suunniteltaessa. Jokaiselle toimijalle lasketaan reitti A*-algoritmillä ja suunnitellun reitin solut merkitään taulukkoon, esimerkiksi kolmiulotteiseen hajautustauluun, jossa kaksi ensimmäistä alkioita 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 jomankumman 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.



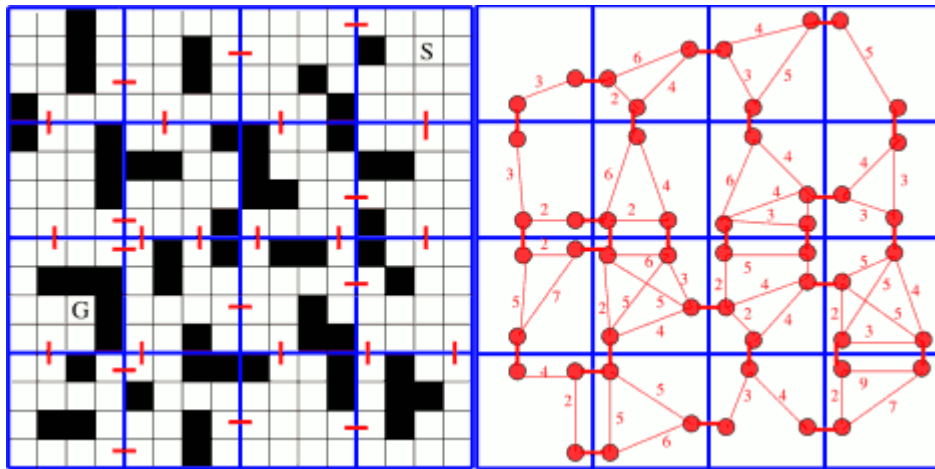
Kuva 4: Esimerkki kohtaamis-ongelmasta, jossa toimijat S_1 ja S_2 voivat päästä maalisolmuihin G_1 ja G_2 vain jos jompikumpi väistää sivulle [17].

4.3 Hierarchical Cooperative A* (HCA*)

R.C.Holte ja kumppanit esittelivät vuonna 1996 hierarkkisen A*-reitinetsintäalgoritmin HA* [18] ratkaisemaan CA*-algoritmin ongelmia. Siinä varsinaisen 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 liikumalla siinä tasolta toiselle sen mukaan miten tarkkaa jakoa esimerkiksi esteiden ympärillä tarvitaan, ja käyttämällä sen antamia etäisyyksiä reitinhaualgoritmin apuna. Etäisyydet kohteeseen lasketaan vain tarvittaessa [17]. [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



Kuva 5: Vasemmalla olevasta ruudukosta on muodostettu hierarkiamallin mukainen abstraktio [19].

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 [17]. 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 määrittäminen, 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 [17]. Tässä, kuten aiemmissakin yhteistoiminnallisissa reitinhakualgoritmeissa, on resurssien käytön suhteen ongelmana potentiaalisesti turhaan tehty työ.

4.4 Windowed Hierarchical Cooperative A* (WHCA*)

WHCA* eroaa HCA*:sta siinä, että abstraktiotasolla reitti lasketaan maaliin asti, mutta konkreettisen topologian tasolla reitinhaku on rajoitettu johonkin ennaltamääritellyyn syvyyteen [17, 1]. Kun reittiä on kuljettu johonkin ennaltamääritellyyn raja-arvoon asti, esimerkiksi puolet aiemmin lasketusta reitistä, lasketaan uusi osittainen reitti ja niin edelleen. ”Windowed” tar-

koittaa 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 alemmalla 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 lisäämällä siihen solmuja löysemmin kriteerein. Tämän jälkeen jokaisen solmun kohdalla tehdään uusi alemman tason haku ja toistetaan menettely tarvittaessa.

5 Muut rinnakkaisen reitinhaun algoritmit

Tässä kappaleessa käsiteltävät algoritmit edustavat uudempaa sukupolvea, eikä niiden toiminta kiinteästi riipu A*-algoritmin hyödyntämisestä. Yhteisenä nimittäjänä voitaneen pitää pyrkimystä rajoittaa resurssien käyttö minimiin tinkimättä kuitenkin tehokkuudesta.

