Jaakko Karhunen

Minimax ja alfa-beta-karsinta

Tietotekniikan kandidaatintutkielma

28. toukokuuta 2018

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Jaakko Karhunen

Yhteystiedot: jakubmedvedjev@gmail.com

Ohjaaja: Antti-Jussi Lakanen

Työn nimi: Minimax ja alfa-beta-karsinta

Title in English: Minimax and alpha-beta pruning

Työ: Kandidaatintutkielma

Sivumäärä: 20+0

Tiivistelmä: Vuoropohjaisia pelejä pelaavien ohjelmien pitää pystyä suunnittelemaan siirtonsa. Tutkielmassa perehdytään minmax-alrogitmin ja alfa-beta-karsinnan toimintaan, perehtymällä niistä tuotettuun kirjallisuuteen. Tutkielmassa on tarkoitus selvittää, ovatko minimax- ja alfa-beta-algoritmit tehokkaita vuoropohjaisten pelien tekoälyn päätöksenteossa. Niiden toimintaan ja rakenteeseen perehdytään shakkia esimerkkinä käyttäen. Algoritmit ovat toimivia tarkoituksessaan, ja niiden avulla tekoäly pystyy suunnittelemaan siirtonsa hyvin.

Avainsanat: minimax, alfa-beta, tekoäly, algoritmi

Abstract: Programs playing turn-based games need to be able to plan their moves. This study will examine minimax- and alpha-beta-algorithms by reading up on the literature written about them. The studys purpose is to investigate, are minimax- and alpha-beta-algorithms efficient in the decision making of turn-based games artificial intelligence. Their functions and structure is examined by using chess as an example. The algorithms are functional for this purpose, and with their help the ai can plan its moves well.

Keywords: minimax, alpha-beta, artificial intelligence, algorithm

Kuviot

Sisältö

1 Johdanto

Vuoropohjaisia pelejä pelaavat ohjelmat, kuten esimerkiksi shakkiohjelmat, tarvitsevat jonkin tavan valita tekemänsä siirrot. Tämä voitaisiin tehdä käymällä läpi mahdolliset siirrot ja valitsemalla niistä paras vaihtoehto. Kuitenkin kuka tahansa shakin pelaaja osaa sanoa, että välittömästi parhaalta vaikuttava siirto saattaa johtaa muutaman vuoron jälkeen häviöön, tai vastaavasti oman nappulan uhraaminen saattaa avata tilaisuuden voittoon. Knuthin ja Mooren (1975) mukaan ohjelman pitääkin osata valita vaihtoehdoista todennäköisimmin voittava siirto, mikä ei välttämättä ole siirto joka näyttää välittömästi parhaalta. Ohjelmalla täytyy siis olla jokin tapa suunnitella siirtonsa pidemmällä aikavälillä (Knuth & Moore 1975).

Tutkielmassa perehdytään tutkimukseen minmax- ja alfa-beta-algoritmeistä vuoropohjaisten pelien tekoälyssä. Nämä algoritmit on kehitetty auttamaan tekoälyn päätöksentekoa, ja ne toimivat tehokkaasti tässä käyttötarkoituksessa (Knuth & Moore 1975). Ne ovat tekoälyn päätöksentekoa auttamaan kehitettyjä algoritmeja, jotka ovat osoittautuneet käytännössä tehokkaiksi ja toimiviksi.

Tekoälyn, joka pelaa vuoropohjaisia pelejä, täytyisi pystyä arvioimaan siirtonsa mahdollisimman syvälle. Niiden pitäisi pystyä näkemään ainakin niin syvälle kuin ihmiset, jotka yleensä pystyvät miettimään 5-8 siirtoa eteenpäin. Parhaat ihmiset, kuten shakin suurmestaritason pelaajat, pystyvät suunnittelemaan siirtonsa yli kymmenen siirtoa eteenpäin (Schwab 2009).

Minimax algoritmi käy läpi pelin mahdoliset tilanteet ja valitsee niistä optimaalisen siirtovaihtoehdon. Monimutkaisissa peleissä siirtovaihtoehtoja on kuitenkin valtavat määrät. Minimax-algoritmin ongelmana on, että se analysoi pelipuusta myös niitä osia joita ei ole mielekästä analysoida. Sen toimintaa tehostaa alfa-beta karsinta, joka vähentää läpikäytävien tilanteiden määrää, muuttamatta kuitenkaan saatavaa tulosta. Algoritmi toimii Abramsonin (1989) mukaan parhaiten täydellisen tiedon nollasummapeleissä, joissa toisen pelaajan voitto on aina toiselle tappio, ja molempien pelaajien kaikki sallitut siirrot tiedetään (Abramson 1989). Shakkia käy-

tetään tyypillisisesti esimerkkinä algoritmin toiminnasta. Näin tehdään Fullerin, Gashnigin ja Gilloglyn (1973) mukaan, koska shakissa algoritmien toiminta ja rajoitteet tulevat hyvin esille (Fuller, Gaschnig & Gillogly 1973). Abramsonin (1989) mukaan shakki on hyvä esimerkki täydellisen tiedon nollasummapelistä (Abramson 1989).

