

Rinnakkainen reitinhaku

Santeri Martikainen

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 12. tammikuuta 2017

[illegible]

Sisältö

1 Johdanto	1
2 Reitinhaku	1
2.1 Dijkstran algoritmi	2
2.2 A*-algoritmi	3
2.3 Hierarchical Pathfinding A* (HPA*)	4
3 Rinnakkainen reitinhaku	8
4 A*-pohjaiset rinnakkaisen reitinhaun algoritmit	8
4.1 Local Repair A* (LRA*)	9
4.2 Cooperative A* (CA*)	9
4.3 Hierarchical Cooperative A* (HCA*)	9
4.4 Windowed Hierarchical Cooperative A* (WHCA*)	11
4.5 Conflict-Based Search (CBS)	12
5 Muut rinnakkaisen reitinhaun algoritmit	12
5.1 Increasing Cost Tree Search	12
5.2 Tree-based agent swapping strategy (TASS)	15
6 Yhteenveto	15
Lähteet	17

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 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 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]. Topologia voidaan toisinaan mallintaa myös puurakenteena [3].

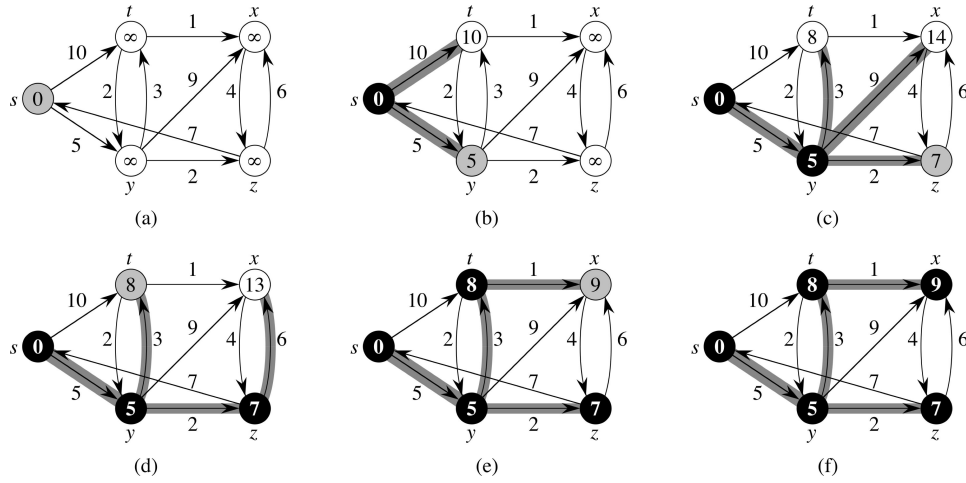
Edellämainittujen esteiden ja muuttuvan toimintaympäristön lisäksi reitinhaun keskeisiä ongelmia on resurssien käyttö. Mitä laajemmassa topologiassa reitinhaku suoritetaan, sitä enemmän laskentaresursseja (muistia, prosessoriaikaa) siihen menee. Joissakin tilanteissa voidaan tyytyä osittaiseen reitinhakuun, 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 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 syvyys-suuntainen-haarautuva-rajattu (depth-first branch-and-bound, DFBnB) [7]. Resurssitarpeiden kurssapitämiseksi 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 viisisolmuksessa 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 etäisyydsarvot. 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

