Samu Kumpulainen

AlphaGo ja Monte Carlo -puuhaku

Tietotekniikan kandidaatintutkielma

28. huhtikuuta 2017

Jyväskylän yliopisto

Tietotekniikka

Tekijä: Samu Kumpulainen

Yhteystiedot: samu.p.kumpulainen@student.jyu.fi

Työn nimi: AlphaGo ja Monte Carlo -puuhaku

Title in English: AlphaGo and the Monte Carlo Tree Search

Työ: Kandidaatintutkielma

Sivumäärä: 26+0

Tiivistelmä: Tässä tutkielmassa käsittelen Monte Carlo -puuhakualgoritmia sekä sen roolia hyvin menestyneessä AlphaGo-tekoälyssä. Tavoitteena oli muodostaa kokonaiskuva AlphaGo:n toiminnasta, painottuen etenkin Monte Carlo -puuhaun näkökulmaan. Tutkimuksen perusteella selvisi miten ohjelma hyödyntää omassa hakualgoritmissään kyseistä puuhakua, jota se parantaa hyödyntäen koneoppimista ja useita eri tarkoituksiin opetettuja neuroverkkoja. AlphaGo saavutti näin tehokkuuden, johon muut go-ohjelmat eivät vielä ole kyenneet.

Avainsanat: AlphaGo, Monte Carlo -puuhaku, koneoppiminen, tietokone-go

Abstract: In this thesis I study the Monte Carlo tree search algorithm, and it's role in successful go-program, AlphaGo. The goal was to form a general view of AlphaGo's inner workings, especially from Monte Carlo tree search's point of view. I found out how the program's own search algorithm used Monte Carlo's basic structure, improving it with machine learning and neural networks trained for different purposes, thus obtaining the efficiency none of the other go-programs have yet to achieve.

Keywords: AlphaGo, Monte Carlo Tree Search, machine learning, computer-go

Olennaisia käsitteitä

Tässä listataan tutkielman kannalta olennaisia käsitteitä lyhyesti selitettyinä, sekä mahdollisia lyhenteitä.

Algoritmi Askelista koostuva toimintaohje, jota soveltamalla päästään haluttuun lopputulokseen.

Hakualgoritmi Algoritmi, jota käyttämällä saadaan noudettua tietoa jostain tietorakenteesta, esimerkiksi puusta.

Puu Abstrakti tietorakenne, joka koostuu solmuista ja niiden välisistä kaarista. Puussa on yksi juurisolmu, jolla voi olla useita lapsisolmuja, jotka puolestaan voivat muodostaa omat alipuunsa. Puun solmuja, joilla ei ole lapsisolmuja, sanotaan lehtisolmuiksi.

Pelipuu Puurakenne, jossa jokainen puun solmu esittää yhtä pelitilannetta, ja jokainen solmujen välinen kaari pelin siirtoa.

MCTS Englanniksi Monte Carlo tree search, eli Monte Carlo -puuhaku. Hakualgoritmi, joka hyödyntää satunnaisuutta ja simulaatiota pelipuun läpikäyntiin.

Koneoppiminen Koneoppiminen tarkoittaa ohjelman toimintaa, jossa se itse oppii ja kehittyy datan avulla ilman, että sitä erikseen ohjelmoidaan jokaiseen tilanteeseen.

Säie Kevyempi tietokoneohjelman prosessi, jolla ei ole omia resursseja, vaan se käyttää sen prosessin resursseja, johon se kuuluu. Prosessi voi siis sisältää yhden tai useampia säikeitä.

Kuviot

Kuvio 1. Tyypillinen alkutilanne eli *fuseki* (philosopher 2006). 4
Kuvio 2. Atari-tilanne (Lindström 2005). 5
Kuvio 3. Ko-tilanne (B 2006). 5
Kuvio 4. MCTS:n päävaiheet (mukaillen Chaslot 2010). 10
Kuvio 5. APV-MCTS:n päävaiheet (mukaillen Silver ym. 2016). 16

Sisältö

1 JOHDANTO 1
1.1 Tutkielman rakenne 1

2 GO-PELI 3
2.1 Säännöt 3
2.2 Tietokone-go 6

3 MONTE CARLO -PUUHAKU 8
3.1 Yleistä 9
3.2 Monte Carlo -puuhaun rakenne ja toiminta 9
3.3 Parantaminen muilla tekniikoilla 11
3.3.1 RAVE 11
3.3.2 Rinnakkaistaminen 12

4 ALPHAGO 14
4.1 Koneoppiminen 14
4.2 AlphaGo:n toiminta 15

5 YHTEENVETO 18

LÄHTEET 19

1 Johdanto

AlphaGo on Google DeepMind -yrityksen kehittämä tekoäly, joka pelaa Kiinasta lähtöisin olevaa klassista go-peliä. Go:ssa pelialueen tarjoama siirtomahdollisuuksien valtava määrä ja eri liikkeiden arvioinnin hankaluus tekevät siitä erittäin vaikean tekoälylle (Silver ym. 2016). Tässä kirjallisuuskatsauksessa tutkitaan, miten AlphaGo toimii, keskittyen etenkin sen hyödyntämään Monte Carlo -puuhakualgoritmiin ja sen toimintaan. Viime vuosina koneoppiminen ja tekoäly ovat olleet tutkimuksen ja uutisoinnin kohteena, ja AlphaGo on hyvä esimerkki siitä miten merkittäviä tuloksia moderni tekoäly voi saavuttaa. Vuonna 2015 AlphaGo voitti 5-0 kolminkertaisen Euroopan mestarin Fan Huin, mikä oli ensimmäinen kerta kun tietokone voitti ammattilaispelaajan go:ssa ilman tasoitusta. Wangin ym. (2016) mukaan tällainen läpimurto tuli merkittävästi aiemmin mitä kukaan osasi odottaa, mikä kertoo tekoälyjen huomattavasta kehittymisnopeudesta.

Tekoäly tieteenalana pyrkii ymmärtämään älyllisiä olentoja, sekä siten rakentamaan niitä (Russell, Norvig ja Intelligence 1995). Klassiset kahden pelaajan pelit, kuten shakki, tammi, ja backgammon sopivat erinomaisesti tekoälyn tutkimiseen, koska ne tarjoavat suljetun ympäristön ja tarkat säännöt, joten vertailu ihmisälyyn on helppoa (Gelly ym. 2012). Tutkittavat pelit ovat niin kutsuttuja täyden informaation pelejä, eli kaikki mahdollinen tieto pelistä on pelaajien nähtävissä ja hyödynnettävissä. Go on pitkään ollut näistä vaikein tekoälylle, mutta viime vuosina etenkin Monte Carlo -puuhaun ansiosta edistystä ja huomattavia voittoja on saatu aikaan myös tässä pelissä.

Monte Carlo -puuhausta (engl. Monte Carlo tree search), josta käytän tässä tutkielmassa lyhennettä MCTS, kirjallisuutta löytyi runsaasti, koska algoritmia on aiemmin käytetty monissa go-peliä pelaavissa tekoälyissä (Gelly ym. 2012). Koska AlphaGo on aiheena uusi, löytyi siitä tietoa varsin rajallisesti.