Tutkielma koostuu johdannon lisäksi kolmesta luvusta. Luvussa 2 käydään läpi minimax-algoritmin käyttöä pelien tekoälyssä. Luvussa 3 käydään läpi alfa-beta-karsinta algoritmiä. Lopuksi luvussa 4 tehdään yhteenveto aiemmin tutkielmassa läpikäydyistä asioista.

2 Minimax

Vuoropohjaisia pelejä pelaavan tekoälyn pitää osata suunnitella siirtonsa, koska vuoropohjaisissa peleissä pinnallisesti paras mahdollinen siirto ei ole välttämättä optimaallinen siirto pitkällä aikavälillä. Tämän taikia tekoälyllä pitää olla jokin tapa, jolla se pystyy suunnittelemaan siirtonsa pitkällä aikavälillä. Tekoälyn pitää myös ottaa huomioon, että sillä on vastustaja joka yrittää omilla siirroillaan voittaa ja estää vastustajansa voiton. Knuthin ja Mooren (1975) mukaan pelissä paras siirtopolku eli joukko siirtoja on siis siirto joka tuo parhaan mahdollisen tuloksen, kun myös vastustaja tekee parhaat mahdolliset siirtonsa. Minimax on algoritmi tämän optimaalisen siirtopolun löytämiseen (Knuth & Moore 1975).

Knuthin ja Mooren (1975) mukaan minimax algoritmi pyrkii minimoimaan häviöt huonoimmassa mahdollisessa tilanteessa. Minimax tuottaa pelaajan pisteiden arvoksi pienimmän mahdollisen pistemäärän, minkä pelaaja vähintään saa, kun vastapelaaja ei tiedä pelaajan tekemiä siirtoja. Se on myös suurin pistemäärä, jonka pelaaja voi varmasti tietää saavansa, kun hän tietää vastustajansa siirrot. Toisin sanottuna se on pienin mahdollinen pistemäärä, jonka pelaaja vähintään saa pelaamalla täysin optimaalisesti (Knuth & Moore 1975).

Samankaltainen algoritmi on maximin-algoritmi, joka on suurin pistemäärä, jonka pelaaja voi varmasti saada tietämättä vastustajiensa siirtoja. Saman pelin maximin arvo on myös aina enintään sama kuin minimax, mutta nämä eivät yleensä ole sama arvo (Fuller, Gaschnig & Gillogly 1973). Maximin toimii lähes samalla tavalla kuin minimax.

Minimax algoritmi alunperin kehitettiin kahden pelaajan vuoropohjaisiin nollasummapeleihin, joista shakki on hyvä esimerkki. Sitä käytetään myös useamman pelaajan peleissä, ja se on Schwabin (2009) mukaan pääasiallinen keino, jolla klassisia strategiapelejä pelaava tekoäly suunnittelee siirtonsa (Schwab 2009). Sitä on mahdollista käyttää myös muussa päätöksenteossa. Aspreyn, Rustemin & Žakovićin (2001) mukaan sitä voidaan käyttää esimerkiksi osakekaupassa pörrsikurssien en-

nustamisen ja kemiallisten reaktioiden ennustamiseen (Asprey, Rustem & Žaković 2001).

2.1 Minimax algoritmin toiminta

Knuthin ja Mooren (1975) mukaan vuoropohjaisia pelejä voidaan kuvata joukkona tilanteita ja sääntöjä siitä, kuinka tilanteesta toiseen on sallittua siirtyä. Pelaajat siirtävät kukin vuorollaan pelinappuloitaan sääntöjen mukaan muuttaen pelin tilannetta toiseksi. Lisäksi minimax algoritmin toiminta olettaa, että säännöt sallivat vain jonkin äärellisen määrän mahdollisia siirtoja, ja että säännöt eivät salli pelin jatkua loputtomasti (Knuth & Moore 1975). Pelin tilanteista luodaan pelipuu, jonka juurisolmuna on aloitustilanne ja lapsisolmuina ovat kaikki mahdolliset tilanteet, joihin päästään pelaajan tehtyä yhden siirron. Lapsisoluille asetetaan sitten lapsisolmuiksi kaikki siirrot, jotka vastustaja voi vuorollaan tehdä. Solmujen lisäämistä jatketaan tällä tavalla rekursiivisesti kunnes kaikissa haaroissa solmuja on lisätty haluttu määrä tasoja tai tullaan tilanteeseen, jossa peli saadaan päätökseen. Puun jokainen taso kuvaa kaikkia mahdollisia siirtoja, jotka vuorossa oleva pelaaja voi vuorollaan tehdä. Tällöin saadaan aikaan pelipuu, joka sisältää kaikki mahdolliset siirrot jotka pelissä voi tehdä tietyn siirtomäärän aikana. (Fuller, Gaschnig & Gillogly 1973) Minimax algoritmi käy sitten pelipuun solmut läpi rekursiivisesti.