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

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

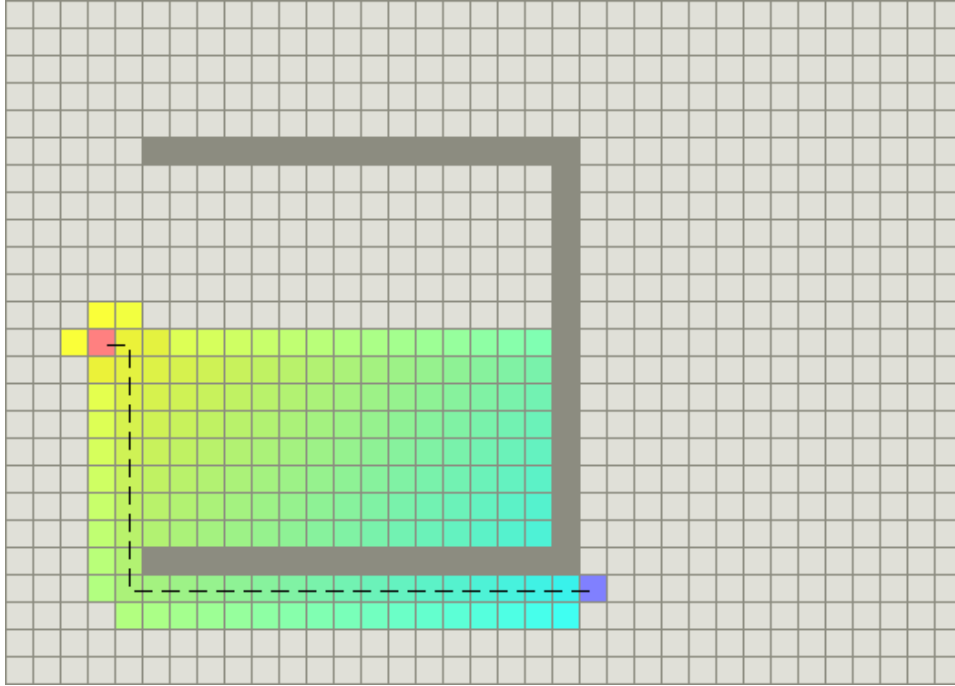
missä $g(n)$ on lyhin tunnettu reitti lähtösolmusta solmuun n ja $h(n)$ on heuristinen arvio etäisyydestä maalisolmuun [5, 10]. Näin jokaisen läpikäydyn solmun kohdalla tiedetään sille lasketusta arvosta kuinka suoralla reitillä kohteeseen ollaan. Solmuille voidaan myös asettaa edellämainittuun kaavaan 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

Algorithm 1 A*-algorithm in pseudocode

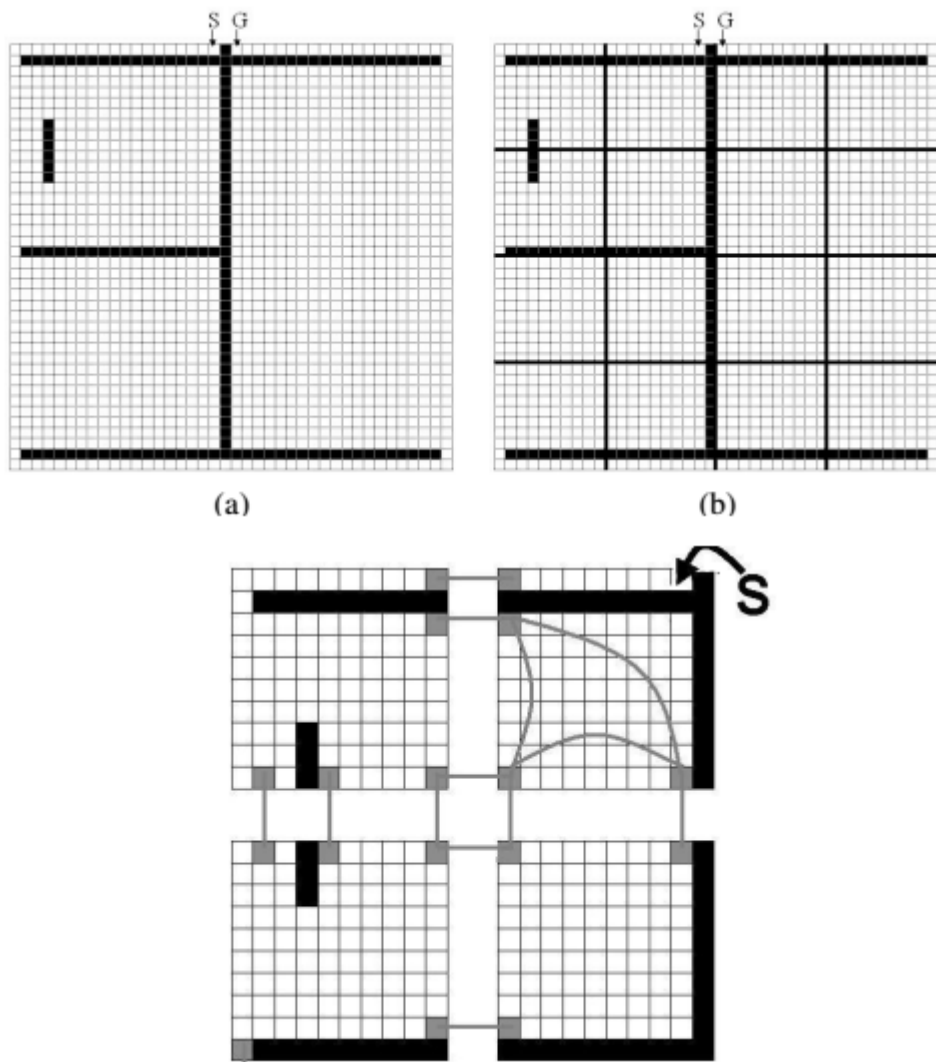
```
1: procedure ASTAR(startnode, goalnode)
2:   initialize the open list
3:   initialize the closed list
4:   put the starting node on the open list (you can leave its f at zero)

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
```