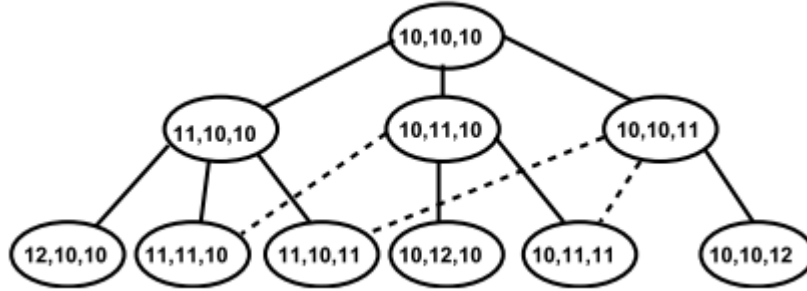
5.1 Increasing Cost Tree Search

Vuonna 2012 julkaistu ICTS-algoritmi on kaksitasoista hierarkiaa hyödyntävä algoritmi [16]. Hierarkian ylemmällä tasolla käydään läpi puuta nimeltä *increasing cost tree*. Se muodostuu solmuista, joista jokaisessa oleva taulukko $[C_1, \dots, C_k]$ 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 suurimmat 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 keskimäinen ja kolmannessa viimeinen.

ICT-puuta läpikäydessä jokaisen solmun kohdalla kutsutaan alemman tason rutiinia tarkistamaan josko solmun määrittämä reittiyhdistelmä on löytynyt. Eli aluksi kokeillaan onko juurisolmun etäisyyksillä löydettävissä reitit kaikille toimijoille, mutta jos havaitaan törmäyksiä, siirrytään puussa alaspäin. Mitä alemmas puussa siis mennään, sitä suurempi on solmuissa olevien reittien hintojen summa. Puuta käydään läpi leveyssuuntaisella haululla 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 käsittelymiseen on kehitetty oma algoritminsa, *multi-value decision diagram* (MDD). Varsinaisten reittien muodostamiseen konkreettisessa topologiassa voidaan käyttää sopivaksi katsottua yhden toimijan reitinhakualgoritmia.



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

5.2 Tree-based agent swapping strategy (TASS)

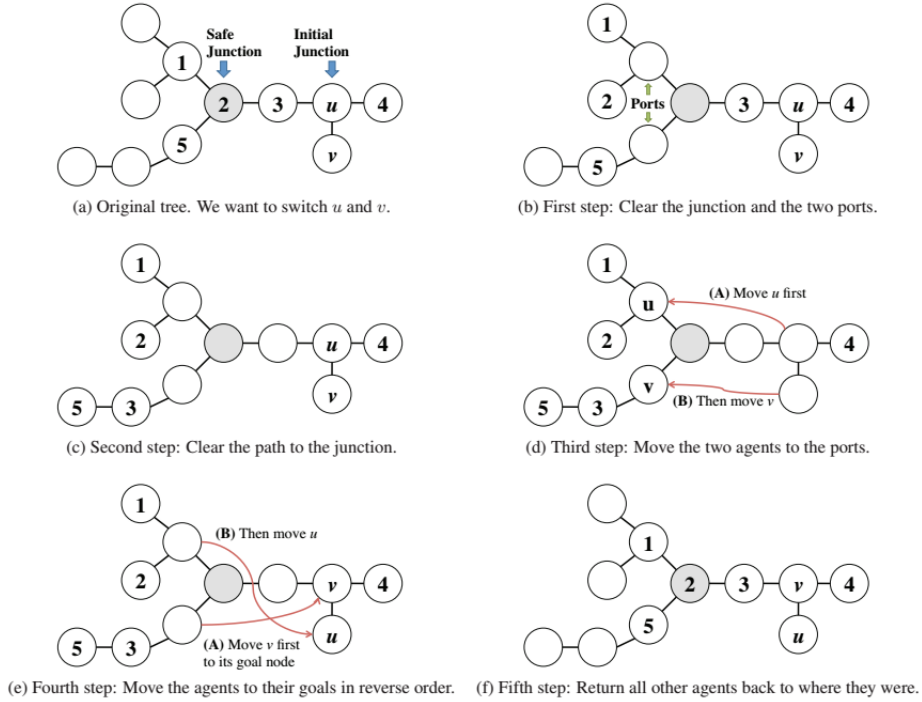
Tämä vuoden 2011 algoritmi hyödyntää puurakenteita ja olemassaolevien ei-puumaisten graafien konvertoimiseen onkin kehittäjien toimesta luotu *Graph-to-Tree Decomposition (GTD)* -algoritmeja. TASS [3] on kehitetty erityisesti tilanteisiin, joissa vapaata liikkumatilaa on hyvin vähän.