1.1 Tutkielman rakenne

Toisessa luvussa käsittelen ensimmäiseksi go-peliä, kerron hieman sen historiasta ja alkuperämyytistä, sekä pelin säännöt, jotka ovat melko yksinkertaiset pelin strategisesta monimut-

kaisuudesta huolimatta.

Kolmannessa luvussa tarkastelen Monte Carlo -puuhakualgoritmin toimintaa, taustoja, eri käyttötarkoituksia ja hyödyntämismahdollisuuksia. Lisäksi kerron eri tekniikoista, joita on kehitetty sen parantamiseen go-pelin suhteen, kuten rinnakkaistamisesta.

Neljäs luku käsittelee AlphaGo-tekoälyohjelman toimintaa. Aluksi kerron lyhyesti neuroverkoista, joita se hyödyntää, ja sitten siirryn kuvaamaan sen toimintalogiikkaa.

Tutkielman lopussa oleva yhteenveto kokoaa tutkimieni asioiden pääpiirteet ja mahdolliset tekemäni johtopäätökset aiheesta.

2 Go-peli

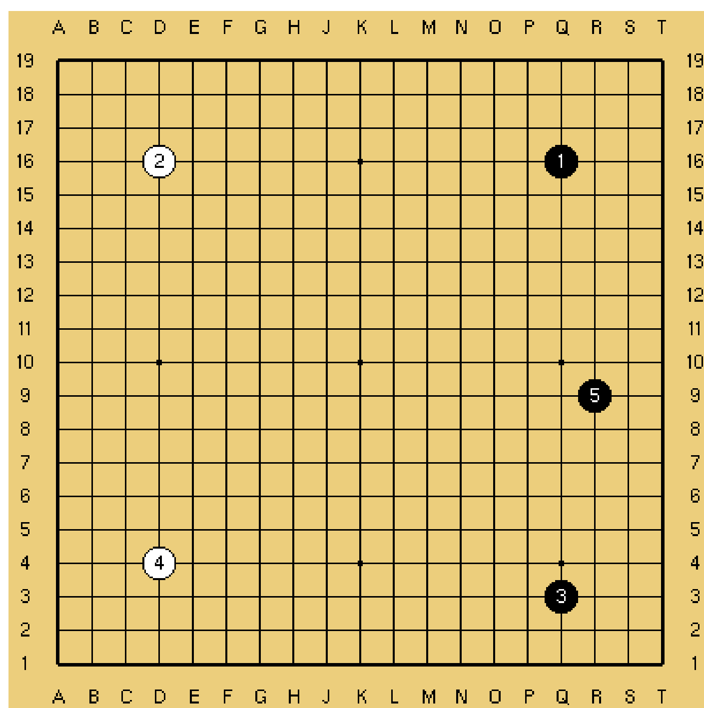
Go on alun perin Kiinasta lähtöisin oleva kaksin pelattava lautapeli. Peli tunnetaan myös nimellä baduk (Korea) ja weiqi(Kiina). Erään legendan mukaan sen keksi yli 4000 vuotta sitten Kiinan keisari Yaon neuvonantaja Shun opettaakseen keisarin pojalle strategiaa, sotataitoa ja kärsivällisyyttä (Sensei's Library 2017b). Historiallisesti vanhin maininta go:sta on kuitenkin vasta Kungfutsen teksteistä 500-luvulta eKr. (Sensei's Library 2017b). Peliä pelataan pysty- ja vaakaviivoista koostuvalla pelilaudalla, *goban*, jonka koko on yleensä joko 9x9, 13x13, tai 19x19, sekä valkoisilla ja mustilla pelinappuloilla eli kivillä, *goishi*. Go on siis vain kahden pelaajan peli, kuten toinen klassinen peli, johon sitä usein verrataan, shakki. Gon säännöt ovat yksinkertaiset, mutta mahdollisten siirtojen määrä ja pelilaudan suuri koko tekevät siitä strategisesti vaativan.

Go:ssa on käytössä yleisesti tasojärjestelmä, jossa pelaajat jaetaan kyu- ja dan-arvoihin. Aloittelevan pelaajan taso on noin 30. kyu, ja siitä ylöspäin arvot pienenevät 1. kyuhun asti. Kyu-arvojen jälken tulevat dan-arvot, joita on noin 6. daniin asti. Nämä ovat amatööripelaajien arvoja, ammattilaispelaajille on toinen, korkeimmillaan 9. daniin nouseva arvoasteikko. Eri tasoisten pelaajien taitoeroa tasoitetaan antamalla heikommalle pelaajalle tasoituskiviä pelin alussa.

2.1 Säännöt

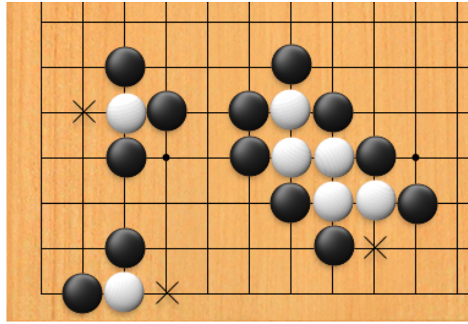
Pelaajan tavoite go:ssa on vallata pelilaudalta mahdollisimman suuri alue ympäröimällä se omilla kivillään. Pelilaudan viivojen risteyskohtia kutsutaan pisteiksi (engl. point). Kumpikin pelaaja asettaa vuorotellen yhden omista kivistään mihin tahansa pelilaudan pisteeseen, jossa ei ole jo kiveä. Tätä kutsutaan siirroksi. Kiveä ei voi sen asettamisen jälkeen enää liikuttaa. On kuitenkin joitain tilanteita, joissa tiettyyn pisteeseen ei voi pelata kiveä, ja joitain tilanteita, joissa siirron jälkeen poistetaan kiviä laudalta. Mustilla kivillä pelaava aloittaa aina. Pelaaja voi myös kiven pelaamisen sijaan ohittaa vuoronsa. Molempien pelaajien ohitettua vuoronsa peräkkäin peli päättyy.