Kuva 3: Esteen kiertäminen A^* -algoritmilla

sa. 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: Ensimmäisessä 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 abstraktiotasoa, jolloin jokainen uusi abstraktiotaso on edellistä karkeampi esitys kentän topologiasta.



Kuva 4: Ylärivillä 40x40 -ruudukon jakaminen 16:een klusteriin HPA*-algoritmin esiprosessoinnissa, alla vasemman yläneljänneksen klustereiden siirtymät [6].

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 [11]. 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 [12]. 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 $O(b + 1)$ 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 [12].

Kuten edellä todettiin, useimmat rinnakkaisen reitinhaun ratkaisemiseen kehitetyt menetelmät ovat A*-pohjaisia. A*-algoritmi 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

kuvaukset:

Pattern Databases

Independence detection (ID)

Operator decomposition (OD)
Enhanced partial expansion A* (EPEA*)

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 [13]. 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.

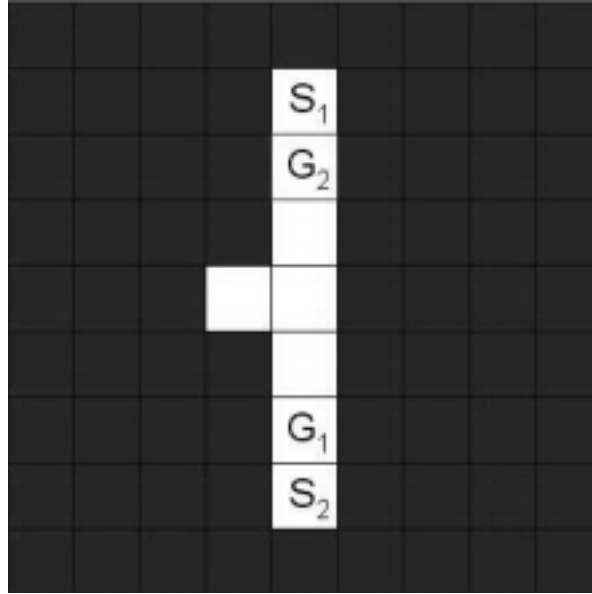
4.2 Cooperative A* (CA*)

CA* 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 5 nähdään tilanne, missä toimijat S1 ja S2 pyrkivät vastaavasti maaleihin G1 ja G2. Intuiitiivisesti 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 algoritmilla ei ole tällaista toiminnallisuutta.

4.3 Hierarchical Cooperative A* (HCA*)

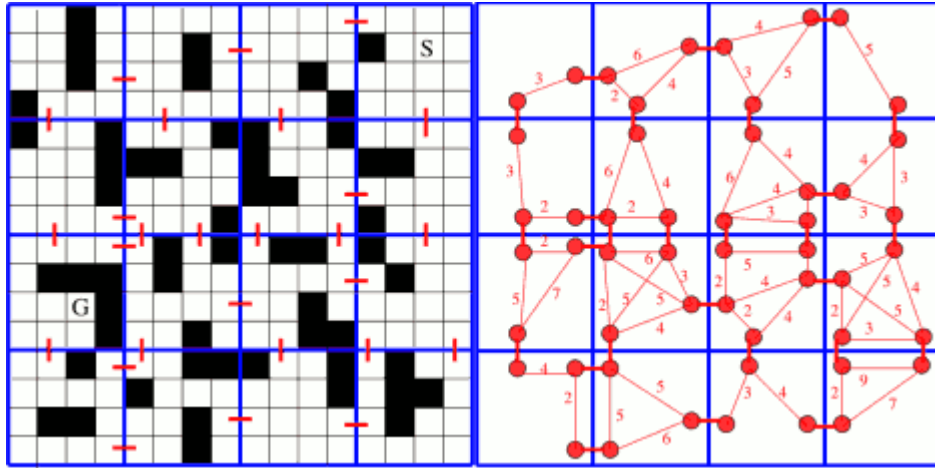
R.C.Holte ja kumppanit esittelivät vuonna 1996 hierarkkisen A*-reitinetsintäalgoritmin HA* [14] ratkaisemaan CA*-algoritmin ongelmia. Siinä varsina-



Kuva 5: Esimerkki kohtaamis-ongelmasta, jossa toimijat S_1 ja S_2 voivat päästä maalisolmuihin G_1 ja G_2 vain jos jompikumpi väistää sivulle.