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, missä u ja v ovat vierekkäin, eikä välittömässä läheisyydessä ole vapaita solmuja joihin väistää. Tässä algoritmista paikkoja ei voi vaihtaa keskenään, 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, koska tämä keskimäärin lyhentää etäisyyttä lähimpään sellaiseen paikkaan puussa, missä uudelleenjärjestely voidaan suorittaa.

5.3 Push and Swap ja Push and Rotate

Push and Swap edustaa rinnakkaisen reitinhaun uutta sukupolvea (2011) ja on verrattaen yksinkertainen toimintaperiaatteeltaan. Nimensä mukaisesti algoritmista on kaksi keskeistä operaatiota, *Push* ja *Swap*, sekä kaksi apuoperaatiota, *Clear* ja *Resolve*. Toimijoiden suoritusjärjestys määräytyy jonkin ennaltamääritellyn prioriteettijonon perusteella [20].

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

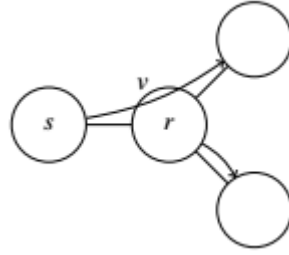


Kuva 7: Haarautuman hyväksikäyttö uudelleenjärjestämisessä TASS-algoritmilla [3].

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 työnnettyt 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:iä kutsutaan r :lle uudestaan. Jos tämä ei onnistu, suoritetaan Swap t :lle [21].

Tässä algoritmissa vaivaa kuitenkin joukko ongelmia: Niin kutsuttujen *monikulmiograafien* (polygon graph) tapauksessa algoritmi voi jäädä silmukkaan.



Kuva 8: Push and Swap-algoritmi: Alkioiden r ja s paikat vaihdetaan Swap-operaatiolla [20].

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.

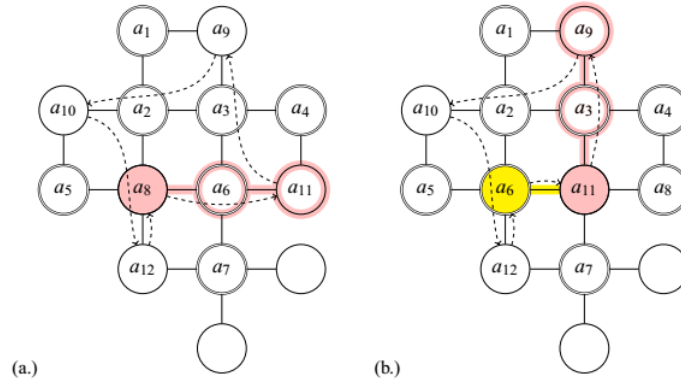
Push and Swap-algoritmin ongelmat on pyritty ratkaisemaan vuonna 2013 julkaistulla Push and Rotate-algoritmillä [21]. Siinä monikulmiograafien ongelma on ratkaistu priorisoimalla reitinhaussa sellaisia reittejä, joiden varrella ei ole lainkaan jo maaliin päässeitä toimijoita. Tämä lähestymistapa välttää niin sanotun *elolukon* (livelock), missä kaksi toimijaa vaihtaa keskenään paikkaa vuorotellen ilman, että reitinhaku koskaan ratkeaa. Myös Clear-operaation toimintaa on laajennettu ottamaan huomioon laajempi siirtomahdollisuuksien kirjo. Niin kutsuttu *kannasongelma* on ratkaistu jakamalla kokonaisuus eri tavalla priorisoituihin aliongelmiin: eri toimijaryhmät saavat eritasoiset prioriteetit, jolloin korkeamman prioriteetit ryhmät pääsevät etenemään maaliin ikäänkuin etuajo-oikeudella.