Minimax algoritmin toimintaa kuvaa Maschlerin, Solanin ja Zamirin (2013) mukaan lauseke:

$$v_i = max_{a_i} min_{a_{-i}} v_i(a_i, a_{-i})$$

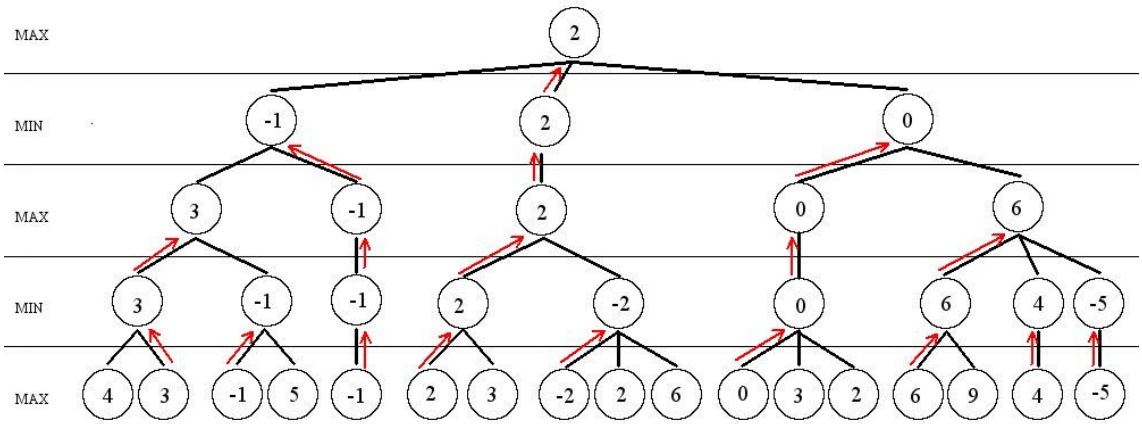
Kaavan $v_i$tarkoittaa pelaajan siirron lopullista arvoa, i on pelaaja, ja -i vastustaja, a_i on pelaajan tekemä siirto ja a_{-i} vastustajan siirto (Maschler, Solan & Zamir 2013).

Algoritmi käy puun alimmat lehtisolmut läpi ja laskee jokaiselle solmulle numeerisen arvon. Arvo hankitaan esimerkiksi shakissa niin, että jokainen laudalla oleva pelaajan oma nappula kasvattaa saatavaa pistemäärää, ja vastustajan nappula vä-

hentää sitä. Solun arvo kuvaa sitä, kuinka hyvä solun edustama tilanne on pelaajan kannalta. Suuremmat arvot ovat yleensä parempia tilanteita, joissa pelaaja on voitolla. Ne siirrot, joilla maksimoitava pelaaja voittaa, saatava arvokseen ääretön, ja siirrot joilla vastustaja voittaa miinus ääretön.

Algoritmi siirtyy sitten seuraavaan tasoon ja vertaa jokaisen tason solmun lapsisolmujen arvoa. Sitten se asettaa solmuun joko suurimman tai pienimmän lapsisolmun arvon. Suurimman, jos taso kuvaa maksimoitavan pelaajan vuoroa, ja pienimmän, jos minimoitavan. Algoritmi käy koko tason läpi ja siirtyy sitten seuraavalle tasolle ja jatkaa tätä, kunnes koko puu on käyty läpi. Lopulta algoritmi saa pelipuun juurisolmuun arvoksi halutun parhaan mahdollisen tuloksen, jos vastustaja myös tekee parhaat mahdolliset siirrot (Fuller, Gaschnig & Gillogly 1973). Täten tekoäly löytää optimaalisen siirtopolun ja tekee sen mukaisen siirron. Kun vastustaja on tehnyt siirtonsa, algoritmi ajetaan uudestaan uudesta tilanteesta. Nyt voidaan löytää uusi optimaalinen siirtopolku, koska tästä tilanteesta voidaan suunnitella yksi vuoro pidemmälle kuin aiemmasta tilanteesta.