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 [13]. [aukaisu]

HCA* eli hierarkkinen yhteistoiminnallinen A*-reitinetäälgoritmi puolestaan käyttää yksinkertaistettua hierarkiaa, joka koostuu vain yhdestä topologian kaksikulotteisesta 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.



Kuva 6: Vasemmalla olevasta ruudukosta on muodostettu hierarkiamallin mukainen abstraktio.

Kun reitti on laskettava uudelleen esimerkiksi jonkun toisen toimijan tullessa eteen, pyritään säästämään resursseja käyttämällä käännteistä 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 [13]. 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 [13]. 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 ast, mutta konkreettisen topologian tasolla reitinhaku on rajoitettu johonkin ennaltamääritettyyn syvyyteen [13, 1]. Kun reittiä on kuljettu johonkin en-

naltamääritelyyn 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 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 vastaavasti laajentamalla rajoituspuuta lisäämällä siihen solmuja. Tämän jälkeen tehdään uusi alemman tason haku ja toistetaan edellinen vaihe tarvittaessa.

5 Muut rinnakkaisen reitinhaun algoritmit

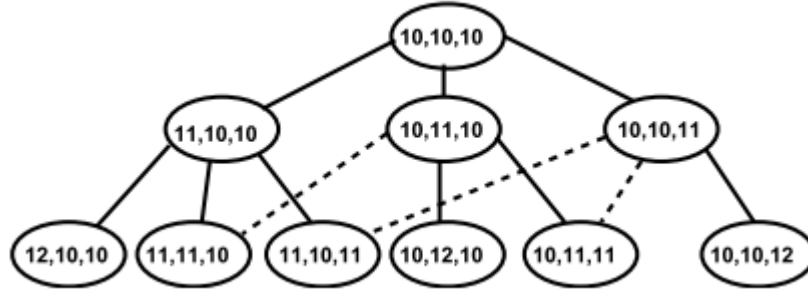
5.1 Increasing Cost Tree Search

Vuonna 2012 julkaistu ICTS-algoritmi on kaksitasoista hierarkiaa hyödyntävä algoritmi [12]. 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