Vastustajan kiven voi poistaa laudalta ympäröimällä sen kokonaan omilla kivillään. Kiven



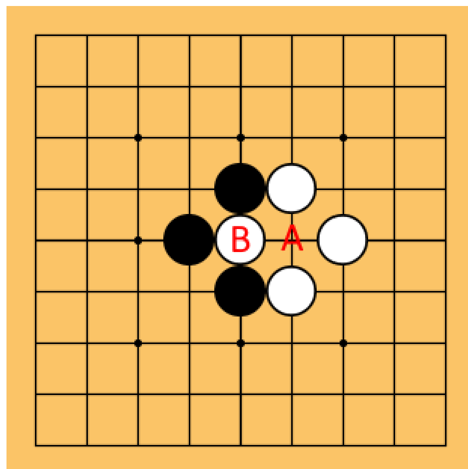
Kuvio 1. Tyypillinen alkutilanne eli *fuseki* (philosopher 2006).

pysty- ja vaakasuorassa viereisiä pisteitä sanotaan *vapauksiksi* (Sensei's Library 2017a). Kun kaikki kiven vapaudet on viety, eli toinen pelaaja on ympäröinyt sen omilla kivillään, kivi on 'valloitettu', ja se poistetaan laudalta. Kiveä ei saa pelata tilanteeseen, jossa sillä ei ole yhtään vapautta, ellei sille siirron seurauksena muodostu sellaisia. Pelaaja voi turvata kiviään paremmin yhdistämällä niitä ketjuiksi, lisäten näin kiviryhmän vapauksia ja hankaloittaen kivien valtaamista. Pelilaudan reunassa olevilla kivillä on luonnollisesti vähemmän vapauksia, joten ne on helpompi vallata. Kun kivellä tai kiviryhmällä on vain yksi vapaus jäljellä ja sen voi vallata seuraavalla vuorolla, sanotaan tätä tilannetta *atariksi* (Sensei's Library 2017a). Kuvassa 2 on useita atari-tilanteita. Pelaamalla seuraavalla vuorolla johonkin merkityistä pisteistä saisi musta vallattua ryhmän valkoiset kivet.



Kuvio 2. Atari-tilanne (Lindström 2005).

Viimeinen go:n sääntö on *ko*-sääntö, jonka mukaan edellistä laudan tilannetta ei saa toistaa (Sensei’s Library 2017a). Tämä koskee kuitenkin vain edellisen vuoron tilannetta, eli jos pelaaja pelaa välille jonnekin muualle, laudan tilanne muuttuu, ja ko-tilanteeseen pelaaminen on taas sallittua (Sensei’s Library 2017a). Kuvan 3 ko-tilanteessa musta voi nyt pelata kiven pisteeseen A, vallaten ja poistaen laudalta valkoisen kiven pisteessä B. Valkoinen ei saa nyt pelata kiveä pisteeseen B, koska silloin edellinen laudan tilanne toistuisi.



Kuvio 3. Ko-tilanne (B 2006).

Go-pelin säännöistä on useita eri variaatioita, jotka vaikuttavat pisteytykseen, mutta eivät itse peruspeliin. Kaksi eniten käytettyä tapaa pisteidenlaskuun ovat **kiinalainen** ja **japanilainen** pisteytys (Lubberts ja Miikkulainen 2001). Kiinalainen pisteytys on yleisempi, ja siinä lasketaan yhteen pelaajakohtaisesti vastustajalta vallatut kivet ja omien kivien ympäröimät laudan

pisteet. Japanilaisessa pisteytyksessä lasketaan yhteen vastustajalta vallatut kivet, omien kivien ympäröimät laudan pisteet, sekä omien kivien määrä. Lisäksi koska aloittaminen antaa mustalle edun, myös tasavertaisten pelaajien pelejä tasoitetaan lopussa *komi*:n verran. Komin suuruus vaihtelee, mutta yleensä se on 5,5–8 pistettä.

2.2 Tietokone-go

Tietokone-go tarkoittaa tekoälytutkimusta, jonka tavoitteena on kehittää hyvin go-peliä pelaava tekoäly. Go:ta pelaavia tekoälyjä on yritetty rakentaa jo 1960-luvulta asti, vaihtelevin tuloksin. 1980-luvulla halvempien tietokoneiden yleistymisen ja suurten turnaussponsoreiden ansiosta alan tutkimus ja ohjelmien menestys lisääntyi monissa turnauksissa (Müller 2002). Merkittäviä go-turnauksia tietokoneille on useita. Näistä vanhinta, Ing-cup:ia järjestettiin vuosittain vuosina 1987-2000, ja siinä kilpailun voittajaohjelma pelasi nuoria, mutta taidokkaita ihmispelaajia vastaan (Bouzy ja Cazenave 2001). Vuosien aikana saavutettiin monia merkittäviä edistysaskeleita ohjelmien saralla. Vuonna 1991 *Goliath* voitti useita turnausvoittajia Ing cup:issa, 17 kiven tasoituksella, ja esimerkiksi *Handtalk* ja *KCC Igo* saivat 90-luvun lopulla 3. ja 2. kyu-arvot Japanin Go-yhdistykseltä menestyksensä johdosta (Müller 2002).

Go eroaa muista klassisista täyden informaation peleistä todella suurella etsintäavaruudellaan. Gossa normaalilla, 19x19-laudalla erilaisia asetelmia voi olla $3^{19x19} \approx 10^{170}$, joista 1.2% on sallittuja (Müller 2002). Näiden vertailu ja arviointi keskenään on tietokoneelle monimutkaista ja hidasta (Müller 2002). Perinteinen ratkaisumalli pelitekoälyissä, joka koostuu arviointifunktiosta, siirron generoimisesta, sekä puuhausta, on toimiva esimerkiksi shakissa, tammessa ja othellossa, mutta ei go-pelissä sen monimutkaisuuden vuoksi (Bouzy ja Cazenave 2001). Siinä missä toisten pelien tekoälyissä painopiste ja hankaluus on pelipuun optimoinnissa, go:ssa suurin hankaluus tulee hyvän arviointifunktion puutteesta (Bouzy ja Cazenave 2001). Lisäksi go:ssa siirrot voivat vaikuttaa kymmenien siirtojen päähän ja niiden ennakointi on ihmisille helpompaa kuin tekoälylle, ammattilaisille se on jopa alitajuista (Müller 2002).

Alpha-beta-karsinnan ($\alpha\beta$-karsinnan) käyttäminen oli Chaslotin (2010) mukaan pitkään ol-

lut normi kahden pelaajan peleissä, mutta sen toimiminen vaati hyvän arviointifunktion, jollaista ei go-pelille ole onnistuttu kehittämään. Vuoteen 2005 asti go-ohjelmat olivatkin $\alpha\beta$-karsinnan jostain variaatiosta, heuristiikoista, asiantuntijajärjestelmistä ja kuvioista koostuvia yhdistelmiä (Chaslot 2010). Viime vuosina uusien tekniikoiden, kuten Monte Carlo -puuhaun, kehittyminen on mullistanut tietokone-go:n (Gelly ja Silver 2011a). Kocsis'n ja Szepesvarin vuonna 2006 esittelemään, Monte Carlo -pohjaiseen UCT-algoritmiin (Kocsis, Szepesvári ja Willemson 2006) perustuva *Crazy Stone* voitti kaksi peliä 4. danin ammattilaista vastaan 8 ja 7 kiven tasoituksilla, ja *MoGo* voitti myös useita ammattilaispelaajia vastaan 7 ja 6 kiven tasoituksilla (Gelly ym. 2012). On siis selvästi nähtävissä, että Monte Carlo -tekniikoihin pohjautuvat algoritmit ja niihin perustuvat ohjelmat paransivat osaamistaan yhä enemmän. Viimeisin merkkipaalu alalla on tämänkin tutkielman aiheena oleva AlphaGo, joka saavutti tietokone-go:n huipun voitoillaan pelin mestareista vuosina 2015–2016.