2.2 Minimax-algoritmi esimerkki



Kuvio 1. Esimerkki-pelipuu minimax-algoritmistä.

Esimerkki-kuvassa on pelipuu siirroista. Kuvassa punaiset nuolet merkitsevät, minkä lapsisolun arvo valitaan vanhemman solun arvoksi. Alimman tason solmuille al-

goritmi on laskenut näkyvät arvot. Arvot laskettuaan algoritmi aloittaa puun toisen läpikäymisen sen vasemmaisimmasta solmusta. Tämä on min-taso, joka kuvaa vastustajan vuoroa, eli algoritmi valitsee solun arvoksi vaihtoehdoista pienimmän. Algoritmi tutkii ensin solmun ensimmäistä lapsisolmua ja vertaa sen arvoa 4 sitten seuraavan lapsisolmun arvoon. Koska seuraavan solmun arvo 3 on pienempi, ja se on ainoa lapsisolmu, solun arvoksi tulee 3 eli pienin vaihtoehto. Tämä valitaan, koska jos vastustajalla on mahdollisuus valita näistä arvoista, hän varmasti valitsisi pelaajalle heikomman. Sitten algoritmi siirtyy alipuun seuraavaan solmuun ja vertaa lapsisolmujen arvoja -1 ja 5 ja valitsee niistä pienemmän -1.

Seuraavaksi alkaa alipuun seuraavan tason läpikäyminen, jossa pyritään maksimoimaan pistemäärä. Algoritmi vertaa ensimmäisen solmun lapsien arvoja 3 ja -1 ja valitsee 3. Seuraava taso on min-taso, ja siinä läpikäytävän solmun arvoksi tulee -1. Sitten algoritmi jatkaa siirtymällä käymään läpi seuraavaa alipuuta. Lopulta kun päästään juurisolmuun, verrataan vaihtoehtoja -1, 2 ja 0, ja niistä valitaan suurin eli 2. Tämä arvo on siis paras, jonka pelaaja voi saada, kun sekä hän että vastustaja tekevät täysin optimaaliset siirrot. Nyt ohjelma tietää, että tämän siirron tekemällä se todennäköisimmin pääsee parhaaseen loputulokseen. Seuraavalla vuorolla uudesta tilanteesta tehtäisiin uusi pelipuu, ja algoritmi ajettaisiin sille samalla lailla. Esimerkin puu on hyvin suppea, koska jokaisessa solmussa on vain 1-3 siirtomahdollisuutta. Oikeissa peleissä pelaajalla on yleensä jokaisella vuorolla useita kymmeniä siirtovaihtoehtoja, ja läpikäytävät puut ovat paljon laajemmat.

Ohessa pseudokoodi-esimerkki minimax-algoritmistä.

```
function minimax (pelaaja, solu, syvyys);
if (solu = viimeinensolu or solu = 0) then
    return solu;
end
if pelaaja = true then
    parasarvo := −∞ ;
    for solunlapsi do
        solunarvo := minimax(false, solunlapsi, syvyys − 1);
        parasarvo := maksimi(parasarvo, solunarvo);
    end
    return parasarvo ;
else
    parasarvo := ∞ ;
    for solunlapsi do
        solunarvo := minimax(true, solunlapsi, syvyys − 1);
        parasarvo := minimi(parasarvo, solunarvo);
    end
    return parasarvo ;
end
/* algoritmia kutsutaan komennolla
   minimax(true,alkusolu,syvyys)                                    */
```

Algoritmi 1: Pseudokoodi esimerkki minimax-algoritmista

2.3 Algoritmin puutteet

Minmaxin ongelmana on, että se joutuu käymään läpi kaikki mahdolliset siirtopuun solmut haluttuun syvyyteen asti, kuten esimerkistä nahdään. Monimutkaisissa peleissä kuten shakki on kuitenkin valtava määrä siirtoja, esimerkiksi shakissa niitä on noin 10^120 (Shannon 1950). Näin suuren määrän vaihtoehtoja läpikäyminen vaatii paljon prosessointikapasiteettia eikä ole käytännössä mahdollista.

