Rinnakkainen reitinhaku videopeleissä

Santeri Martikainen

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 29. marraskuuta 2016

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 videopeleissä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	29. marraskuuta 2016	6

Tiivistelmä — Referat — Abstract

Tiivistelmä.

Avainsanat — Nyckelord — Keywords
avainsana 1, avainsana 2, avainsana 3

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Tässä aineessa keskitytään tarkastelemaan reitinhakua tietokonepelien saralla ja erityisesti algoritmeja, jotka on suunniteltu tilanteeseen, jossa monta eri toimijaa (agent) jakaa saman tilan ja joille kaikille on löydettävä reitti kohteeseensa joillakin reunaehdoilla.

Reitinhaussa on pohjimmiltaan kyse mahdollisimman suoran ja nopean polun löytämisestä kahden pisteen välillä jossakin topologiassa. Pelimaailman topologia, eli siis kenttä tai alue, jolla pelaaja ja mahdolliset ei-pelaajahahmot voivat liikkua, voi olla ulkoasultaan hyvin abstrakti ja pelkistetty, tai toisaalta vaikuttaa hyvinkin realistiselta maastolta puineen, vesistöineen, rakennuksineen ja niin edelleen. Kenttä voidaan muodostaa soluista, käytännössä monikulmioista, jotka ovat joko kolmioita, neliöitä tai kuusikulmioita. Edellämainitut muodot ovat ainoat monikulmiot, joita yhdistelemällä voidaan jokin alue kattaa ilman väliin jääviä aukkoja.

Rakenteellisesti kenttä kuitenkin tyypillisesti muodostuu verkosta `G=(V, E)`, missä `V` (vertex) edustaa solmuja ja `E` (edge) näitä yhdistäviä kaaria. Liikkuminen voi tapahtua joko solmusta toiseen, tai sitten nämä voivat edustaa kiintopisteitä joiden välittömässä läheisyydessä voidaan liikkua ilman erillistä reitinhakua. Solmut voivat myös toimia esteinä, jolloin niihin liikkuminen on estetty. Tämä voi myös olla vain väliaikainen tila esimerkiksi jonkun toisen toimijan ollessa liikkumisen tiellä. Molemmissa tapauksissa täytyy luonnollisesti löytää reitti esteen ympäri. Topologia voi siis olla dynaaminen, eli sen rakenne saattaa muuttua, mikä asettaa omat haasteensa reitinhaulle [1].

2 Reitinhaku

Realismia tavoittelevassa pelimaailmassa esteitä ovat luonnollisesti kaikki sellaiset objektit, jotka olisivat reaalimaailmassakin esteitä, kuten puut tai seinät. Siten pelissä liikkuva hahmo ei usein voi kulkea kahden pisteen välillä suoraan, vaan esteiden ympäri on löydettävä jokin reitti, jonka löytämiseen sovelletaan reitinetsintäalgoritmeja.

Reitinetsintä kokonaisuutena jakaantuu kahteen vaiheeseen: Toimintaympäristöstä muodostetaan ensin yksinkertaistettu malli, minkä jälkeen sitä käydään läpi jollakin algoritmilla halutun reitin löytämiseksi. Yksi tunnetuimmista tällaisista algoritmeista on nimeltään A* (eli A-star tai A-tähti), josta on kehitetty lukuisia eri variantteja eri toimintaympäristöjä silmälläpitäen [2].

2.1 Dijkstran reitinetsintäalgoritmi

2.2 A*-algoritmi

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 pyrkimällä viereisiin soluihin ja sieltä jälleen maalia kohti kunnes joko päästään kohteeseen, tai reittiä ei voida löytää. Algoritmin läpikäymille solmuille lasketaan arvo kaavalla

```
f(n) = g(n)+h(n)
```

missä `g(n)` on lyhin tunnettu reitti lähtösolmusta solmuun `n` ja `h(n)` on heuristinen arvio etäisyydestä maalisolmuun [2, 3]. 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.

[muotoilu, lähde, kieli/käännös, tyyli?]

```
// A*-algoritmi pseudokoodina

initialize the open list
initialize the closed list
put the starting node on the open list (you can leave its f at zero)

while the open list is not empty
    find the node with the least f on the open list, call it "q"
    pop q off the open list
    generate q's 8 successors and set their parents to q
    for each successor
       if successor is the goal, stop the search
       successor.g = q.g + distance between successor and q
       successor.h = distance from goal to successor
       successor.f = successor.g + successor.h

       if a node with the same position as successor is in the OPEN
       list
          which has a lower f than successor, skip this successor
       if a node with the same position as successor is in the CLOSED
       list
          which has a lower f than successor, skip this successor
       otherwise, add the node to the open list
    end
    push q on the closed list
end
```

3 Rinnakkainen reitinhaku

Videopeleissä on yleensä useita, jopa satoja tai tuhansia, eri toimijoita, joille on löydettävä reitit kohteisiinsa. Tällainen monen toimijan rinnakkainen reitinhaku (multi-agent pathfinding, MAPF) tuo yksittäisen toimijan reitinhakuun verrattuna uusia ongelmia. Sallitaanko reittien risteäminen, voiko kaksi tai useampi toimija olla samaan aikaan samassa paikassa ja tuleeko liikkumista porrastaa odottamalla että reitti edessä vapautuu [4]. Näiden ongelmien ratkaisemiseen on kehitetty useita eri algoritmeja, joista useimmat hyödyntävät A*-algoritmia [2].