3 Monte Carlo -puuhaku

Monte Carlo -puuhaku (MCTS) on yksi ns. Monte Carlo -metodeista, joille on yhteistä satunnaisuuden hyödyntäminen deterministisissä ongelmissa, esimerkiksi optimoinnissa ja integroinnissa. Metodien nimitys on viittaus tunnettuun Monte Carlon kasinoon Monacossa, koska kuten uhkapelit, nämäkin metodit pohjaavat satunnaisuuteen (Metropolis 1987). Termi yleistyi alun perin Los Alamosin laboratoriossa 1940- ja 1950-luvulla muun muassa John Neumannin, Enrico Fermin ja and Stanislaw Ulamin ansiosta (Hammersley 2013).

Monte Carlo -menetelmien menestyksekäs hyödyntäminen niin Yhdysvaltojen ydinkokeiden yhteydessä kuin deterministisissä ongelmissa sodan jälkeen lisäsi aiheen tutkimusta. Hammersleyn (2013) mukaan 1950-luvulla jokaista mahdollista ongelmaa yritettiin ratkaista näillä samoilla menetelmillä, vaikka ne sopivatkin hyvin vain osaan niistä. Tämä liikakäyttö johtikin menetelmien tehokkuuden arvosteluun. Edelleen Hammersley sanoo että 60-luvulla Monte Carlo teki menestyksekkään paluun, koska sen vahvuudet tunnettiin nyt paremmin ja oli myös kehitetty uusia tekniikoita sen hyödyntämiseen.

Tässä luvussa keskityn moderniin MC-puuhakuun, koska se on oleellinen osa AlphaGo:n toimintaa ja siten tätä tutkielmaa. Puuhaun hyödyntämisympäristö on yleensä pelien ja tekoälyn parissa. Pelien ulkopuolella menetelmää on myös käytetty heuristisessa haussa automatisoidun teoreeman todistuksen yhteydessä (Ertel, Schumann ja Suttner 1989).

MCTS-nimen otti käyttöön Rémi Coulom kehittäessään modernia versiota hausta *Crazy Stone*-ohjelmaansa (Coulom 2006). Sitä kuitenkin hyödynnettiin ensimmäisen kerran eräässä muodossa Bernd Brügmannin go-ohjelmassa *Gobble* jo vuonna 1993 (Brügmann 1993). Vuoden 2006 jälkeen MCTS:a on käytetty monissa hyvin menestyneissä go-tekoälyissa (ks. luku 2.2). Go-pelin lisäksi algoritmia on käytetty monissa muissa peleissä, esimerkiksi vuoden 2014 Total War: Rome II-strategiapelissä (Champandard 2014) ja Magic: The Gathering -keräilykorttipelissä (Ward ja Cowling 2009).

3.1 Yleistä

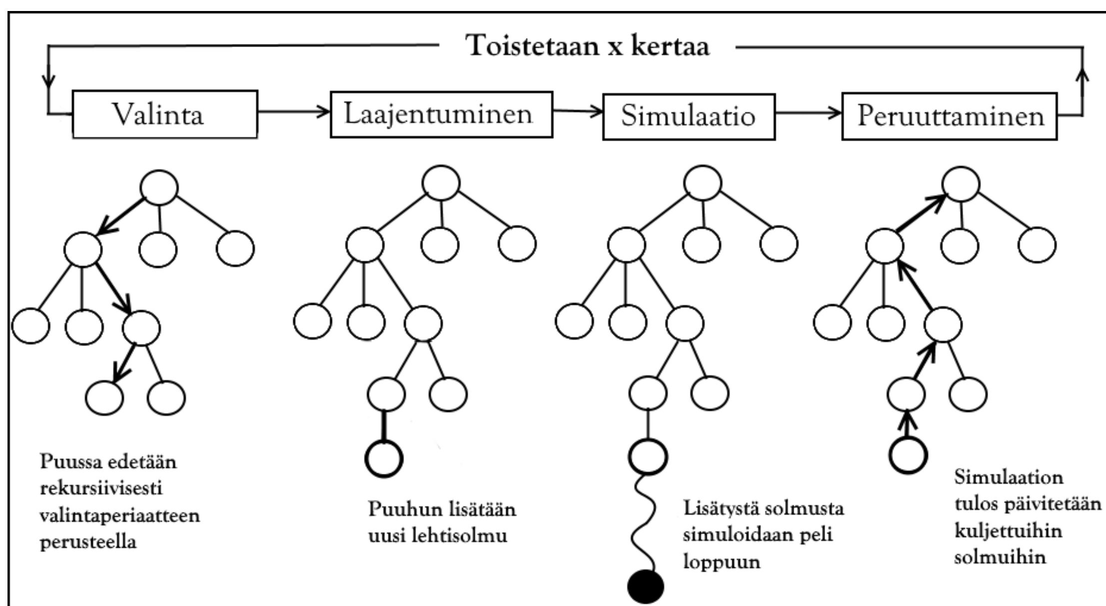
Hakualgoritmi toimii Russellin ja Norvigin (1995) mukaan ottamalla ensin ongelman, ja palauttamalla sitten siihen vastauksen toimintasarjan muodossa. Pelipuissa puun solmut esittävät eri pelitilanteita, ja solmujen väliset kaaret eri siirtoja. Puuhaut siis yleensä pyrkivät löytämään parhaan mahdollisen siirron vertailemalla saavutettavia pelitilanteita ja pyrkimällä niistä parhaaseen. Peleissä, joissa etsintäavaruus on suuri, kuten go:ssa ja shakissa, **kokonaisen pelipuun** tekeminen ja läpikäyminen on käytännössä mahdotonta. Kokonainen pelipuu sisältää kaikki mahdolliset siirrot ja pelitilanteet. Näissä peleissä tutkitaankin **osittaista pelipuuta**, eli tiettyä määrää tilanteita nykyisestä tilanteesta eteenpäin.

MCTS on käytännössä Monte Carlo -simulaatioiden ohjaama paras ensin -haku, eli haku, jossa siirrytään jonkin toimintamallin tai heuristiikan perusteella parhaaseen löydettyyn solmuun (Chaslot 2010). Se ei siis tarvitse erillistä arviointifunktiota pelitilanteiden arviointiin, joten se soveltuu todella hyvin go-peliin, jossa kunnollisen arviointifunktion tekeminen on erittäin vaikeaa (Chaslot 2010).

3.2 Monte Carlo -puuhaun rakenne ja toiminta

MCTS:n perusrakenne on seuraava: Alussa luodaan pelipuun juurisolmu, joka kuvaa nykyistä pelitilannetta (Niekerk ym. 2012). Jokainen puun solmu kuvaa yhtä mahdollista pelitilannetta ja sisältää lisäksi ainakin solmun nykyisen arvon ja tiedon, kuinka monesti solmussa on käyty (Chaslot 2010). Solmujen arvot perustuvat MC-simulaatiolla saatuihin tuloksiin (Gelly ja Silver 2008). Monte Carlo -puuhakualgoritmi koostuu neljästä eri vaiheesta (ks. luku 4), joita toistetaan niin kauan kuin aikaa riittää (Chaslot 2010):

1. Valinta (Selection)
2. Laajentuminen (Expansion)
3. Simulaatio (Simulation)
4. Peruuttaminen (Backpropagation)



Kuvio 4. MCTS:n päävaiheet (mukaillen Chaslot 2010).