Peleissä on myös rajoitteita, joiden takia algoritmilla on rajallinen aika laskea siirtonsa. Esimerkiksi shakkialgoritmien pitää Fullerin et al. artikkelin (1973) mukaan pystyä pelaamaan shakkipeli shakin turnaussääntöjen mukaisessa kahdessa tunnissa. Tämän takia algoritmillä ei voida tutkia koko pelin pelipuuta, vaan tutkittavan puun syvyys pitää rajata järkevämpään määrään. Tämä tarkoittaa, että tekoälyn kyky suunnitella siirtonsa pitkälle rajoittuu. Yleensä käytännössä esimerkiksi shakissa minimax-algoritmilla ei pystytä suunnittelemaan siirtoja edes kymmenen siirron päähän. Schwabin (2009) mukaan shakissa pelaajalla voi hyvinkin olla noin 35 mahdollista siirtoa joka vuorolla. Jos tästä tehtäisiin pelipuu, jossa on kuusi tasoa, eli kolmen vuoron siirrot, alimmalla tasolla solmuja olisi lähes 2 miljardia. Kymmenen tasoisessa puussa niitä olisi lähes kolmetuhatta biljoonaa ($3\times10\hat{}15$), ja tämä puu tutkii peliä vain viisi vuoroa eteenpäin (Schwab 2009). Tämänkään puun läpikäyminen ei olisi järkevää, vaan veisi liikaa aikaa ja resursseja tehokkailtakin tietokoneilta, eikä siltikään olisi tarpeksi hyviä ihmispelaajia vastaan.

3 Alfa-beta-karsinta

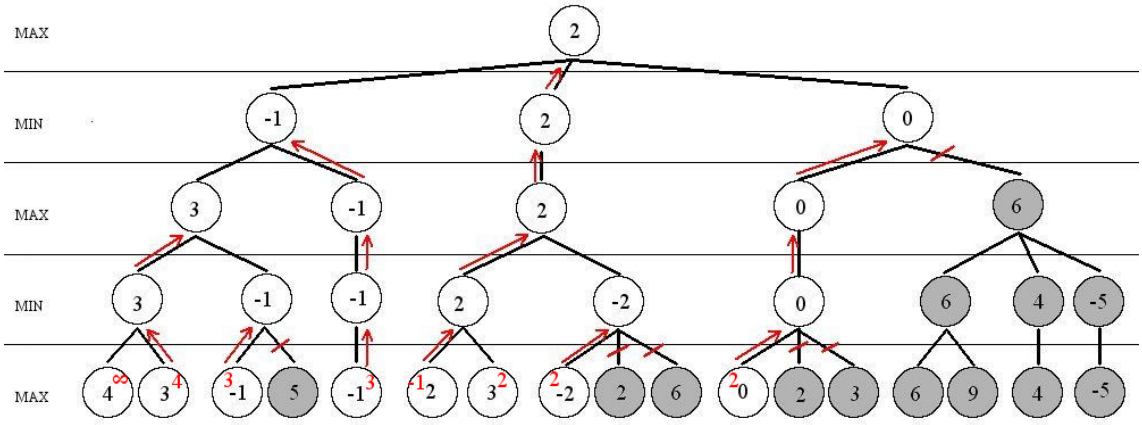
Alfa-beta-karsinnan tarkoitus on vähentää niiden solmujen määrää, jotka minimax algoritmi käy läpi. Sen kehittämiseen johti Abramsonin (1989) mukaan se, että havaittiin että minimax-algoritmissä olevan helposti tunnistettava joukko siirtoja, joita ei valita, ja jättämällä ne pois tarkastelusta pystyttäisiin siirtoja tutkimaan syvemmälle (Abramson 1989). Alfa-beta-algoritmi lopettaa siirron arvioimisen, kun se on löytänyt vaihtoehdon, joka todistaa siirron olevan huonompi kuin aiemmin läpikäydyt siirrot. Tällöin se siirtyy arvioimaan seuraavaa siirtoa. Knuthin ja Mooren (1975) mukaan Alfa-beta-karsinta päätyy samaan siirtoon kuin minimax ja poistaa puusta haarat, jotka eivät voi vaikuttaa lopputulokseen. Se myös yleensä pääsee lopputulokseen vähemmällä vaivalla ja nopeammin kuin minimax (Knuth & Moore 1975). Fuller et al. (1973) mukaan ”Alfa-beta algoritmin käyttö yksinkertaisen minimaxin sijaan vähentää tutkittavien tilanteiden määrää suuresti” (Fuller, Gaschnig & Gillogly 1973). Kun tutkittavien tilanteiden määrä vähenee, voidaan samassa ajassa tutkia pelipuuta syvemmin. Käytännössä on havaittu, että alfa-beta-karsintaa käyttämällä pystytään tutkimaan samaa puuta kaksi kertaa syvemmälle kuin pelkkää minimax-algoritmia käyttämällä (Fuller, Gaschnig & Gillogly 1973). Koska alfa-beta-karsinta toimii samalla lailla kuin yksinkertainen minimax-algoritmi, sitä voidaan soveltaa samoihin käyttötarkoituksiin.