3.1 Local Repair A*

LRA* on itse asiassa 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 [5]. 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 kun jokaisen yhteentörmäyksen yhteydessä reitti pitää laskea uudestaan.

3.2 Cooperative A*

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 1 nähdään tilanne, missä toimijat S1 ja S2 pyrkivät vastaavasti maaleihin G1 ja G2. Vaikka intuitiivisesti nähdään, että jommankumman toimijan olisi mahdollista joko väistää sivulle

tai vaihtaa paikkoja törmätessään ja siten ratkaista kohtaamisongelma, ei Cooperative A* pysty tällaiseen ratkaisuun, vaan toimijat eivät koskaan pääse maaliin.

3.3 Hierarchical Operative A*

R.C.Holte ja kummpanit esittelivät vuonna 1996 hierarkkisen A*-reitinetsintäalgoritmin HA* [6] 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 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 dynaamisesti muuttuvasta ympäristöstä johtuen [5]. [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 [5]. 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 [7]. 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 [5]. Tässä, kuten aiemmissakin yhteistoiminnallisissa reitinhakualgoritmeissa, on resurssien käytön suhteen ongelmana potentiaalisesti turhaan tehty työ.

3.4 Windowed Hierarchical Operative A*

WHCA* eroaa HCA*:sta siinä, että konkreettisen topologian tasolla reittejä ei lasketa maaliin asti, vaan reitinhaku on rajoitettu johonkin ennaltamääriteltyyn syvyyteen. Jotta toimijat saadaan hakeutumaan varmasti oikeaan suuntaan, lasketaan reitti abstraktiotasolla sen sijaan maaliin asti [5, 7]. Kun reittiä on kuljettu johonkin ennaltamääriteltyyn raja-arvoon asti, kuten 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, niin että taakka reittien laskemisesta jakautuu mahdollisimman tasaisesti ajan suhteen. Kuten HCA*, myös WHCA* hyödyntää RRA*:ta ja hyödyntää edellisen ikkunan aikana tehtyä hakua, mikä luonnollisesti tarkoittaa toimijakohtaista kirjanpitoa läpikäydyistä solmuista.

3.5 Conflict Based Search

Rinnakkaisen reitinhaun optimoimiseen pyrkivä CBS-algoritmi [2] 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. Toisekseen rajoituspuun solmuihin on myös talletettu joukko alemmalta tasolta saatuja reittejä, yksi jokaista toimijaa kohti, joiden tulee olla vapaita tiedetyistä yhteentörmäyksistä. Kolmantena 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* käyttäen samalla hyväksi ylemmältä tasolta saatua tietoa siitä, milloin ja missä on odotettavissa yhteentörmäys jonkun toisen toimijan kanssa. Mikäli kaikesta huolimatta alemmalla tasolla havaitaan yhteentörmäyksiä, päivitetään ylemmän tason rajoituspuuta vastaavasti ja laajennetaan rajoituspuuta lisäämällä siihen solmuja uusin rajoittein. [selvennys, solmujen lapset]

Lähteet

[1] Algfoor Z.A.; Sunar M.S.; Kolivand H. *A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games.* International Journal of Computer Games Technology Volume 2015, Article ID 736138.

[2] R.; Felner A. Sharon, G.; Stern. *Conflict-based search for optimal multi-agent pathfinding.* Artificial Intelligence, 2015.

[3] Stout B. *Smart Moves: Intelligent Pathfinding.* Game Developer Magazine, july 1997 edition.

[4] Erdem E.; et al. *A General Formal Framework for Pathfinding Problems with Multiple Agents.* AAAI, 2013.

[5] Silver D. *Cooperative Pathfinding.* In Proc. of AIIDE, 117-122, 2005.

[6] Holte R.C.; Perez M.B.; Zimmer R.M.; MacDonald A.J. *Hierarchical A*: Searching Abstraction Hierarchies Efficiently.* AAAI/IAAI, Vol. 1, 1996.

[7] Botea A.; Bouzy B.; Buro M.; Bauckhage C.; Nau D. *Pathfinding in Games.* Artifcial and Computational Intelligence in Games, 2013.