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:      $N \leftarrow$  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:      $C \leftarrow$  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:       $A \leftarrow$  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  $\leftarrow N$ .constraints +  $(a_i, s, t)$ 
26:    $A$ .solution  $\leftarrow N$ .solution
27:   Update  $A$ .solution by invoking low level( $a_i$ )
28:    $A$ .cost  $\leftarrow SIC(A$ .solution)
29:   return  $A$ 
```



Kuva 7: Increasing Cost Tree kolmella toimijalla [5].

generoitu luku. Toisin sanoen toimijan a_i reittikustannus löytyy taulukon kohdasta C_i .

Kuvassa 7 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 ja jos ei, siirrytään puussa alaspäin. Mitä alemmas puussa siis mennään, sitä suurempi on solmuissa olevien reittien hintojen summa. Puuta käydään läpi leveyssuuntaisella haulilla 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 algoritminsä, *multi-value decision diagram* (MDD). Varsinaisten reittien muodostamiseen konkreettisessa topologiassa voidaan käyttää sopivaksi katsottua yhden toimijan reitinhakualgoritmia.

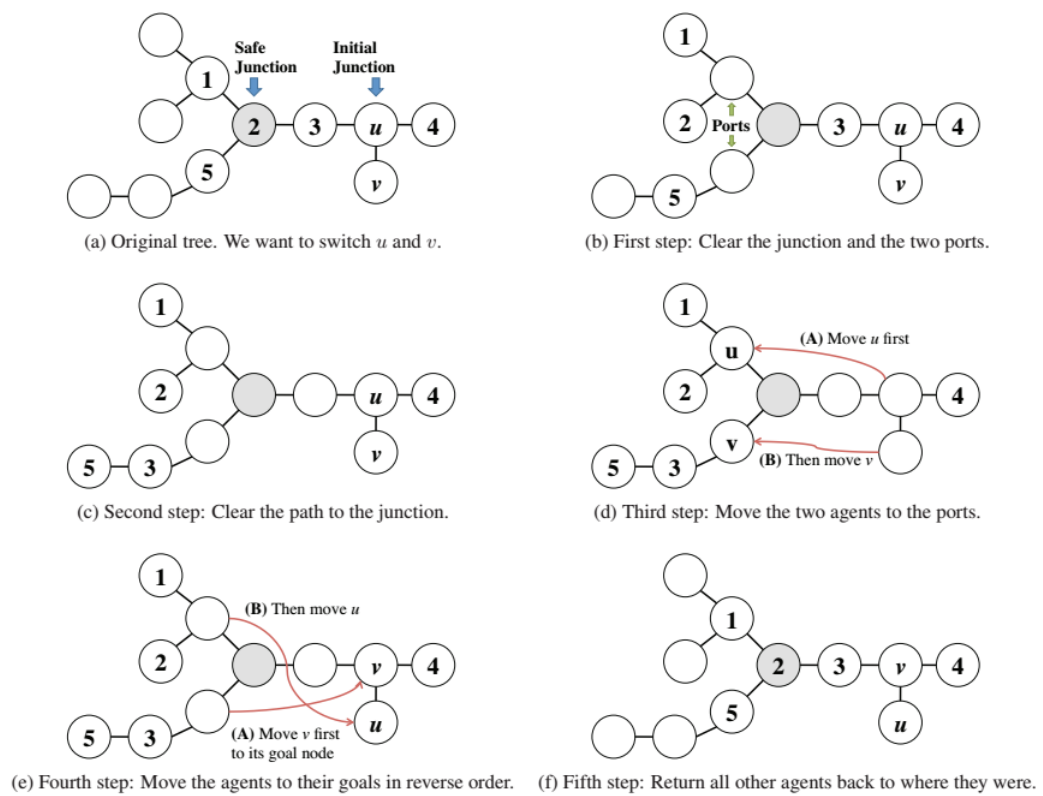
Algorithm 3 Increasing Cost Tree

```
1: Input:  $(k, n)$  MAPF
2: Build the root of the ICT
3: foreach ICT node in a breadth-first manner do
4:   foreach agent  $a_i$  do
5:     Build the corresponding  $MDD_i$ 
6:   [ //optional
7:     foreach pair (triple) of agents do
8:       Perform node-pruning
9:       if node-pruning failed then
10:        Break //Conflict found. Next ICT node
11:   ]
12:   Search the  $k$ -agent MDD search space // low-level search
13:   if goal node was found then
14:     return Solution
```

5.2 Tree-based agent swapping strategy (TASS)

Tämä algoritmi hyödyntää graafien sijaan puurakenteita ja olemassaolevien graafien konvertoimiseen onkin luotu *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 8. 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].

6 Yhteenveto



Kuva 8: Haarautuman hyväksikäyttö uudelleenjärjestämisessä TASS-algoritmilla [3].

Lähteet

- [1] Adi Botea, Bruno Bouzy, Michael Buro, Christian Bauckhage, and Dana Nau. Pathfinding in games. *Dagstuhl Follow-Ups*, 6, 2013.
- [2] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015:7, 2015.
- [3] Mokhtar M Khorshid, Robert C Holte, and Nathan R Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Fourth Annual Symposium on Combinatorial Search*, 2011.
- [4] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.
- [5] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.
- [6] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 1(1):7–28, 2004.
- [7] Ariel Felner, Richard E Korf, and Sarit Hanan. Additive pattern database heuristics. *J. Artif. Intell. Res.(JAIR)*, 22:279–318, 2004.
- [8] EW Dijkstra. Numerische mathematik. *Numerische Mathematik*, 1(1):269–271, 1959.
- [9] S Skiena. Dijkstra’s algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Reading, MA: Addison-Wesley, pages 225–227, 1990.
- [10] Bryan Stout. Smart moves: Intelligent pathfinding. *Game developer magazine Vol. 10*, pages 28–35, 1996.
- [11] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schueller. A general formal framework for pathfinding problems with multiple agents. In *AAAI*, 2013.
- [12] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013.
- [13] David Silver. Cooperative pathfinding. *AIIDE*, 1:117–122, 2005.

- [14] Robert C Holte, Maria B Perez, Robert M Zimmer, and Alan J MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI, Vol. 1*, pages 530–535. Citeseer, 1996.