3.1 Alfa-beta-karsinnan toiminta

Alfa-beta-karsinta-algoritmi on kehitetty tehostamaan minimax-algoritmin toimintaa. Sen toimintaa kuvaavat Knuthin ja Mooren (1975) , Fuller et al. (1973) sekä Abramsonin (1989) artikkelit. Se lisää minimax-algoritmin toimintaan alfa- ja beta-arvot, jotka ovat läpikäytävän solmun väliaikaiset voittoarvot. Alfa-arvo on maksimoitavan solmun arvon alaraja ja beta minimoitavan arvon yläraja. Alfan alkuarvo on miinus ääretön ja betan ääretön, eli molemmilla on alussa huonoin mahdolinen arvo (Knuth & Moore 1975). Puuta läpikäytäessä algoritmi vertaa min-tason soluja tutkittaessa solun arvoa betaan, ja max-tasolla alfaan. Kun algoritmi löytää

paremman arvon kuin niissä jo on, kuten alfalle suuremman, se sijoittaa tämän arvon alfaksi. Samoin toimitaan, kun betalle löytyy pienempi arvo. Esimerkiksi, jos maksimoitavan solmun jälkeläisiä läpikäytäessä löytyy lapsisolmu, jonka arvo on pienempi kuin alfa-arvo, on varmaa, että tämä solmun jälkeläisten läpikäynnillä ei voi saada parempaa arvoa kuin sen vanhemmalla on. Nyt solmun käsitteleminen voidaan lopettaa. Näin siksi, että minimoiva pelaaja valitsisi tässä tilanteessa aina siirrot, jotka tuovat pienimmän arvon. Minimoitaville solmuille algoritmi toimii samoin ja keskeytyy, kun arvo on suurempi kuin beta (Fuller, Gaschnig & Gillogly 1973). Algoritmi voi tämän jälkeen siirtyä käsittelemään seuraavaa solmua ilman, että kaikkia läpikäydyn solmun lapsisolmuja tarvitsee tutkia. Täten puusta ei tarvitse käydä läpi turhia solmuja joihin lopputulos ei voi päätyä, eikä niitä ei tarvitse edes välttämättä luoda eikä niiden arvoja laskea puuta luotaessa. Tämä säästää aikaa ja tehoa myös tässä mielessä. Fullerin et al. (1973) mukaan alfa-beta-karsinta-algoritmi on identtinen minimax algoritmin kanssa, paitsi kun arvo saadaan pienemmäksi kuin alfa, ja toisin päin suurempi kuin beta, joissa tilanteissa tapahtuu seuraavien solmujen poisleikkaus (Fuller, Gaschnig & Gillogly 1973).

3.2 Alfa-beta esimerkki



Kuvio 2. Esimerkki-pelipuu alfa-beta-algoritmistä.

Esimerkissä harmaat solut ovat algoritmin karsimia soluja. Alfa-arvot ovat punaisella solun arvon vasemmalla puolella ja beta-arvot oikealla. Alfa-beta-karsinta toi-

mii samoin tavoin kuin minimax-algoritmi joillakin muutoksilla. Ennen puun tutkimisen alkamista luodaan alfa-arvo, jonka aloitusarvo on negatiivinen ääretön, ja beta, joka on ääretön. Esimerkissä puun tutkiminen aloitetaan samalla lailla alimman min-tason vasemmanpuoleisimmasta solmusta. Kun tutkitaan sen ensimmäistä lapsisolmua, sen arvoa 4 verrataan betaan, ollaan min-tasolla. Koska se on pienempi kuin betan arvo ääretön, se sijoitetaan betan uudeksi arvoksi, ja asetetaan solmun vanhemman arvoksi. Sitten tutkitaan seuraavaa solmua, jonka arvo 3 on pienempi kuin beta, joten se sijoitetaan taas betaan ja vanhempaan solmuun. Koska nyt kaikki min-tason vasemmanpuoleisimman solmun lapset on tutkittu, siirrytään tutkimaan sen vanhempaa. Nyt siis tutkitaan max-tason vasemmanpuoleisinta solmua, ja sen ensimmäisen lapsisolmun arvo 3 asetetaan alfa-arvoksi, koska se on suurempi kuin alfan alkuarvo miinus ääretön.

Aletaan käymään läpi solun seuraavaa lasta, josta siirrytään edelleen sen ensimmäiseen lapseen. Nyt tarkasteltavan solmun arvo on -1, joka on pienempi kuin alfan arvo 3. Tullaan siis tilanteeseen, jossa jos puussa siirrytään tänne päin, vastustaja voi aina tehdä siirron, joka tuottaa huonomman loppuarvon kuin toinen siirtomahdollisuus. Tämän takia pelaajan ei missään tapauksessa kannata tehdä siirtoja, jotka johtaisivat tähän tilanteeseen, koska parempi siirtomahdollisuus on aina olemassa. Tämän takia puun tämän haaran solmujen tutkinta lopetetaan tähän, ja palataan takaisin edelliseen solmuun, jonka arvoksi asetetaan -1, koska tästä haarasta voidaan enää saada vain huonompia tuloksia. Sitten ylemmän tason solmulle tulee arvoksi 3, joka on sen lasten arvoista suurempi. Tutkinta jatketaan sitten ylös seuraavan tason solmuun, josta tutkitaan sen toinen lapsisolmu ja sen lehtisolmu. Sen arvoa -1 verrataan betaan, joka on nyt 3 joten, koska arvo on pienempi kuin beta, se otetaan solmun arvoksi.