Valintavaiheessa edetään puussa jonkin valintaperiaatteen mukaan, kunnes saavutaan lehtisolmuun. Periaate voi perustua solmujen arvoon, mutta siihen usein lisätään satunnaisuutta arvojen mahdollisen epätarkkuuden vuoksi. Esimerkiksi jollain hetkellä arvoltaan huonommalta näyttävä siirto voi lopussa johtaakin parempaan tilanteeseen.

Laajentumisvaiheessa lisätään nykyiseen lehtisolmuun uusi lapsisolmu, jonka jälkeen simulaatiovaiheessa pelataan peli loppuun juuri lisätystä lapsisolmusta eteenpäin. Pelaaminen tapahtuu simuloimalla Monte Carlo -simulaatiolla, eli pelaamalla osittain satunnaisia siirtoja. Strategia, jolla simuloitavan pelin siirrot päätetään, vaatii Chaslotin (2010) mukaan tasapainottelua kahden valinnan välillä. Ensiksi, tiedon ja haun välinen yhteys tarkoittaa, että lisäämällä tietoa siirron arviointiin siirroista voi tulla parempia, mutta jos tietoa on liikaa, itse haku hidastuu ja simuloituja pelejä ei ehditä pelata yhtä paljon. Tämä siis vähentäisi mahdollisten siirtojen määrää. Toinen valinta pitää tehdä satunnaisuuden ja omatun tiedon välillä. Jos pelatut siirrot ovat liian satunnaisia, ne ovat usein myös liian heikkoja. Toisaalta jos liikkeet määräytyvät aina varmempaan suuntaan, tulee ohjelmasta liian yksioikoinen ja siten sen pelitaito huononee.

Lopuksi peruutusvaiheessa peruutetaan takaisin nykyiseen pelitilanteeseen, päivittäen samalla kuljettujen solmujen arvot simulaation tuloksen mukaisiksi. Ajan päättyessä suoritettava siirto on esimerkiksi se lapsisolmu, jossa käytiin eniten. Muita vaihtoehtoja ovat esimerkiksi solmu, jolla on korkein arvo tai edellä mainittujen yhdistelmä (Chaslot 2010).

3.3 Parantaminen muilla tekniikoilla

Tässä kappaleessa esittelen eri tekniikoita, joita on käytetty MCTS:n parantamiseen gopelissä. MCTS itsessään tarjoaa tietyn rakenteen (ks. luku 3.2), jota pystyy muokkaamaan monipuolisesti. Ensin esittelen usein käytetyn RAVE-menetelmän, jonka jälkeen käsittelen rinnakkaislaskennan avulla tapahtuvaa tehostamista.

3.3.1 RAVE

Eräs tehokas menetelmä MCTS:n parantamiseen on ns. RAVE (engl. Rapid action value estimation). Siinä missä puhtaassa MC-puuhaussa joudutaan arvioimaan jokaisen tilan ja siirron arvo useissa eri simulaatioissa ja solmuissa, RAVE tarjoaa tavan jakaa tietoa solmujen kesken, nopeuttaen huomattavasti arvojen laskemista (Gelly ym. 2012; Gelly ja Silver 2011a). Sen idea perustuu AMAF-heuristiikkaan (engl. all-moves-as-first), jonka perusidea on muodostaa jokaiselle siirrolle arvo, joka ei riipu siitä missä vaiheessa peliä siirto tehdään (Gelly ym. 2012). RAVE:ssa siirron arvon yleistys tapahtuu alipuun suhteen, eli siis siirron a arvo arvioidaan tilasta s alkavista simulaatioista, pelattiin a missä vaiheessa tahansa s:n jälkeen (Gelly ja Silver 2011a).

RAVE:n heikkous tulee esille taktisissa tilanteissa, joissa pienikin viereinen liike voi muuttaa siirron elintärkeästä hyödyttömäksi tai toisinpäin (Gelly ym. 2012). MC-RAVE-algoritmi paikkaa tätä heikkoutta yhdistämällä puhtaan Monte Carlon ja RAVEN antamat arvot (Gelly ja Silver 2011b). Siinä AMAF-arvon painoarvoa vähennetään simulaatioiden määrän kasvaessa ja Monte Carlon antamaa arvoa painotetaan enemmän (Gelly ym. 2012).

3.3.2 Rinnakkaistaminen

Perusajatus rinnakkaistamisessa on lisätä suoritettujen simulaatioiden tahtia, saaden enemmän simulaatioita suoritettua samassa ajassa (Niekerk ym. 2012). Tämä tapahtuu joko käyttämällä yhtäaikaisesti useita prosessoriytimiä ja säikeitä samalla koneella, tai käyttämällä laskentaan useista koneista koostuvaa klusteria (Niekerk ym. 2012). Laittoiston näkökulmasta MCTS-algoritmi on helppo rinnakkaistaa verrattuna esimerkiksi minimax-hakuun, mutta tämä ei Gellyn (2012) mukaan tehosta sen toimintaa niin paljon kuin odottaisi. Ongelman muodostaa Niekerkin (2012) mukaan rinnakkaisvaikutus (engl. parallel effect), eli tehon menetys rinnakkaistettaessa tietty määrä simulaatioita: useat eri säikeet voivat esimerkiksi tutkia samasta tilasta lähteviä pelejä, vaikka tila todellisuudessa olisikin epäsuotuisa aloituskohta. Peräkkäisissä simulaatioissa ongelmaa ei ole, koska myöhemmät simulaatiot tietävät, mitkä solmut ovat huonompia, ja voivat siten tutkia muita puun solmuja.

Monte Carlo -puuhaun rinnakkaistamiseen käytetyimmät kolme tekniikkaa ovat lehtirinnakkaistaminen, juuririnnakkaistaminen ja puurinnakkaistaminen (Chaslot, Winands ja Herik 2008). Chaslotin ym. (2008) esittelemässä puurinnakkaistamisessa kaikki prosessointiyksiköt, esimerkiksi säikeet, käyttävät jaettua MCTS-puuta. Tässä tekniikassa rinnakkaisvaikutus näkyy siinä, että prosessointiyksiköt voivat tehdä samoja asioita päällekkäin, tuhlaten laskenta-aikaa (Niekerk ym. 2012). Tätä päällekkäisyyttä voidaan vähentää esimerkiksi virtuaalitappio-tekniikalla, jossa valintavaiheessa lisätään väliaikainen häviö käytyihin solmuihin, jotta muut prosessointiyksiköt eivät tutkisi samaa polkua, vaan keskittyisivät muihin (Niekerk ym. 2012).