Ahtaissa ja ruuhkaisissa tilanteissa Swap-operaation kutsuma Resolve saa aikaan lisää Resolve-operaatioita toimijoiden peräjälkeen pyrkiessä hakeutumaan uusiin asemiin, mikä voi johtaa virhetilanteeseen Resolve:n sijoittaessa toimijoita oman toimintaperiaatteensa vastaisesti. Käytännössä tämä näkyy tilanteina, missä kahden toimijan piti päätyä operaation tuloksena vierekkäin, mutta rekursiiviset Resolve-kutsut jättivät nämä lopulta erilleen.

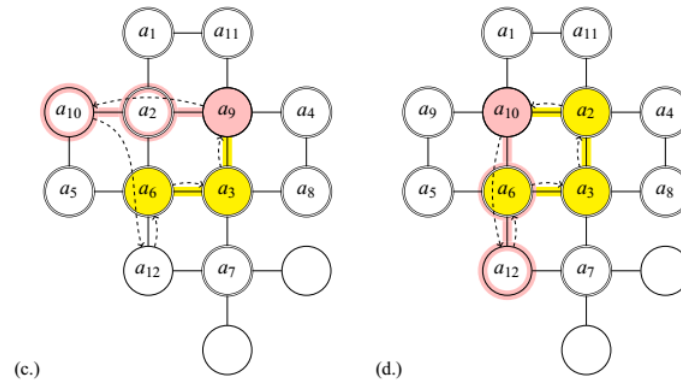
Kuvassa 9 on havainnollistettu algoritmin toimintaa. Solmujen mahdollinen kaksinkertainen kehä tarkoittaa, että siinä oleva toimija on jo löytänyt maaliinsa. Katkoviivalliset nuolet osoittavat mihin solmuihin eri toimijat pyrkivät, vaaleanpunainen väri tarkoittaa Plan-apuoperaatiota ja keltainen väri

Resolve-operaatiota. Plan suorittaa kaksi Swap-operaatiota siirtäen toimijan a_8 graafin oikeaan laitaan. Tieltä piti siirtää toimijat a_6 , joka ei ole enää maalisolmussa, sekä a_{11} . Jotta a_6 saadaan takaisin oikealle paikalleen, täytyy a_{11} siirtää pois tieltä. Tämä tapahtuu siirtämällä a_{11} kohti omaa maalisolmuaan. Tämä johtaa a_3 :n ja a_9 siirtämiseen pois edestä.

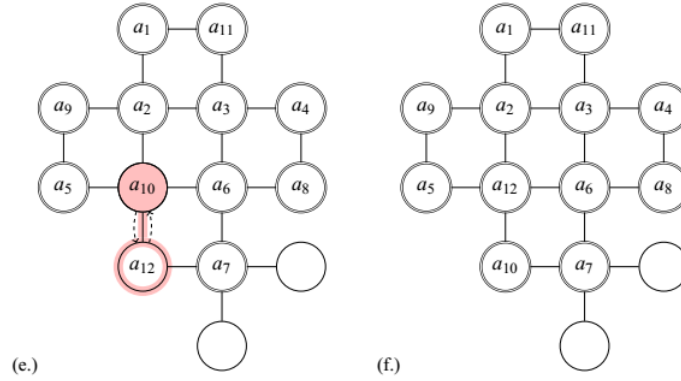
Kuvassa 10 puolestaan nähdään, että a_{11} on nyt maalissa, mutta keltaisella merkityjä Resolve-operaation odottajia on nyt kaksi, a_6 ja a_3 . Siirtämällä a_9 maalisolmuunsa, a_2 siirtyy pois maalista, ja graafissa on selvä sykli (a_2 , a_3 , a_6 ja a_{10}). Tilanne ratkaistaan kiertämällä (rotate) toimijoita graafissa askel vastapäivään, jonka tulokset nähdään kuvassa 11. Jäljellä on enää kaksi toimijaa, a_{10} ja a_{12} . Koska nämä ovat vierekkäin ja pitävät hallussaan toistensa maalisolmuja, voidaan Swap-operaatiolla vaihtaa niiden paikat ja reitinhaku on valmis.



Kuva 9: Plan-operaatio merkitty punaisella, Resolve keltaisella. Ensimmäinen siirto siirtää a_8 kaksi solmua oikealle, ja sitten a_{11} kaksi solmua ylös [22].