Tutkintaa jatketaan tällä tavalla, ja kun siirrytään tutkimaan juurisolmun oikeanpuoleisinta jälkeläistä, huomataan sen ensimmäistä jälkeläistä tutkittaessa, että jos tähän solmuun siirrytään, lopputulokseksi saadaan enintään 0, joka on huonompi kuin tämän hetken alfa-arvo 2. Tällöin voidaan katkaista koko oikeanpuoleinen puun haara, koska vaikka sieltä voitaisiin löytää parempi tulos, sinne ei kuiten-

kaan voida päätyä. Lopputulokseksi juurisolmuun tulee siis 2, joka saatiin tulokseksi minimax-algoritmillakin. Nyt samaan lopputulokseen päästiin kuitenkin tutkimalla vähemmän solmuja.

Esimerkistä nähdään, että lopputulokseksi tulee sama tulos kuin minimax-algoritmilla, mutta alfa-beta-karsinnalla ollaan voitu jättää useita solmuja kokonaan tutkimatta. Vaikka esimerkissä eri vaihtoehtoja ja tasoja oli vain muutamia, karsittiin siitä silti yli puolet lehtisolmuista. Suuremmissa puissa, kuten shakkipuissa joissa on useita kymmeniä siirtovaihtoehtoja joka siirrolla, karsittavia solmuja on paljon enemmän.

Tässä esimerkissä koko puu luotiin ja sen alimpien solmujen arvot laskettiin ennen kuin sitä alettiin käymään läpi, mutta käytännössä usein samalla kun puuta luodaan, sen solmujen arvoja vertaillaan. Eli samalla kun algoritmi luo solun alimmalle tasolle, se vertaa sille laskettua arvoa alfa- tai beta-arvoon. Jos katkaisu tapahtuu, lopetetaan nykyisen solmun sisarsolmujen luominen ja siirrytään solmun vanhemmasta seuraavaan solmuun puussa, ja tästä jatketaan puun luomista. Näin tehtäessä kun suoritetaan alfa-beta-leikkaus, säästytään myös leikattavien solujen luomiselta ja arvojen laskemiselta sekä niiden vertaamiselta muiden solmujen arvoihin, mikä parantaa ohjelman tehokkuutta vielä enemmän.

Ohessa pseudokoodi-esimerkki alfa-beta-algoritmistä.

```
function alfabeta (pelaaja, solu, syvyys, α, β);
if (solu = viimeinensolu or solu = 0) then
    return solu;
end
if pelaaja = true then
    parasarvo := −∞ ;
    for solunlapsi do
        solunarvo := alfabeta(false, solunlapsi, syvyys − 1, α, β);
        parasarvo := maksimi(parasarvo, solunarvo);
        α := maksimi(α, parasarvo) ;
        if β ≤ α then
            break
        end
    end
    return parasarvo ;
else
    parasarvo := ∞ ;
    for solunlapsi do
        solunarvo := alfabeta(true, solunlapsi, syvyys − 1, α, β);
        parasarvo := minimi(parasarvo, solunarvo);
        β := minimi(β, parasarvo) ;
        if β ≤ α then
            break
        end
    end
    return parasarvo ;
end
/* algoritmia kutsutaan komennolla
   minimax(true,alkusolu,syvyys,−∞,∞ )                    */
```

Algoritmi 2: Pseudokoodi esimerkki alfa-beta-algoritmista

3.3 Alfa-beta-karsinta käytännössä

Minimax-algoritmin ja Afla-beta karsinnan on todettu toimivan käytännössä. Niitä käyttävät pelien tekoäly-ohjelmat osaavat pelata hyvin shakin kaltaisia pelejä. Esimerkiksi kuuluisa IBM:n kehittämä Deep Blue -shakkiohjelma, joka ensimmäisenä tekoälynä voitti shakin ihmissuurmestarin kun se voitti Garri Kasparovin vuonna 1997. Deep Blue käytti minimax ja alfa-beta-algoritmejä siirtojensa suunnittelussa (IBM 2018).