Toinen tekniikka, lehtirinnakkaistaminen, käyttää yhtä isäntäsäiettä puun ylläpitämiseen ja useita orjasäikeitä simulaatioiden suorittamiseen tietyistä pelitilanteista (Niekerk ym. 2012). Tämän toteuttaminen on helppoa, mutta sisältää silti useita ongelmia: puuta ylläpitävä säie voi muodostaa pullonkaulan, useiden pelien yhtäaikainen pelaaminen vie enemmän aikaa, koska joudutaan odottamaan kaikki pelit loppuun, sekä kolmanneksi, koska tietoa ei jaeta, voidaan suorittaa häviöön johtavia simulaatioita turhaan, tuhlaten laskenta-aikaa (Chaslot, Winands ja Herik 2008; Niekerk ym. 2012).

Juuririnnakkaistamisessa puolestaan jokainen säie ylläpitää omaa MCTS-puutaan, ja suorit-

taa kaikkia neljää vaihetta MCTS-algoritmista (Niekerk ym. 2012). Tietyin väliajoin säikeet jakavat osia puistaan muiden kanssa (Niekerk ym. 2012). Myös tämä tekniikka on helppo toteuttaa vähäisen säikeiden välisen kommunikaation ansiosta (Chaslot, Winands ja Herik 2008).

4 AlphaGo

Tässä luvussa käsittelen AlphaGo-ohjelman toimintaa hyödyntäen edellisissä luvuissa käsiteltyjä tietoja go-pelistä ja Monte Carlo -puuhausta. AlphaGo-tekoälyn kehitti Google Deepmind -yritys. AlphaGo teki historiaa vuonna 2015 ollen ensimmäinen tekoäly, joka voitti go-pelissä ammattilaispelaajan täysikokoisella 19x19-laudalla ilman tasoitusta (Silver ym. 2016). Seuraavana vuonna sen voitto yhdestä maailman parhaista pelaajista, Lee Sedolista, nosti AlphaGo:n otsikoihin.

AlphaGo hyödyntää toiminnassaan MCTS-menetelmän lisäksi myös neuroverkkoja. Koska käsittelen tutkielmassa toimintaa enemmän MCTS:n näkökulmasta, esittelen neuroverkkojen ja koneoppimisen taustaa vain lyhyesti seuraavassa alaluvussa, ennen kuin siirryn itse kokonaisuuden toimintaan.

4.1 Koneoppiminen

Koneoppiminen tutkii ja kehittää algoritmeja, jotka pystyvät itse oppimaan ja kehittymään pohjadatan perusteella (Kohavi ja Provost 1998). Sen osa-alueisiin kuuluu muun muassa ohjattu oppiminen (engl. supervised learning) ja vahvistamisoppiminen (engl. reinforcement learning), joita hyödynnetään myös AlphaGo:n neuroverkkojen opettamisessa (Silver ym. 2016). **Ohjattu oppiminen** tarkoittaa Lisonin (2015) mukaan sitä, että opetettavalle algoritmille annetaan opetusdataa, joka koostuu syötteistä ja niitä vastaavista oikeista tuloksista. Prosessin tarkoitus on saada algoritmi tunnistamaan mistä syötteistä seuraa haluttuja tuloksia, jotta sitä voi edelleen käyttää uusien syötteiden analysointiin. **Vahvistamisoppiminen** puolestaan tarkoittaa sitä, että esimerkkidatassa ei ole tietoa, onko jokin tulos täysin oikea vai täysin väärä, mutta sen hyvyyttä voidaan kuitenkin arvioida jollain tavalla. Tätä arvion antamaa arvoa kutsutaan usein palkkioksi (engl. reward), ja se voi olla positiivinen tai negatiivinen (Lison 2015).

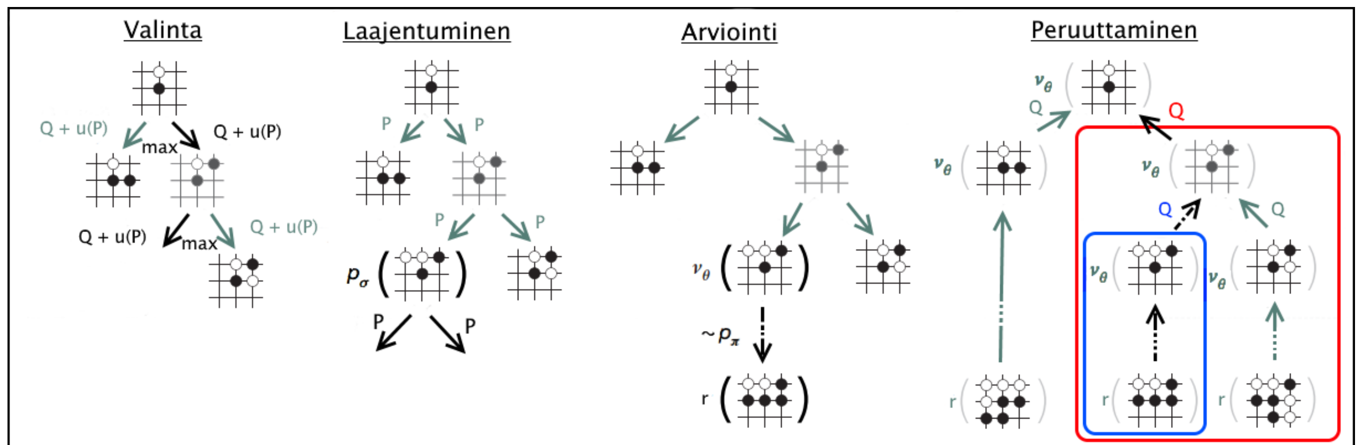
Neuroverkot (engl. neural networks) ovat laskennan malleja, joiden toiminta muistuttaa ihmisaivojen hermoverkkojen toimintaa. Ne ovat yksi koneoppimisen osa-alueista, joten niitä käytetään paljon tekoälyissä. Neuroverkon rakenne on Dayhoffin ja DeLeon (2001) mukaan

usein seuraava: verkko koostuu useista kerroksista, jotka sisältävät **neuroneja**, yksinkertaisia tiedonkäsittely-yksiköitä, jotka suorittavat epälineaarista laskentaa. Neuroneita yhdistävät toisiinsa painotetut yhteydet. Neuroverkkojen opettaminen tarkoittaa sitä, että verkolle annetaan suuria määriä esimerkkidataa, jota käsitellessään verkko muuttaa sen neuroneiden välisten yhteyksien painoarvoja. Siten se voi siis oppia esimerkiksi tunnistamaan tiettyjä toistuvia kuvioita datasta.