Kuva 10: Resolve-operaatiota (keltaisella) odottavat toimijat muodostavat a_9 :n siirtämisen jälkeen syklin yhdessä a_{10} :n kanssa [22].



Kuva 11: Rotate-operationin jäljiltä jäljelle jää enää $a_{10:n}$ ja $a_{12:n}$ paikkojen vaihtaminen [22].

6 Yhteenveto

Reitinhaun toteuttaminen pysyy edelleen avoimena ongelmana 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 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 [20].

Ohjelmoijan vastuulle jääkin pohtia topologiaa, toimijoiden lukumäärää ja laskentanopeuden tarvetta sopivaa reitinhakualgoritmia valitessaan. Kaikkiin tilanteisiin sopivaa yleisalgoritmia ei toistaiseksi ole olemassa, eikä sellaista kenties ole mahdollista toteuttaakaan.

Lähteet

- [1] Botea, A., Bouzy, B., Buro, M., Bauckhage, C. ja Nau, D.: *Pathfinding in games*. Dagstuhl Follow-Ups, 6, 2013.
- [2] Algfoor, Z, Sunar, M ja Kolivand, H.: *A comprehensive study on path-finding techniques for robotics and video games*. International Journal of Computer Games Technology, 2015:7, 2015.
- [3] Khorshid, M., Holte, R. ja Sturtevant, N.: *A polynomial-time algorithm for non-optimal multi-agent pathfinding*. Teoksessa *Fourth Annual Symposium on Combinatorial Search*, 2011.
- [4] Cui, X. ja Shi, H.: *A*-based pathfinding in modern computer games*. International Journal of Computer Science and Network Security, 11:125–130, 2011.
- [5] Sharon, G., Stern, R., Felner, A. ja Sturtevant, N.: *Conflict-based search for optimal multi-agent pathfinding*. Artificial Intelligence, 219:40–66, 2015.
- [6] Botea, A., Müller, M. ja Schaeffer, J.: *Near optimal hierarchical path-finding*. Journal of game development, 1:7–28, 2004.
- [7] Felner, A., Korf, R. ja Hanan, S.: *Additive pattern database heuristics*. Journal of Artificial Intelligence Research, 22:279–318, 2004.
- [8] Dijkstra, E.: *A note on two problems in connexion with graphs*. Numerische Mathematik, 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] Moy, J.: *OSPF version 2*. 1997.
- [11] Oran, D.: *OSI IS-IS intra-domain routing protocol*. 1990.
- [12] 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 .
- [13] Stout, B.: *Smart moves: Intelligent pathfinding*. Game developer magazine, 10:28–35, 1996.
- [14] University, Stanford: *Introduction to A**, 12.1.2017. <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>, vierailtu 2017-01-12 .

- [15] Erdem, E., Kisa, D., Öztok, U. ja Schueller, P.: *A General Formal Framework for Pathfinding Problems with Multiple Agents*. Teoksessa *Association for the Advancement of Artificial Intelligence*, 2013.
- [16] Sharon, G., Stern, R., Goldenberg, M. ja Felner, A.: *The increasing cost tree search for optimal multi-agent pathfinding*. *Artificial Intelligence*, 195:470–495, 2013.
- [17] Silver, D.: *Cooperative Pathfinding*. *Artificial Intelligence and Interactive Digital Entertainment Conference*, 1:117–122, 2005.
- [18] Holte, R., Perez, M., Zimmer, R. ja MacDonald, A.: *Hierarchical A*: Searching abstraction hierarchies efficiently*. Teoksessa *Association for the Advancement of Artificial Intelligence/Innovative Applications of Artificial Intelligence Conference*, nide 1, sivut 530–535. Citeseer, 1996.
- [19] Duc, L., Sidhu, A. ja Chaudhari, N.: *Hierarchical pathfinding and ai-based learning approach in strategy game design*. *International Journal of Computer Games Technology*, 2008:3, 2008.
- [20] Luna, R. ja Bekris, K.: *Push and swap: Fast cooperative path-finding with completeness guarantees*. Teoksessa *International Joint Conference on Artificial Intelligence*, sivut 294–300, 2011.
- [21] De Wilde, B., Mors, A. ter ja Witteveen, C.: *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.
- [22] De Wilde, B.: *Cooperative multi-agent path planning*. väitöskirja, TU Delft, Delft University of Technology, 2012.