Sitä seuranneiden kahdenkymmenen vuoden aikana tietokoneiden laskentateho on kasvanut vielä suuresti. Siitä johtuen niiden pelipuiden koko, jonka algoritmit pystyvät tutkimaan, on kasvanut. Nykyään shakissa ihmisten ja tekoälyn pelatessa hyvä tekoäly voittaa aina, koska se pystyy suunnittelemaan siirtonsa alfa-beta-karsinnalla paljon pidemmälle kuin paraskaan ihminen. Tämän hetken tehokkaimmat shakkiohjelmat Stockfish (Stockfish 2018) ja Houdini (Houdini 2018) käyttävät molemmat alfa-beta-karsintaa osana toimintaansa. Shakkiohjelmat käyttävät myös muitakin algoritmejä siirtojen suunnitteluun, kuten tietoja aiemmista peleistä ja yleisiä tietoja hyvistä siirroista, mutta yleensä alfa-beta-karsinta on niissä tärkeänä osana (Abramson 1989).

4 Yhteenveto

Tutkimuksen keskeiset johtopäätökset ovat, että minimax-algoritmia käyttämällä vuoropohjaisia pelejä pelaava tekoäly pystyy löytämään todennäköisimmin voittavan siirron. Algoritmin puute on kuitenkin sen tehottomuus, joka aiheutuu siirtojen, jotka eivät lopulta vaikuta lopputulokseen, turhasta läpikäymisestä. Alfa-beta-karsintaa käyttämällä algoritmi pystyy karsimaan pois näitä turhia siirtoja. Koska shakin kaltaisissa monimutkaisissa peleissä jokaisen vuoron mahdollisten siirtojen määrä on suuri, karsimalla turhia siirtoja pystytään pelipuuta tutkimaan huomattavasti syvemmälle. Tekoäly pystyy näin suunnittelemaan siirtonsa pidemmälle ja todennäköisesti siten pelaamaan peliä paremmin. Algoritmit on havaittu toimiviksi käytännössä ja niitä käyttävät ohjelmat pystyvät voittamaan parhaat ihmispelaajat.

Algoritmeissä on kuitenkin mahdollisesti parannettavaa. Abramson (1989) käy artikkelissan läpi muutoksia algoritmeihin ja muita vaihtoehtoisia algoritmeja, jotka mahdollisesti voivat tehostaa minimax-algoritmia. Abramsonin mukaan minimax- ja afla-beta-algoritmien tehokkuus riippuu myös siitä, kuinka hyvin eri lopputilanteiden tarkkuus saadaan arvioitua, ja huono tilanteiden vertailu voi johtaa algoritmin päätymään huonoon lopputulokseen (Abramson 1989).

Mahdollisia tutkimusaiheita onkin se, kuinka algoritmeja on mahdollista kehittää edelleen. Tietokoneiden laskentatehon kasvaessa samalla myös siirtojen määrä, jonka algoritmit voivat käydä läpi, tulee kasvamaan. Myös sitä, kuinka solujen arvot lasketaan, voi kehittää, jotta eri lopputilanteiden hyvyys saadaan arvioitua tarkasti.

Kirjallisuutta

Abramson, B. 1989. *Control Strategies for Two-Player Games* Teoksessa ACM Computing Surveys, Volume 21 (June 1989), Issue 2 s. 137-161.

Asprey, S. , Rustem, B. & Žaković, S. 2001. *Minimax Algorithms with Applications in Finance and Engineering* Teoksessa IFAC Proceedings Volumes, Volume 34 (September 2001), Issue 20 s. 13–17.

Fuller, S. , Gaschnig, J. & Gillogly, J. 1973. *Analysis of the alpha-beta pruning algorithm*. Computer Science Department. Paper 1701.

Houdini kotisivu.
Saatavilla WWW-muodossa <URL: `http://www.cruxis.com/chess/houdini.htm`>. Viitattu 10.4.2018.

IBM Deep Blue. Saatavilla WWW-muodossa <URL: `https://www.research.ibm.com/deepblue/meet/html/d.3.2.html`>. Viitattu 2.4.2018.

Knuth, D. & Moore, R. 1975. *An Analysis of Alpha–Beta Pruning*. Teoksessa Artificial Intelligence, Volume 6 (1975), Issue 4 s. 293–326.

Maschler, M. , Solan, E. & Zamir, S. (2013). *Game Theory* Cambridge: Cambridge University Press

Schwab, B. 2009. *AI game engine programming.* (2nd ed.) luku 12. Boston, Mass.: Course Technology

Shannon, C. 1950. *Programming a Computer for Playing Chess*

Stockfish kotisivu. Saatavilla WWW-muodossa <URL: `https://stockfishchess.org/`>. Viitattu 10.4.2018.