AlphaGo hyödyntää toiminnassaan vahvasti neuroverkkoja. Silverin ym. (2016) mukaan se käyttää kolmea eri tarkoituksiin opetettua neuroverkkoa, toimintamalliverkkoa (engl. policy network) p_σ, kevyempää ja huomattavasti nopeampaa versiota siitä, p_π, ja arviointiverkkoa (engl. value network) v_θ. Toimintamalliverkkojen tehtävänä on arvioida annetusta laudan tilanteesta, kuinka todennäköisesti huipputaitoinen ihmispelaaja pelaisi minkäkin siirron (Silver ym. 2016). Ne siis palauttavat todennäköisyysjakauman laillisten siirtojen todennäköisyydestä (Silver ym. 2016). Hitaampi verkko p_σ pystyy ennustamaan ammattilaispelaajien siirrot 57.0 % tarkkuudella kolmessa millisekunnissa, nopeammalta verkolta p_π aikaa menee vain $2\mu s$, tosin tarkkuus putoaa 24.2 prosenttiin (Silver ym. 2016). Muiden tutkimusryhmien tulokset ovat yltäneet parhaimmillaan 44.4 % tarkkuuteen (Silver ym. 2016). Arviointiverkon v_θ tarkoitus puolestaan on ennustaa annetusta pelitilanteesta sen voittaja (Silver ym. 2016). Silverin ym. mukaan verkkojen opettamiseen käytettiin niin ohjattua oppimista kuin vahvistamisoppimista.

4.2 AlphaGo:n toiminta

Tässä käsittelen, kuinka AlphaGo kokonaisuudessaan toimii, eli miten toimintamalliverkot, arviointiverkko ja Monte Carlo -puuhaku toimivat yhteen ohjelman omassa hakualgoritmissa. MCTS tarjoaa hakualgoritmille rakenteen, jota neuroverkkojen tehokas toiminta tukee. Tätä AlphaGo:n hakualgoritmia kutsutaan nimellä APV-MCTS (engl. asynchronous policy and value MCTS algorithm) (Silver ym. 2016). Kuvassa 5 näkyy algoritmin päävaiheet.



Kuvio 5. APV-MCTS:n päävaiheet (mukaillen Silver ym. 2016).

Jokainen pelipuun solmu s sisältää lähtevät kaaret kaikille laillisille siirroille (Silver ym. 2016). Kaari (s,a) sisältää toiminta-arvon $Q(s,a)$, vierailujen määrän $N(s,a)$ ja edellisen todennäköisyyden $P(s,a)$ (Silver ym. 2016). Valintavaiheessa pelipuuta laskeudutaan juuresta, eli nykyisestä pelitilanteesta, simuloidusti alaspäin, valiten siirtoja, joissa maksimoidaan toiminta-arvon ja bonuksen $u(s,a)$ suuruus (Silver ym. 2016). Bonus riippuu siirron edellisestä todennäköisyydestä, mutta huononee käyntien myötä, eri siirtojen kokeilun lisäämiseksi (Silver ym. 2016).

Kun saavutetaan lehtisolmu, voidaan laajentumisvaiheessa lisätä uusi lehtisolmu (Silver ym. 2016). Nyt kuvaan astuvat neuroverkot. Toimintamalliverkko p_σ prosessoi uuden solmun vain kerran ja sen tuottamat todennäköisyydet tallennetaan jokaisen tilanteesta lähtevän laillisen siirron aiempiin todennäköisyyksiin (Silver ym. 2016). Sen jälkeen muodostetaan lehtisolmulle arvo hyödyntäen kahta eri tapaa: Ensin arviointiverkko v_θ arvioi tilanteen, jonka jälkeen suoritetaan simulaatio tilanteesta pelin loppuun käyttäen nopeampaa toimintamalliverkkoa p_π (Silver ym. 2016). Lopuksi nämä arvioinnit yhdistetään käyttäen yhdistysparametria λ, muodostaen solmun lopullisen arvion (Silver ym. 2016).

Simulaation lopuksi päivitetään kaikkien kuljettujen kaarien toiminta-arvot ja käyntien määrät (Silver ym. 2016). Kaaren toiminta-arvo $Q(s,a)$ muodostuu siis kaikkien sen läpi kuljettujen simulaatioiden keskimääräisestä arvosta. Haun päätyttyä algoritmi valitsee nykyisestä tilanteesta lähtevistä siirroista sen, jota käytettiin eniten (Silver ym. 2016).

Silverin ym. (2016) mukaan neuroverkkojen hyödyntäminen vaatii monia kertaluokkia suurempaa laskentatehoa kuin perinteiset hakuheuristiikat, jonka takia AlphaGo käyttää asynkronista, monisäikeistä hakua simulaatioiden suorittamiseen useilla prosessoreilla, ja se laskee toimintamalli- ja arviointiverkot rinnakkain käyttäen grafiikkaprosessoreita (Silver ym. 2016). Samojen pelipuiden yhtäaikaista läpikäyntiä ehkäistään virtuaalitappio-tekniikalla (ks. luku 3.3.2)(Silver ym. 2016). Lopullisessa AlphaGo:ssa käytettiin 40 eri hakusäiettä, 48 eri prosessoria, sekä 8 grafiikkaprosessoria (Silver ym. 2016). Ohjelmasta tehtiin myös hajautettu versio, joka käytti useita tietokoneita, 40 hakusäiettä, 1 202 prosessoria ja 176 grafiikkaprosessoria (Silver ym. 2016). Hajautettu AlphaGo oli selvästi normaalia versiota vahvempi, voittaen 77 % peleistä sitä vastaan (Silver ym. 2016).

5 Yhteenveto

Tässä työssä kävin aluksi läpi go-lautapelin säännöt, jonka jälkeen esittelin Monte Carlo -puuhakua ja sen algoritmista rakennetta. Lopuksi käsittelin itse AlphaGo:ta, ja sitä miten MCTS-algoritmin tarjoamaa rakennetta hyödynnetään sen omassa hakualgoritmissa, APV-MCTS:ssa.

Go-pelissä on yksinkertaiset säännöt, mutta mahdollisten siirtojen määrä ja hyvien siirtojen erottamisen vaikeus tekevät siitä tietokoneelle todella vaikean pelin (Müller 2002). Tietokone-go:n juuret ovat 1960-luvulla, mutta vasta 1980-luvun turnausten ansiosta go-ohjelmien kehitys parani ja tuloksia alkoi näkyä voittojen muodossa (Müller 2002). Silti kesti yli 50 vuotta kehittää tekoäly, joka pystyi voittamaan ammattilaispelaajan.

Monte Carlo -puuhaku on satunnaisuutta ja simulaatioita hyödyntävä hakualgoritmi, jota ovat vuodesta 2006 asti käyttäneet monet menestyneimmistä go-ohjelmista. Se on rakenteeltaan nelivaiheinen (Chaslot 2010): valintavaiheessa puuta edetään lehtisolmuun asti rekursiivisesti, jonka jälkeen laajentumisvaiheessa lisätään uusi lapsisolmu. Simulaatiovaiheessa pelataan lisätystä lapsisolmusta peli loppuun käyttäen Monte Carlo -simulaatiota, eli käytännössä valitsemalla pääosin satunnaisia siirtoja. Pelin loputtua tulokset päivitetään pelipuun solmuihin peruutusvaiheessa. Näitä vaiheita toistetaan niin kauan kuin aikaa riittää, jonka jälkeen valitaan useimmin pelattu siirto.

AlphaGo:n hakualgoritmin toiminta perustuu MCTS:n rakenteeseen, mutta pelitilanteiden arviointiin käytetään myös useita opetettuja neuroverkkoja, joita myös kävin lyhyesti läpi. Työn pääpainoksi valitsin MCTS:n. Koneoppiminen ja neuroverkot tarjoavat todella laajan tutkimusalueen itsessään, ja ovat selvästi yksi syy siihen, miksi AlphaGo on niin hyvä pelaaja. Toinen vaihtoehto tutkielmaan olisi ollutkin tutkia toimintaa juuri niiden näkökulmasta ja perehtyä syvällisemmin aiheeseen.

Lähteet

B, Jan. 2006. "Go Regeln 3.png". Public Domain, muokattu. Viitattu 4. huhtikuuta 2017. https://commons.wikimedia.org/w/index.php?curid=1267777.

Bouzy, Bruno, ja Tristan Cazenave. 2001. "Computer Go: An AI oriented survey". *Artificial Intelligence* 132 (1): 39–103. ISSN: 0004-3702. http://www.sciencedirect.com/science/article/pii/S0004370201001278.

Brügmann, Bernd. 1993. *Monte carlo go.* Tekninen raportti. Citeseer.

Champandard, Alex J. 2014. "AIGameDev, Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI". Viitattu 3. 22 2017. http://aigamedev.com/open/coverage/mcts-rome-ii/.

Chaslot, Guillaume. 2010. "Monte-carlo tree search". *Maastricht: Universiteit Maastricht.*

Chaslot, Guillaume M. J. -B., Mark H. M. Winands ja H. Jaap van den Herik. 2008. "Parallel Monte-Carlo Tree Search". Teoksessa *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings,* toimittanut H. Jaap van den Herik, Xinhe Xu, Zongmin Ma ja Mark H. M. Winands, 60–71. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-87608-3. http://dx.doi.org/10.1007/978-3-540-87608-3_6.

Coulom, Rémi. 2006. "Efficient selectivity and backup operators in Monte-Carlo tree search". Teoksessa *In: Proceedings Computers and Games 2006.* Springer-Verlag.

Dayhoff, Judith E., ja James M. DeLeo. 2001. "Artificial neural networks". *Cancer* 91 (S8): 1615–1635. ISSN: 1097-0142. http://dx.doi.org/10.1002/1097-0142(20010415)91:8+%3C1615::AID-CNCR1175%3E3.0.CO;2-L.

Ertel, Wolfgang, Johann M. Ph. Schumann ja Christian B. Suttner. 1989. "Learning Heuristics for a Theorem Prover using Back Propagation". Teoksessa *5. Österreichische Artificial-Intelligence-Tagung: Igls/Tirol, 28.-31. März 1989 Proceedings,* toimittanut Johannes Retti ja Karl Leidlmair, 87–95. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-74688-8. http://dx.doi.org/10.1007/978-3-642-74688-8_10.

Gelly, Sylvain, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári ja Olivier Teytaud. 2012. "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions". *Commun. ACM* (New York, NY, USA) 55, numero 3 (maaliskuu): 106–113. ISSN: 0001-0782. http://doi.acm.org/10.1145/2093548.2093574.

Gelly, Sylvain, ja David Silver. 2008. "Achieving Master Level Play in 9 x 9 Computer Go." Teoksessa *AAAI,* 8:1537–1540.

———. 2011a. "Monte-Carlo tree search and rapid action value estimation in computer Go". *Artificial Intelligence* 175 (11): 1856–1875. ISSN: 0004-3702. http://www.sciencedirect.com/science/article/pii/S000437021100052X.

———. 2011b. "Monte-Carlo tree search and rapid action value estimation in computer Go". *Artificial Intelligence* 175 (11): 1856–1875. ISSN: 0004-3702. http://www.sciencedirect.com/science/article/pii/S000437021100052X.

Hammersley, John. 2013. *Monte carlo methods.* Springer Science & Business Media.

Kocsis, Levente, Csaba Szepesvári ja Jan Willemson. 2006. "Improved monte-carlo search". *Univ. Tartu, Estonia, Tech. Rep* 1.

Kohavi, Ron, ja Foster Provost. 1998. "Glossary of terms". *Machine Learning* 30 (2-3): 271–274.

Lindström, Jonatan. 2005. "Go - atari.png". CC BY-SA 3.0. Viitattu 4. huhtikuuta 2017. https://commons.wikimedia.org/w/index.php?curid=611842.

Lison, Pierre. 2015. *An introduction to machine learning.*

Lubberts, Alex, ja Risto Miikkulainen. 2001. "Co-evolving a go-playing neural network". Teoksessa *Proceedings of the GECCO-01 Workshop on Coevolution: Turning Adaptive Algorithms upon Themselves,* 14–19.

Metropolis, Nicholas. 1987. "The Beginning of the Monte Carlo Method". Viitattu 3. 25 2017. http://library.lanl.gov/cgi-bin/getfile?00326866.pdf.

Müller, Martin. 2002. "Computer Go". *Artificial Intelligence* 134 (1): 145–179. ISSN: 0004-3702. doi:http://doi.org/10.1016/S0004-3702(01)00121-7.

Niekerk, Francois van, Gert-Jan van Rooyen, Steve Kroon ja Cornelia P. Inggs. 2012. “Monte-Carlo Tree Search Parallelisation for Computer Go”. Teoksessa *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference,* 129–138. SAICSIT ’12. Pretoria, South Africa: ACM. ISBN: 978-1-4503-1308-7. http://doi.acm.org/10.1145/2389836.2389852.

philosopher, Mystic. 2006. “Go game example.png”. CC BY-SA 3.0. Viitattu 4. huhtikuuta 2017. https://commons.wikimedia.org/w/index.php?curid=656367.

Russell, Stuart, Peter Norvig ja Artificial Intelligence. 1995. “Artificial Intelligence: A modern approach”. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25:27.

Sensei’s Library. 2017a. “Sensei’s Library, Basic rules of Go”. Viitattu 3. maaliskuuta 2017. http://senseis.xmp.net/?BasicRulesOfGo.

———. 2017b. “Sensei’s Library, History of Go”. Viitattu 3. maaliskuuta 2017. http://senseis.xmp.net/?GoHistory.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot ym. 2016. “Mastering the game of Go with deep neural networks and tree search”. *Nature* 529 (7587): 484–489.

Wang, F. Y., J. J. Zhang, X. Zheng, X. Wang, Y. Yuan, X. Dai, J. Zhang ja L. Yang. 2016. “Where does AlphaGo go: from church-turing thesis to AlphaGo thesis and beyond”. *IEEE/CAA Journal of Automatica Sinica* 3, numero 2 (huhtikuu): 113–120. ISSN: 2329-9266. doi:10.1109/JAS.2016.7471613.

Ward, C. D., ja P. I. Cowling. 2009. “Monte Carlo search applied to card selection in Magic: The Gathering”. Teoksessa *2009 IEEE Symposium on Computational Intelligence and Games,* 9–16. Syyskuu. doi:10.1109/CIG.2009.5286501.